———————————————————————————————————————

Event-Action Protocol Pseudocode:

1. recv[CLIENT_REQUEST, m]:

```
if WRITE_OPERATION:
        if server_id == LEADER_ID
                propagate[MULTICAST_COMMIT, m]
        else
                prepare MULTICAST_REQUEST message  from server_id to LEADER_ID
else:
        execute the operation on the server_id immediately
```

2. recv[MULTICAST_REQUEST]

    propagate[MULTICAST_COMMIT, m]

3. recv[MULTICAST_COMMIT, m]

    messageSequenceID++

    m is placed in a Priority Queue in the form of a Timestamped Message (which contains the messageSequenceID).

    processMessagesFromQueue, which is peeking for any message requests in the queue, gets invoked. The method waits until it gets the message with the nextExpectedTimestamp and executes it on the given server.

    nextExpectedTimestamp++

    Once every other server has executed the message, the LEADER_ID server also executes the message.

———————————————————————————————————————

Premise:

Leader Election:

LEADER_ID is decided using loadLeaderFromProperties(), as the first non null server value.

processMessagesFromQueue is invoked as a part of the constructor in MyDBReplicatedServer. It maintains a variable nextExpectedTimestamp to maintain a global ordering of the messages. The method is a ThreadExecutor and constantly listens for any messages in the priority queue.

messageSequenceID maintains the sequence of messages by assigning a unique ID to each message. This sequenceID is attached to each message by creating a JSONObject messageJson, which contains the original message, messageSequenceID and messageType which can either be MultiCastCommit or MultiCastRequest.

Step1:
The client request is first received at handleMessageFromClient(). The method checks if it is a write operation using isWriteOperation(), which if False, the message is executed on the requested server immediately. In case of a write operation, the method first checks if the

current server is the leader or a non-leader. If its a leader, it immediately invokes propagateMultiCastCommit(). Else, it creates a JSONObject messageJson where it stores the message details in a JSON format which includes the actual message request, assigned messageSequenceID and the message type, which in this case is “MultiCastRequest.” This JSONObject is then sent to the leader server, for it to get added to the priority queue of messages and get accepted as a MultiCastCommit to be propagated to all member servers.

Step 2:
For a “MultiCastRequest” that was sent from a non-leader server to the leader server, the message is processed in handleIncomingServerMessage(). Here, we implement a synchronized lock to ensure thread safety. Here, the “MultiCastRequest” is then forwarded to the leader server using propagateMultiCastCommit()

Step 3:
When a “MultiCastCommit”, received by the leader server, the messageSequenceID is incremented. This is then sent to all other member servers, with the message type as “MultiCastCommit”, using serverMessenger.sendToID which internally calls handleIncomingServerMessage(). On receiving this message at handleIncomingServerMessage(), the message is added to a priorityQueue as a TimestampedMessage. This is then picked up by processMessagesFromQueue(), which as explained before, constantly listens for any active message requests in the priorityQueue. However, processMessagesFromQueue() waits to execute the message request until it has timestamp that matches its nextExpectedTimestamp, in which case, it polls the message request from the queue and executes it. This is done until all servers execute the message, after which controls flows back the propagateMultiCastCommit() where the leader server finally executes the message. In this way, we ensure that all servers execute a given “write” operation request message in a global order, using messageSequenceID, priority queues and nextExpectedTimestamp.

Handeling CallBack :

When the leader multicasts the message all the messages received by the non leader are placed in a priority Queue where the priority is the low timestamp or the messagesequenceID, when the expected messageID matches the queues head message id it commits and send back an **ACK** to the Leader server , the leader server maintains a synchronous hash map when the leader server receives a a ack message it increments its hash map value of the key message id. The HashMap has key as messageID and has the count as the value.When the count is increased the method handleACKs will check if the count is equal to the number of nodes - 1 or not if it is then it sends back response to the client appending the id to its response.