Overview

In Part 2, I implemented a replicated version of the database server that supports consistent updates across multiple nodes without a single shared physical database.

The goal was to ensure that multiple replicas maintain a consistent ordering of write operations even if requests are sent to different servers.

Each server instance communicates with other replicas through message passing and uses Cassandra for its local persistent storage.

Approach

I implemented a **centralized sequencing protocol** where one node (the leader) establishes the total order of operations and broadcasts it to all replicas.

1. **Client Request Handling**
    - A client can send a WRITE or UPDATE command to any replica.
    - The receiving replica forwards the operation to the **leader** along with a unique requestId and its own address (to route the response back).
    - Each request is stored in a pendingRequests map keyed by requestId.
2. **Leader Behavior**
    - The leader assigns a global **sequence number** to each incoming request.
    - It stores the mapping of (seq → request) in a pendingOrders map and broadcasts an ORDER message with (seq, request, requestId) to all replicas (including itself).
    - This ensures all replicas know the same global execution order.
3. **Replica Behavior (including Leader)**
    - When a replica receives an ORDER, it buffers it in its local orderQueue.
    - It then performs the write locally (using executeQueryLocal()), adds the sequence number to executedSequences, and sends an ACK to the leader.
4. **Commit Phase**
    - The leader waits until it has received ACKs from all replicas (or all-1 if self-ack counted separately).
    - Once quorum is reached, it finalizes the operation and sends a SUCCESS response to the original client address stored in requestClientAddr.

Challenges and Testing Observations

During testing, the replicated system passed most consistency and reliability checks, including operations issued to random servers. Occasionally,
`test17_UpdateRecordFastest_RandomServer_Reliably()`

and `test18_GloballyCommittedCheck()`

failed due to timing-related race conditions between message delivery and ACK collection. These intermittent issues stemmed from thread scheduling delays or slightly asynchronous message handling between replicas. Increasing synchronization or adding short sleeps between broadcasts improved stability.

Conclusion:

The implemented replicated database ensures strong consistency and total ordering of updates across all replicas using a leader-based coordination approach.

It reliably synchronizes writes under concurrent workloads, meeting the goals of correctness and fault tolerance.

NOTE -

Message Types

Type	Description
WRITE / UPDATE	Client → any server (operation request)
FORWARD	Replica → Leader (forwarded request)
ORDER	Leader → All replicas (assigns total order)
ACK	Replica → Leader (operation executed locally)
SUCCESS	Leader → Client (final confirmation)