Design Document: Custom Protocol Fault-Tolerant Datastore

Ashi Sinha (34743339),

Sameeksha Bhatia (35243160)

This system implements a **fault-tolerant replicated datastore** using a **custom leader-based server-to-server protocol** built directly on top of Cassandra. It does not rely on ZooKeeper, GigaPaxos, or any external coordination service. All coordination happens explicitly between servers.

Overall Idea

At any time, one server acts as the **logical leader**. Clients can send requests to **any** server.

- If a server is the leader, it directly processes the request.
- If it is not the leader, it **forwards the request** to the leader using a simple server-to-server message.

The leader ensures that all updates are applied **in the same order** on every replica so that all replicas stay consistent.

Server Setup and Cassandra Layout

Each server runs an instance of MyDBFaultTolerantServerZK. All servers connect to the **same Cassandra instance**, but each server has its **own keyspace** (e.g., server0, server1, server2, etc.).

For each keyspace, the server creates:

- grade(id, events) – the main table
- grade_backup(id, events) – a checkpoint copy of grade
- oplog(op_id, cmd) – durable operation log
- applied_ops(op_id) – tracks which operations have been applied

Each server keeps a Session per keyspace so that the leader can update **all replicas** by iterating over these sessions.

Leader Selection

Leader selection is **deterministic and simple**:

1. All server IDs are sorted lexicographically.
2. The smallest ID is chosen as the initial leader.

Each server stores the ID of the leader it currently believes is active. If communication with the leader fails, the server **moves to the next server ID** in round-robin order and retries.

This avoids complex leader-election protocols while still allowing recovery from crashes.

Client Request Handling

Client commands are processed in handleMessageFromClient.

INSERT Commands

For commands of the form:

INSERT INTO grade ...

- Any server can execute them directly.
- The command is applied to **all replicas immediately**.
- The inserted row is also copied into grade_backup.

This makes inserts safe even if the leader is down.

UPDATE Commands

For commands like:

UPDATE grade SET ...

- If the server is the leader → it applies the command.

If the server is not the leader → it forwards the command to the leader using:

```
FWD:<CQL command>
```

- The leader acts as the **single serialization point**, ensuring all replicas observe updates in the same order.

Handling Non-Deterministic List Updates

Cassandra's list append operation:

```
events = events + [X]
```

is **non-deterministic** under concurrent updates and may produce different orders on different replicas.

To fix this, the leader detects updates of the form:

```
UPDATE grade SET events=events+[X] WHERE id=Y;
```

Instead of executing this directly, it performs a **deterministic read-modify-write** on every replica:

1. Read the current events list.
2. Add X to the list.
3. Sort the list.
4. Write the sorted list back to grade.
5. Write the same list to grade_backup.

Because every replica performs the same steps, all replicas end up with **identical event lists**, regardless of timing.

Operation Logging for Fault Tolerance

For **non-INSERT commands**, the leader assigns a unique op_id and performs the following steps on every replica:

1. Insert (op_id, cmd) into oplog

2. Execute the command
3. Insert op_id into applied_ops

This ensures that:

- Every update is durably recorded
- Servers can detect which updates they missed while down

Crash Recovery

When a server restarts, recovery happens in two stages:

1. Restore from Backup

If grade is empty but grade_backup contains data:

- The server reconstructs grade from grade_backup

2. Replay Missing Operations

The server:

- Reads all entries from oplog
- Skips operations already listed in applied_ops
- Replays missing operations in increasing op_id order

This guarantees that a restarted server catches up to the latest state.

Leader Failure Handling

If a server tries to forward a command to the leader and gets an IOException:

1. It assumes the leader has crashed
2. It advances the leader to the next server ID

3. It retries forwarding

If the leader eventually becomes the current server, it applies the command locally. This provides **automatic leader failover** without external coordination.

- All updates are **serialized at the leader**
- Non-commutative list updates are made **deterministic**
- All durable state is stored in Cassandra
- Servers can recover independently after crashes