Distributed Database System with Logical Timestamps, Adaptive Quorums, and Performance-Aware Failover

1. Problem Description

When you are working with distributed databases, you want consistency and availability even when things go wrong, like when the master database crashes or a replica falls behind. But in practice, most systems either rely on slow failover scripts or use centralized coordination points that do not scale well. Take AWS RDS, for example. It runs with a single master and a few read replicas. Writes only go to the master, and replication is asynchronous. If the master crashes, failover takes a while, and there is no guarantee your replicas are consistent when you read from them.

What we are trying to build is a system that does better. We are using MySQL as the base, but we are layering it with three key ideas: logical timestamps to keep writes in order, a smarter quorum system that favors the faster replicas, and a leader election process that picks the most stable and responsive replica if the master dies. This should give us a system that handles failure better, responds faster, and still keeps things consistent.

2. Motivation

The kind of systems people rely on today, like standard MySQL setups or cloud-managed databases, do not always do well when something goes wrong. The master might crash, and it takes time for another node to take over. Replicas may not be up to date. And while replication happens, there is no guarantee when a write has really been seen by enough nodes.

In our design, writes go to a single master just like before, and the data is replicated to all the replicas. But we do not wait for all of them to respond. We pick a quorum, the replicas that are currently performing the best, and only wait for them. This cuts down on delay without sacrificing safety. We also use timestamps to keep track of write order, and we check those timestamps before serving reads from a replica. If the master goes down, we have a way to rank replicas based on their performance and stability, so a new leader can be promoted automatically. This is the kind of system we would want running in a real cloud environment.

3. Literature Review

Each of us team members focused on a specific area: timestamps, consistency and replication, or leader election. We read six recent peer-reviewed papers that helped us design different parts of the system.

Timestamps

1. NCC: Natural Concurrency Control (OSDI 2023) - Link
Talks about avoiding timestamp inversion using smart retry logic and delayed responses. Helps keep serializability without needing a central timestamp lock.

2. Timestamp as a Service, Not an Oracle (PVLDB 2023) - Link
Instead of having one timestamp server for the whole system, they use several. That way, no single failure point, and timestamps still stay globally ordered.

Consistency and Replication

3. Cabinet: Dynamically Weighted Consensus (arXiv 2025) - Link
Instead of fixed quorums, this one lets you weigh replicas based on how fast and reliable they are. You do not have to wait for slow nodes if enough good ones respond.

4. A Unified Summary of Non-Transactional Consistency Levels (arXiv 2024) - Link
Gives a framework to define and reason about consistency levels, like linearizability, eventual consistency, and bounded staleness.

Leader Election

5. SEER: Performance-Aware Leader Election (arXiv 2021) - Link
When it is time to elect a new leader, this one uses latency predictions to pick the fastest node, instead of choosing randomly or by ID.

6. A Hierarchical Adaptive Leader Election Algorithm (LADC 2024) - Link
Adds crash history to the equation. Nodes that have failed often are less likely to become leaders.

4. Methodology and Architecture

Our system is made up of several components that work together to handle client requests, maintain consistency, and tolerate failures.

When a client sends a write, the router is the first stop. It talks to our timestamp service, which is spread across multiple servers, and gets a unique, globally ordered timestamp. This avoids using a single central timestamp server, which would be a bottleneck. Once it gets the timestamp, the router forwards the write along with the timestamp to the master, which is a MySQL instance.

The master applies the write and sends it to all replicas. But we do not wait for all of them to respond. We use a quorum, picked based on current performance metrics, and only wait for those replicas to confirm. This gives us a balance between speed and safety. If enough good replicas say they have received the write, we consider it successful.

For reads, the router again checks in with the replicas. It looks at each replica's latest applied timestamp. If the replica is up to date, it can serve the read. If not, the router waits or tries a different one. This gives us flexible consistency. We can support strong reads or stale but fast reads.

If the master crashes, we evaluate all replicas based on their performance and stability metrics. The replica with the highest combined score is promoted to master, and the remaining components update their configurations to follow the new leader.

5. Milestone Plan

Timeline	Milestone
Oct 9 to 15	Finalize design and define APIs
Oct 16 to 22	Build timestamp service, integrate with router
Oct 23 to 31	Add quorum-based replication with performance weights
Nov 1 to 10	Implement leader election logic (SEER and stability)
Nov 11 to 20	Run failure simulations, consistency tests, read freshness checks
Nov 21 to 30	Final testing, reporting, presentation demo

6. Resources Needed

- Docker to run MySQL replicas and services
- Python (Flask or FastAPI) for router and timestamp services
- Cloud Run for deployment and scaling
- Google Cloud Monitoring for tracking latency and replica metrics

7. References

- NCC: https://arxiv.org/pdf/2305.14270
- Timestamp as a Service: https://www.vldb.org/pvldb/vol17/p994-li.pdf
- Cabinet: https://arxiv.org/abs/2503.08914
- Consistency Levels: https://arxiv.org/pdf/2409.01576v1
- SEER: https://arxiv.org/abs/2104.01355
- Hierarchical Election: https://dl.acm.org/doi/10.1145/3697090.3697102