DDIA Notes Chapter -- Replication

Replication is where copies of the same data are maintained on multiple machines (replicas). Replication allows a system to:

- **Reduce latency.** Keep data geographically close to users.
- **Increase availability.** Allow the system to continue to function when some parts have failed.
- **Increase read throughput.** Scale the number of machines that can serve read queries.

The challenge of replication is in handling changes to the data. There are three main approaches to replication: single-leader replication, multi-leader replication, and leaderless replication.

Single-Leader Replication

In single-leader replication, clients send all writes to a single leader node, which communicates changes to all replica follower nodes. Clients can read from any replica. Examples of systems that use this mode include relational databases (MySQL, PostgreSQL, Oracle, SQL Server), some nonrelational databases (MongoDB), and message brokers (Kafka, RabbitMQ).

New Followers

Systems will periodically take a snapshot of the leader's database. This snapshot is used to create the new follower node. The leader will maintain a position in its replication log. The position is also known as the log sequence number (PostgreSQL) or binlog coordinates (MySQL). The follower uses this position to ask the leader for all changes that have happened since the last snapshot. When a node goes down, this same method is used to restore a node.

Leader Failover

Refer to the notes on Raft Distributed Consensus.

Synchronous vs. Asynchronous Replication

In synchronous replication, the leader waits until the follower has confirmed that it received the write before reporting success to the client and making the write visible to other clients. In asynchronous replication, the leader sends the write to the

follower but doesn't wait for a response from the follower.

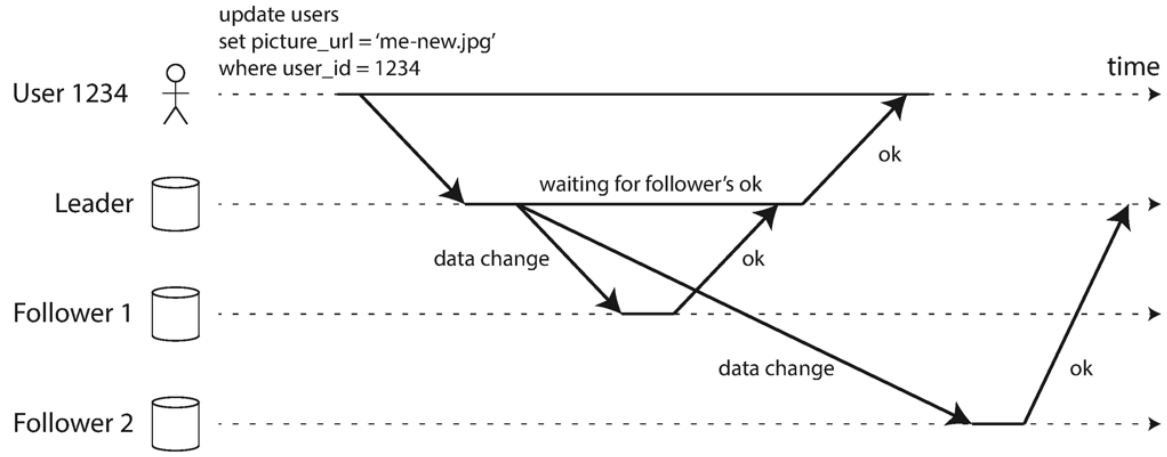

Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

It is impractical for all followers to be synchronous as one node outage would cause the whole system to stall. In practice, only one of the followers is synchronous while the others are asynchronous. If the synchronous follower becomes slow or unresponsive, one of the asynchronous followers is changed to be synchronous. At least two nodes are guaranteed to have an up-to-date copy of the data (semi-synchronous). Fully asynchronous systems are also used in practice.

Eventual Consistency

The delay between a write on the leader to an update on a follower is known as replication lag. In most cases, this lag is only a fraction of a second. This temporary inconsistency is known as eventual consistency.

Read-After-Write Consistency

Read-after-write consistency (or read-your-writes consistency) is a guarantee that clients will always see their own updates. This is implemented as follows:

1. When a client sends read requests for data it previously modified, the leader will handle the request.
2. If the time between the read request and the last update is within one minute, it is considered as previously modified.
3. For more fine-grained control, monitor the replication lag on followers. If the time between the read request and the last update is within the replication lag window, the leader will handle the request.

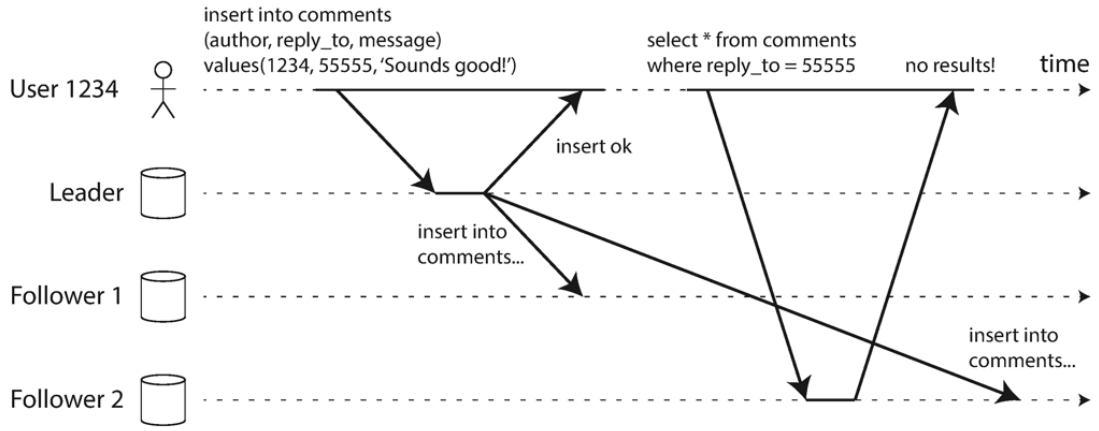

Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

Monotonic Reads Consistency

When reading from multiple asynchronous followers, a client can see data moving backwards in time. Monotonic reads consistency prevents this by ensuring each client processes its reads from the same replica. This can be accomplished by routing clients to replicas using a hash of its client/user ID. If the replica fails, a client's queries are rerouted to a different replica.

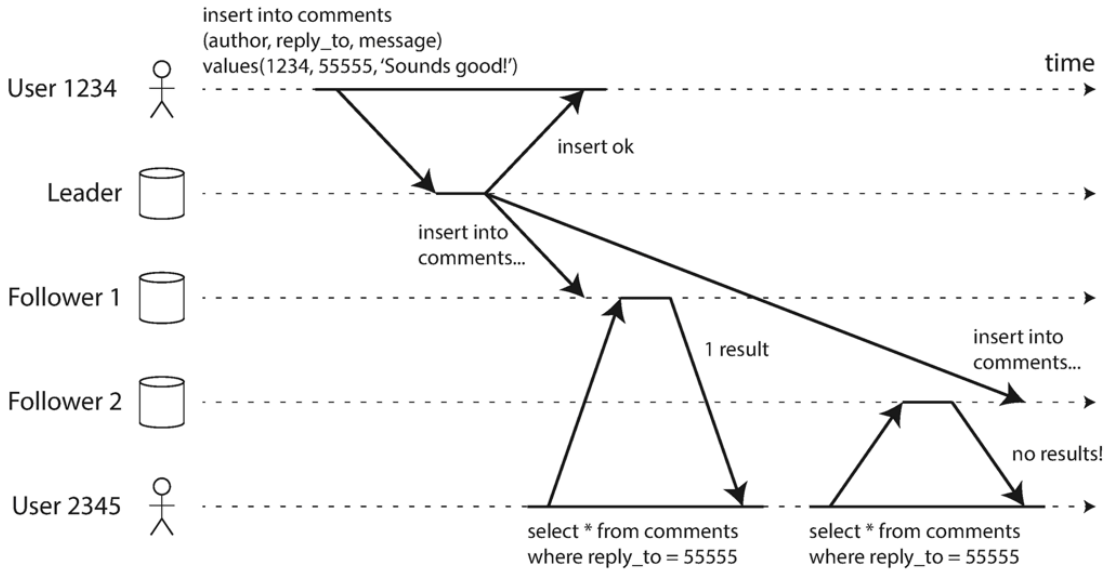

Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

Multi-Leader Replication

In multi-leader replication, clients can send writes to one of several leader nodes, which communicate changes to the other leader nodes and all replica follower nodes. This mode is for expanding beyond a single datacentre. Multi-leader replication does not make sense within a single datacentre due to the added complexity.

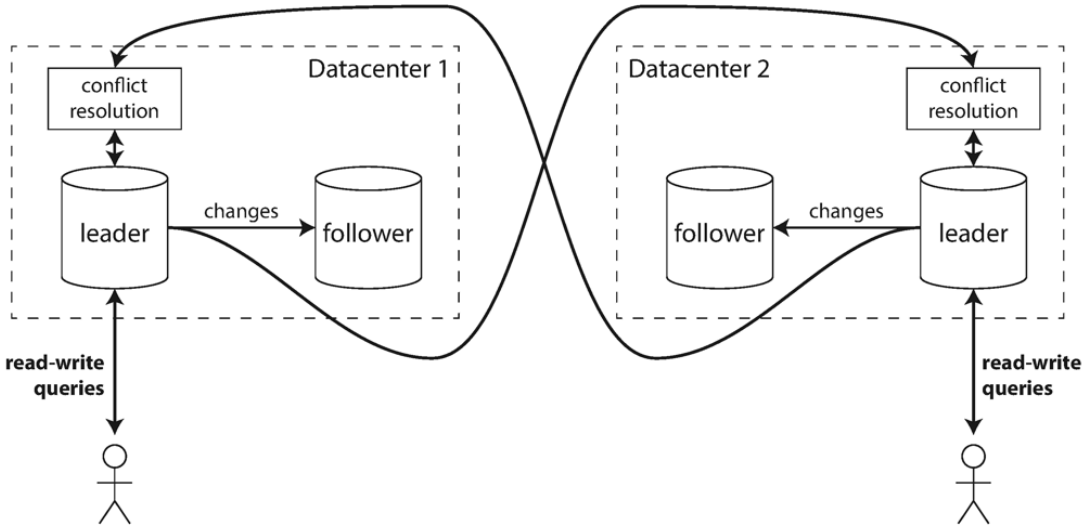

Figure 5-6. Multi-leader replication across multiple datacenters.

Conflict Resolution

With multi-leader replication, write conflicts become a problem--two writes concurrently modify the same object. The system needs to detect conflicts and resolve them. The following are different methods for resolving conflicts. These methods resolve conflict at the individual row or document level.

1. **Conflict Avoidance.** Many systems choose to avoid conflicts all together. This is accomplished by ensuring that all writes for a particular object go through the same leader. In systems where users are primarily editing their own data, each user can be assigned a home datacentre--this is typically the datacentre that is geographically closest to the user. Avoiding conflicts is the most recommended approach.
2. **Last Write Wins (LWW).** Each write is given a unique ID or timestamp. The write with the highest ID or most recent timestamp wins, throwing away any other writes.
3. **Merge Writes.** The system can merge the write together e.g., order alphabetically and then concatenate the writes.
4. **Custom Conflict Resolution Logic.** The system provides users with the means to write their own conflict resolution logic using application code. One

approach is for the application to automatically resolve the conflict. Another approach is for the application to prompt the client with options on how to resolve the conflict.

Leaderless Replication

In leaderless replication, clients send each write to multiple nodes. Clients read from multiple nodes in parallel and if a discrepancy is detected, it will correct the nodes with stale data. This mode was popularized by Amazon Dynamo--Riak and Cassandra are database systems inspired by Dynamo.

Read Repair

When a client detects discrepancies in reads from multiple nodes, it writes the up-to-date data to the replica with the stale data. Version numbers are used to determine which data is newer.

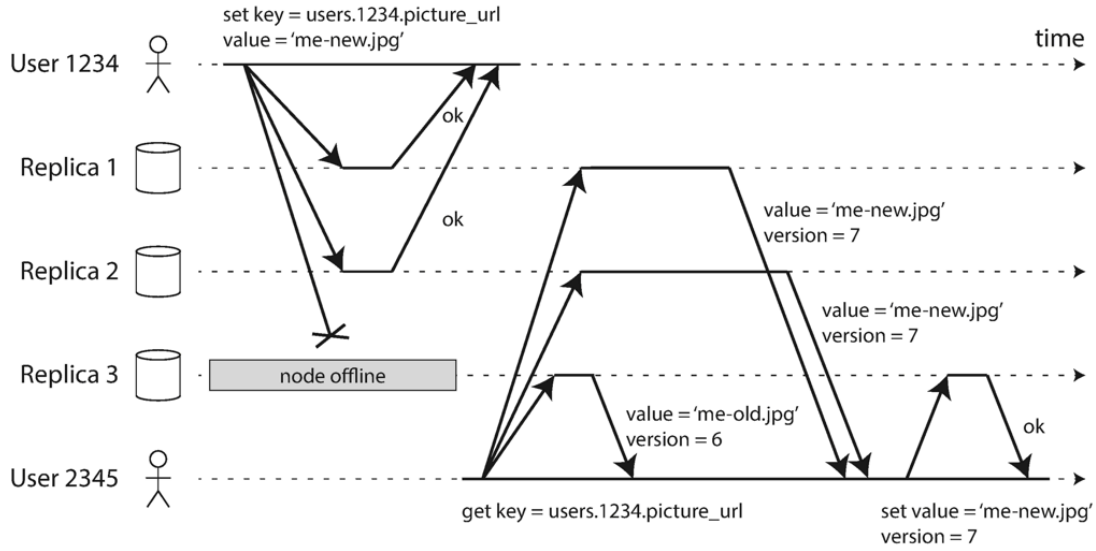

Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

Anti-Entropy Process

Systems have a background process that periodically checks for differences between replicas and resolve them accordingly.

Quorum Consistency

For *N* replicates, every write must be confirmed by at least *W* nodes while every read must be confirmed by at least *R* nodes. To guarantee up-to-date data, we must have `W + R > N`. This ensures that at least one *R* node is up-to-date. *N* is an odd number (typically 3 or 5).

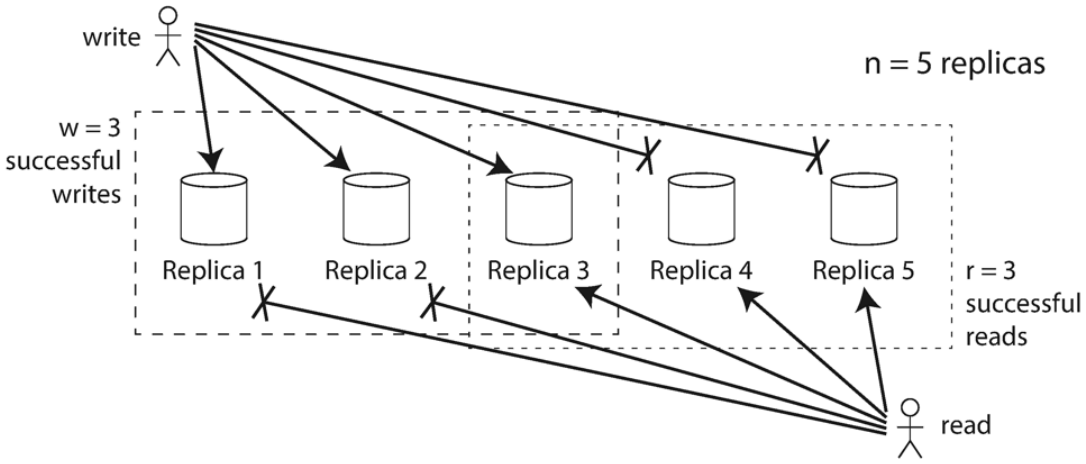

Figure 5-11. If w + r > n, at least one of the r replicas you read from must have seen the most recent successful write.

This mode remains available with node failures due to the conditions `W < N` and `R < N`. Reads and writes are always sent to all *N* replicas in parallel. The client waits for at least *W* or *R* nodes to respond before a write or read is considered successful.