Efficient Code

Abhishek Dey

Leaderless Replication: Dynamo-style, Quorum Consensus, Eventual Consistency, High Availability, Low Latency

DECEMBER 25, 2017DECEMBER 25, 2017 / EFFICIENTCODEBLOG

Single-leader and ***multi-leader replication*** are based on the idea that a client sends a write request only to one of the ***leader nodes*** and then the database system takes care of copying that write to the other replicas. **A leader determines the order in which writes should be processed, and followers apply the leader's writes in the same order.**

However, some data storage systems take a different approach, **abandoning the concept of a leader and allowing any replica to directly accept writes from clients**. *Amazon* used this **leaderless architecture** for its in-house ***Dynamo*** system. That's why ***leaderless replication*** is often called ***Dynamo-style***.

> *Facts:*
> **Dynamo-Style:** Leaderless Architecture
> **Amazon DynamoDB:** Single-Leader Architecture.

In some leaderless implementations

- the client directly sends its writes to several replicas,
- while in others, a ***coordinator node*** sends the writes to the replicas on behalf of the clients. Unlike a leader-based replication, the coordinator does not enforce a particular ordering of writes. And this difference in design has profound consequences for the way the leaderless architecture is used.

> **Leaderless Replication does NOT enforce particular ordering of writes.**

Quorum Consensus

Handling of a Node (Leader in a leader-based configuration) Going Down:

In a leader-based configuration, when a leader is down we need to perform a ***failover*** in order to continue processing writes.

On the other hand, in a **leaderless configuration** *failover* **does not exist.** The client **sends the write to all the replicas** and the available replicas accept the write and the unavailable replica misses it. If according to the configured ***quorum consensus*** (which we will discuss shortly in details) it's sufficient for the available replicas to acknowledge the write, then we consider the write to be successful, even though one replica is unavailable and did not get the write. The client simply ignores the fact that one of the replicas missed the write.
Now if the unavailable node comes back online, and the clients start reading from it then clients might potentially get stale data since any write that happened while the node was down are missing from the node. To solve this problem, when a client reads from the database, it doesn't just send its read request to one replica, rather the client sends the read requests to several replicas in parallel. The client may get different responses from different replicas, i.e. the up-to-date value from some nodes and stale value from another. In this scenario, **version numbers** are used to determine which value is newer.

> In Leaderless architecture **failover** does not exists.
>
> **Failover:** In a leader-based architecture, when a leader node fails:
> 1. One of the followers need to be promoted to be the new leader,
> 2. Clients need to be reconfigured to send their writes to the new leader,
> 3. The other followers need to start consuming data changes from the new leader.
> This whole process of handling a leader node failure in a leader-based architecture is called ***failover.***

The replication scheme must ensure that ***eventually*** all the data is copied to every replica. After an unavailable node comes back online, **two mechanisms are often used in *Dynamo-style datastores* to catch up on the missed writes:**

1. ***Read Repair:***
   When a client makes a read from several nodes in parallel, it can detect any stale responses using ***version numbers***. When a client sees that a replica has a stale value, **it writes the newer value back to that replica.**

   > ***Read Repair*** **mechanism works very well for values that are frequently read.**

2. ***Anti-Entropy Process:***
   In addition to ***Read Repair***, some datastores have a **background process** that constantly looks for differences in the data between replicas and copies any missing data from one replica to another. This process is called ***Anti-entropy process.***

Unlike the replication log in leader-based replication, **anti-entropy process process does not copy writes in any particular order**, and there may be a significant delay before the data is copied.

Point to be noted that not all systems implement both of ***Read Repair*** and ***Anti-entropy Process.*** Note that **without an anti-entropy process, values that are rarely read may be missing from some replicas and thus have *reduced durability*, because read repair is only performed when a value is read by the application.**

Quorum Consensus says, if there are ***n*** replicas, every write must be confirmed by ***w*** nodes to be considered as successful, and we must query at least ***r*** nodes for each read. As long as $w + r > n$, we expect to get an up-to-date value when reading, because at least one of the ***r*** nodes we are reading from must be up-to-date. Reads and writes that obey ***r*** and ***w*** values are called ***quorum*** reads and writes. We can think of ***r*** and ***w*** as the number of votes required for the read and write to be valid.

What matters in *Quorum Consistency* is that the nodes used in reads and writes have at least one node in common (i.e., overlapping).

Quorum Consensus:

At least one node needs to overlap :

$$w >= (n - r) + 1$$

$$r >= (n - w) + 1$$

which gives,

$$w > n - r$$

$$r > n - w$$

which gives,

$$w + r > n$$

In *dynamo-style* databases, the parameters *n, w* and *r* are typically configurable. A common choice is to make ***n*** an **odd number** (typically 3 or 5) and to set

$$w = r = (n + 1) / 2$$

A workload with few writes and many reads may benefit from setting

$$w = n$$

$$r = 1$$

This makes the read faster, but has the disadvantage that just one failed node causes all databases writes to fail.

Limitations of Quorum Consistency:

Higher Latency – Lower Availability

Quorum Consistency can sometimes lead to **higher latency** and **lower availability** if **proper caution is not taken** and there is a **network interruption** and many replicas become unreachable. This is because, **if the number of reachable replicas falls below *w* or *r*, the database would become unavailable for writing or reading**.

Even with ***w* + *r* > *n***, there are likely to be edge cases where stale values are returned. These depend on the implementation, but possible scenarios are:

- **Sloppy Quorum:**
  If a ***sloppy quorum*** is used, the ***w*** writes may end up on different nodes than the ***r*** read nodes, so there is no longer a guaranteed overlap between the ***r*** read nodes and the ***w*** write nodes.
- **Concurrent Writes:**
  If two writes occur concurrently, it is not clear which one happened first. In this case, the only safe solution is to merge the concurrent writes. If a winner is picked based on timestamp, writes can be lost due to clock skew.
- **Concurrent Read And Write:**
  If a write happens concurrently with a read, the write may be reflected on only some of the replicas. In this case, it's undetermined whether the read returns the old or new values.
- **Write succeeds in less than *w* replicas:**
  If a write succeeds on some replicas, but failed on others (for example, because the disks on some nodes are full), and overall succeeded on fewer than ***w*** replicas, **it is not rolled back on the replicas where it succeeded.** This means that is a write was reported as failed, subsequent reads may or may not return the value from that write.
- **A replica fails and later restored from s replica with stale data:**
  If a node carrying a new value fails, and its data is restored from a replica carrying an old value, the number of replicas storing the new value may fall below ***w*,** breaking the *quorum* condition.

Eventual Consistency

From the discussion so far we see that, although *quorums* appear to guarantee that a read returns the latest written value, in practice it is not so simple. *Dynamo-style* databases are generally optimized for ***eventual consistency.***

The parameters ***w*** and ***r*** allow us to adjust the probability of stale values being read, but it's wise to not take them as absolute guarantees.

So in summary, databases with appropriately configured quorums can tolerate the failure of individual nodes without the need for failover. They can also tolerate individual nodes going slow, because requests won't have to wait for all ***n*** nodes to respond – they can return when ***w*** or ***r*** nodes have responded. These characteristics make databases with *leaderless replication* very appealing for use cases that require *high availability* and *low latency*, and that can tolerate occasional stale reads.

Leaderless Replication is ideal for systems that:
1. **can tolerate occasional stale reads (*Eventual Consistency*),**
2. require ***high availability*** and ***low latency.***

Categories: Computer Science, Scalable Systems