Designing Data-Intensive Applications

Chapter 5: Replication

Part II

- Part I discussed building blocks when data is stored, even on a single machine
- Part II discusses storage across multiple machines
- Reasons for horizontal scaling:
  - Scalability
  - Fault tolerance/high availability
  - Reduced Latency, e.g. regional data centers
- There are *shared-memory* and *shared-disk* architectures, but this discussion is about *shared-nothing* architectures because those others have severe limitations, especially for geographic scaling

Introduction

- Replication is keeping an entire copy of the data on multiple machines. The next chapter will extend the discussion where the copies are partitioned to multiple servers.
- The challenge with replication is how to deal with writes (changes) because every node will have to process every write

Single-Leader Replication

Leaders and Followers

- One replica/node is designated the leader; all others are followers
- All writes must go through the leader; reads can come from any replica
- On writes, leader sends change log information to all followers
  - Synchronous - clear success/failure; clients must wait for all replicas
  - Asynchronous - lowest latency; no guarantee of durability after successful write
  - Single synchronous follower improves durability
- Adding followers usually involves a snapshot (like a backup) followed by applying changes since the snapshot
- Follower failure uses catch-up recovery

Leader Failure - Failover

- If the leader fails, promote one of the followers to be leader. Sounds simple, but…
- How do you know the leader has failed? How long for a timeout?

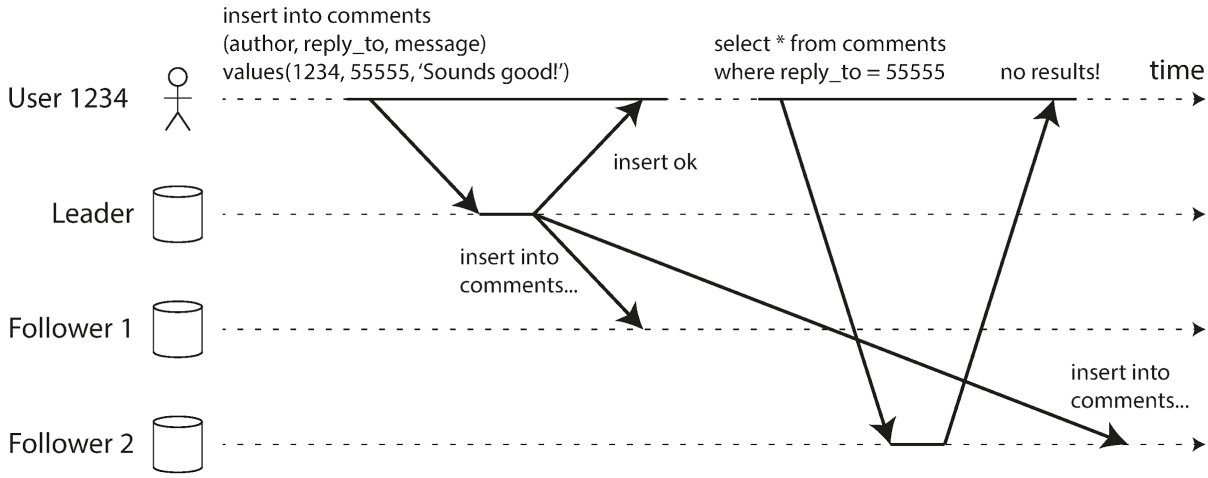
- How do you choose the new leader? Which node is most up to date?
- How do you reconfigure the system for the new leader? How can you be sure everyone has the new configuration? What about when the leader comes back? What if two nodes both think they're the leader?
- What happens to writes that may not yet have been fully replicated from the old leader?
- Chapters 8 and 9 go into more detail about issues with distributed systems, consistency and consensus.
- Because no easy answers, some operations teams fail over manually

Implementation of Replication Logs

- Statement based replication - repeat the original command. Although compact, nondeterministic commands cause problems
    - E.g. RAND(), auto-incrementing column, other side effects
- Write-ahead log (WAL) shipping - send detailed physical write change log. Although simple, can be voluminous, and physical details makes hard to support version upgrades
- Logical (row based) log shipping - send row level key and data
- Trigger-based replication - manually coded. Flexible, higher overhead, risk bugs

Problems with Replication Lag

- Can realistically only scale lots of followers asynchronously
- Ideally, *eventual consistency* is only brief, but *replication lag* can be seconds or minutes
- Reading your own writes



A write followed by a read from a stale replica will look to the user like their data was lost

    - One option if few things modifiable, read them all from the leader
    - Could track most recent update and force reads from the leader for a short period, e.g. one minute
    - If client knows timestamp of last write, replicas can delegate reads if they aren't up to date at least to that timestamp
    - Cross-device read-after-write consistency is even trickier
- Monotonic reads - user makes two reads, the second is from a more lagged replica. This will look to the user like information went backward in time.

    - Consistently reading from same replica will solve this (user affinity hashing)
- Consistent prefix reads - If two writes appear to third person in the reverse order, may seem to violate causality. More common when also partitioning.
    - Could try to force causally related write to same partition, but that's really hard
    - At end of chapter will discuss sequential vs. concurrent writes

Solutions for Replication Lag

- Transactions are a single-node way of encapsulating behavior guarantees
- Ideas for distributed systems will be discussed in chapter 9 and Part III

Multi-Leader Replication

- With multiple leaders, each leader also acts as a follower to the other leaders

Use Cases

- Multi-data center operation - leader per data center improves performance and tolerates outages and slow traffic between data centers better
- Clients with offline operation - offline clients act as leaders, then sync when online again
- Collaborative editing - Google Docs is essentially multi-leader replication

Handling Write Conflicts

- Downside of multi-leader is that write conflicts can happen, requiring conflict resolution
- Conflicts are generally detected asynchronously, so too late to prompt user
- Conflict avoidance - avoidance is highly recommended
    - Assigning each user a home datacenter works until a failure or they move
- Converging toward a consistent state - order of writes can be different at different nodes
    - Last write wins - often based on somewhat arbitrary unique ID
    - Pre-define arbitrary replica precedence rules, then node A always overrules B
    - Somehow merge the conflicting values being written, e.g. concatenation
    - Record in a conflict log and resolve later, possibly with user input
- Custom conflict resolution logic - write your own application-specific logic
    - Resolve on write - once conflict detected, run automated code
    - Resolve on read - save all conflict info; when read either run automated code or show user info and prompt for resolution

Automatic Conflict Resolution

- Conflict-free replicated data types (CRDTs) - data structures for sets, ordered lists, counters, etc. which have sensible conflict resolution rules built in
- Mergeable persistent data structures - track history and use three-way merge functions (not two-way merge)
- Operational transformation - the conflict resolution algorithm for ordered lists, which Google Docs uses (for an ordered sequence of characters)

Multi-Leader Replication Topologies

- Common topologies include:

    - All-to-all - simplest
    - Circular - simple version is unidirectional
    - Star - one central hub; can be generalized to a tree
- Circular and star topologies require multiple hops
    - Need to track nodes seen for each write
    - Single node failure could cause replication failure
- All-to-all could have the consistent prefix reads kind of problem
    - Version vectors (later in this chapter) could be used, but not popular at time of printing of this book

Leaderless Replication

- Amazon Dynamo led the way for leaderless replication
- Either the client directly sends writes to multiple replicas, or a coordinator handles that

Writing to the Database When a Node Is Down

- The benefit of leaderless is that as long as sufficient nodes complete a write, you can successfully continue. When nodes come back up, some of the data is stale.
- Clients detect stale reads by reading from multiple nodes too
- Catching up on stale data from missed writes:
    - Read repair - when a client detects a stale read, it writes the newer value back
    - Anti-entropy process - background process that searches for differences between replicas and corrects them. No guarantee of order or time until repair.

Quorums for Reading and Writing

- The minimum number of nodes needed above is called a *quorum*
- If there are *n* nodes, *r* are required for reads, *w* are required for writes, then $w + r > n$
- Typically, *n* is an odd number. Often *w* and *r* are half *n*, rounded up, or $(n + 1) / 2$
- Writes can tolerate $n - w$ nodes being down; reads can tolerate $n - r$ nodes down. If more than that are down, the operation returns an error

Limitations of Quorum Consistency

- Choosing $w + r <= n$ risks stale reads. Even with proper quorums, edge cases include:
    - Sloppy quorums (next section)
    - Concurrent writes
    - If a write operation fails to get a quorum, a subset of nodes won't roll back, so will incorrectly have the newer value
    - If a failed node A is restored from node B, anything stale on B is now stale on A, and both A and B can contribute toward a read quorum

Monitoring Staleness

- With leader based replication it is usually fairly easy to monitor staleness because you can compare the "version" of each follower relative to each leader to see how far behind the follower is on applying writes.

- Beware that with leaderless replication it's harder to measure because writes aren't applied in the same order. Further, if anti-entropy background processes aren't used, it is reasonable behavior for an infrequently read item to be stale even if the write happened forever ago.

Sloppy Quorums and Hinted Handoff

- In a large cluster, a network interruption could easily cause there to be not enough nodes for a quorum
- Is it better to simply return errors, or to try a workaround?
- *Sloppy quorums* is a workaround where we get *w* nodes, but some of them are not the proper nodes where the write is supposed to be stored. When connectivity is restored, temporary nodes use *hinted handoff* to write the data back to nodes where it belongs.
- This provides fault tolerance, but violates quorum properties, allowing stale reads

Multi-Datacenter Operation

- Replication that includes multiple datacenters can be implemented with some of the *n* nodes in each datacenter, but otherwise like vanilla replication
- Given that communication between datacenters is expected to be slow, some products are configured to send cross-datacenter writes asynchronously. Riak limits initial replication to be within a datacenter, and uses a multi-leader strategy between datacenters.

Concurrent Writes

- Concurrent writes can happen in multi-leader replication or leaderless. In leaderless replication they can also happen during read repair and hinted handoff.
- A simple approach is *last write wins* (LWW). Biggest value of some "timestamp" with each write wins. Even though clients saw multiple successful writes, all but one will be lost.
- To better address concurrent writes, we need to store more information, and we need to define concurrency.
    - If A was aware of write B, we say B happened before A.
    - If B was aware of A, then A happened before B.
    - In all other situations where neither A nor B were aware of the other, regardless of time according to wall clocks, we define A and B as concurrent writes.
- The author shows a single-node algorithm for tracking writes. Some details are:
    - Server maintains a version with each key
    - Clients must do a read, which includes version(s) and value(s), before they write
    - Clients must merge multiple values read before doing a write
    - Server increments max version each write
    - Server receiving a write can discard data from that version or older, but keeps newer data
- Note that clients merging multiple values is nontrivial, and an example is given where a simple union of values in a shopping cart caused removed items to reappear (because tombstones were not implemented)

- *Version vectors* is the multi-replica version of the above algorithm
    - This requires the servers to maintain a version per key *and* per replica
    - Servers now keep track of multiple versions & values from each replica
    - Clients merging multiple *sibling* values must consider versions on each replica

Summary

- Three main approaches to replication are single-leader, multi-leader, and leaderless
- Single-leader is simplest, but can't scale as much as the other two
- Replication activity can be synchronous or asynchronous, with very different performance properties
- Even single-leader replication can suffer from replication lag, possibly failing to provide: read-after-write consistency, monotonic reads, or consistent prefix reads
- There are algorithms to address concurrent writes without loss of data