Designing Data-Intensive Applications

Chapter 9: Consistency and Consensus

Introduction

- We've looked at the building blocks, and last chapter discussed all of the things that can go wrong in distributed systems
- This chapter talks about some of the ways to address consensus, one of the hardest problems, and we'll see that it's very tricky, and approaches can be difficult or expensive

Consistency Guarantees

- Most replicated databases offer eventual consistency, but that's so weak it makes it hard to reason about and build applications
- We're going to look at stronger consistency guarantees that distributed systems can provide, covering linearizability, total ordering, and consensus

Linearizability

- Instead of getting different answers from different replicas, *linearizability* abstracts away implementation details, and provides the illusion of a single replica
- A sports score that appears to go backward in time is not linearizable
- Linearizability can be explained by each read/write operation having a specific moment in time it represents between the time of the request and the response, and that the results are consistent with some set of specific moments in time that keeps moving forward.
- *Note: linearizability and serializability sound similar, but are independent concepts. A system can provide neither, one, or both of these.*

Relying on Linearizability

- Use cases include:
    - Locking and leader election
    - Unique constraints
    - Cross-channel timing dependencies

Implementing Linearizable Systems

- Replicated systems can be linearizable as follows:
    - Single-leader replication -- potentially; not if it uses snapshot isolation, not if asynchronous writes, must ensure leader failover is satisfactory
    - Consensus algorithms -- yes
    - Multi-leader replication -- not lineariable, concurrent writes
    - Leaderless replication -- probably not linearizable; depends on conflict resolution and requires strict quorums
        - With Dynamo-style quorums, you must do synchronous read repair, and readers must read from a quorum of nodes before each write
- Cost -- imagining the use case for replication between two datacenters exposes a problem. If the connection between datacenters goes down in a linearizable system, it will not be able to service requests to datacenters that don't have the leader.

- CAP theorem -- this theorem says that in any system that requires linearizability, if some nodes become disconnected, they must wait to process requests. Systems that do not require linearizability can be written so that disconnected nodes don't have to wait.
    - Beware of misunderstanding and misleading claims around CAP
- Linearizability and network delays -- Linearizability costs a lot in performance. It has been proved that the response time of requests is at least proportional to the uncertainty of delays in the network. Even RAM on a single system isn't linearizable when you have multiple cores.

Ordering Guarantees

- There is a strong connection between ordering, linearizability, and consensus
- A *causally consistent* system obeys the ordering required by causality
    - Examples of causality already discussed include: consistent prefix reads, snapshot isolation, and write skew
- The causal order is only a partial order, whereas a linearizable system has a total order. The former can have concurrent operations, the latter cannot.
- Linearizability ensures a causal order, which is why linearizability is easy to reason about
- There is research into capturing causal dependencies and providing causal consistency without going all the way to linearizability. One possibility is to extend version vectors across all keys.

Sequence Number Ordering

- Simpler than tracking causal relationships is simply tracking sequence numbers to order events
- A logical sequence number can provide a total order (which is causally consistent). For example, single leader replication with a numbered replication log does this.
- Lamport timestamps -- method for handling sequence numbers for a distributed environment, which is causally consistent
    - Uses counter and node ID pairs
    - Every node and every client tracks the max counter it has seen. Whenever it sees a value larger than its current counter, it bumps up its counter to match
    - Provides a total order
- Unfortunately, a total order is not enough to resolve unique constraints.
    - This is because the order is only known after the fact. To deny an insert due to violation of a unique constraint, you need to know if you're not the first requestor at the time of the request

Total Order Broadcast

- Total order broadcast is a protocol for exchanging messages between nodes with:
    - Reliable delivery - if delivered to one node, must be delivered to all
    - Totally ordered delivery - every nodes gets messages in the same order
    - If any nodes are disconnected or down, delivery can be delayed, but it must eventually happen, and in the correct order
- Total order broadcast can be used to implement replication or serializable transactions

- Implementing linearizable storage using total order broadcast
    - Linearizable compare-and-set writes
        - Append to the log the key you want written
        - Read the log until you see your write
        - Check for any messages to your key prior to your message
    - For linearizable reads, three options mentioned to avoid stale reads
        - Append to the log, wait to see it appear, then do read
        - If you can fetch the latest log position, wait to see it appear, then do read
        - If you have a synchronously updated replica, read from it
- Implementing total order broadcast using linearizable storage
    - Assume you have a linearizable register with atomic increment-and-get or atomic compare-and-set
    - Increment this register and get the new value
    - Use this number as a sequence number for the message you send
        - Keep retrying any failed messages forever
- It turns out that a total order broadcast or linearizable register with atomic compare-and-set are equivalent to consensus

Distributed Transactions and Consensus

- You want all nodes to agree on certain things, including leader election and atomic commit (across multiple nodes)

Atomic Commit and Two-Phase Commit (2PC)

- On a single node, data is written first, then the commit record
- In a distributed system, you can't just ask each node to commit independently
- Two-phase commit uses a separate coordinator
    - Coordinator sends a prepare to all nodes and requires a confirmation from every node
    - Then coordinator writes the commit record
    - The commit (or abort) decision is then sent to each node, and it must retry forever
- Node failures are fairly easy to handle with 2PC, but if the coordinator fails after the prepare but before the commit, nodes are left in limbo
- There is a design for three-phase commit which is non-blocking, but it requires bounded delay and response times (i.e. perfect failure detection is possible), so it's not used

Distributed Transactions in Practice

- Distributed transactions have pros and cons. The atomic commit is desirable functionality, but lots of systems have performance issues and operational problems
- The author differentiates two use cases:
    - Database-internal - all on same software
    - Heterogeneous distributed transactions - compatible with other systems, and harder to support

- Exactly-once message processing - in a message queue, atomically commit the message acknowledgement and the database write in one distributed transaction
- X/Open XA is a standard for heterogeneous 2PC since 1991
    - It is an API, and often the coordinator is run in the same process as the application starting the transaction
    - Holding locks while in doubt - if the coordinator crashes after the prepare, all of the participants will hold locks forever, even across restarts, until the coordinator successfully sends an abort or commit
    - If a glitch happens causing *orphaned transactions* on the coordinator, manual intervention is required to release participant locks (or *heuristic decisions*, although they violate atomicity rules)
    - Here are some operational issues:
        - The coordinator usually isn't fault-tolerant
        - If the coordinator runs on the application tier, that server is no longer stateless
        - As a lowest common denominator across many systems, it can't do advanced things like detect deadlocks or participate in SSI
        - All 2PC tends to amplify failures

Fault-Tolerant Consensus

- Consensus is generalized as a problem where node(s) may propose values, and consensus decides on one of those values
    - Uniform agreement - same decision
    - Integrity - only decide once
    - Validity - choice must be one of the proposals
    - Termination - every node that doesn't crash eventually decides
        - 2PC does *not* meet this requirement
- Best known consensus algorithms are:
    - Viewstamped Replication (VSR)
    - Paxos (and Multi-Paxos)
    - Raft
    - Zab
- Most of these implement total order broadcast, which is equivalent to repeated rounds of consensus
- In single leader replication, the leader provides a total order broadcast, which is consensus. But if a leader fails, you need consensus to elect a new leader. So what do you do?
- Epoch numbering
    - These consensus algorithms track a monotonically increasing epoch number, and only ensure that within each epoch number, the leader is unique
    - If two would-be leaders disagree, the one with the higher epoch number wins
    - Before a leader can make a decision, it needs to collect votes from a quorum of nodes
- Limitations

    - The voting process for proposals is a form of synchronous replication - and potentially slow
    - Since consensus requires a majority, a network partition will make minority nodes unable to proceed
    - Most consensus algorithms don't allow adding or removing nodes
    - If you have highly variable network delays, timeouts can cause frequent leader elections, which consumes many resources
    - Edge cases exist, such as where Raft got stuck bouncing the leader back and forth between two nodes forever

Membership and Coordination Services

- ZooKeeper or etcd are often described as "distributed key-value stores" but you wouldn't use them as application databases. Typically they hold system configuration settings.
- They're designed to hold small amounts of data, all in memory. This small set is kept in sync using total ordered broadcast.
- Other features offered by ZooKeeper:
    - Linearizable atomic operations. You can use this to implement a distributed lock (or more typically a lease, which has an expiration)
    - Total ordering of operations
    - Failure detection, using heartbeat connections
    - Change notifications, e.g. push notification of setting changes
- Typically you wouldn't run ZooKeeper on thousands of nodes. Rather you'd have 3 or 5 that handle key coordination, and all other nodes would look to those handful of nodes.
- Allocating work to nodes - ZooKeeper is good for choosing a single node, like a leader, and it is also good for allocating resources
    - Job schedulers
    - Assigning and rebalancing partitions to nodes
    - Can be done using atomic operations, ephemeral nodes, and notifications. But it's still complicated enough that higher level libraries like Apache Curator can be helpful
- Service discovery
    - ZooKeeper, etcd, and Consul are often used for service discovery
    - Service recovery may not really require consensus (DNS works pretty well), but if you're electing a leader anyway, perhaps help with discovery. An additional service is read-only caching replicas that can serve results of all decisions, but don't participate in voting.
- Membership services
    - A membership service tracks which nodes are alive and active