adamconrad.dev

Replication

20-25 minutes

> This article is part of the Systems Design Series following the book *Designing Data-Intensive Applications* by Martin Kleppmann. If you missed the previous article, check that out first.

This chapter is the real introduction to distributed systems. We are shifting into the second part of the book here. You will notice that I intentionally skipped chapter 4 based on a suggested curriculum I outlined in the first post of this series.

Horizontal versus vertical scaling

Before the chapter gets started, Kleppmann briefly discusses how we've collectively arrived at a *shared-nothing architecture* of horizontal scaling. Distributed systems design focuses on scalability so it is important to know the 3 main types of scaling you'll see in real-world systems:

1. **Vertical scaling.** Also known as *shared-memory architectures* or *scaling up*, this involves adding more RAM, CPU, and resources to a single machine. The key observation is that **doubling resources will likely more than double your costs**. As such, vertical scaling is recommended when heavy computation is required at the expense of fault tolerance and reliability. There is a variant of this called *shared-disk architecture* which pertains mostly to data warehouse applications that provide additional fault tolerance with multiple machines accessing the same disks on a centralized

network. This still suffers from the same issues when you colocate all data in one data center but has limited use.

2. **Horizontal scaling.** Also known as *shared-nothing architectures* or *scaling out*, this refers to adding more machines with commodity resources. This is the primary strategy used when companies scale out their services and hence is the primary focus of this section of the book.

Why do we replicate?

1. **Reduce latency.** Keeping data close to your users reduces the time it takes to physically travel over the wire to your customers.
2. **Increase availability.** The server goes down? You've got a backup on a different machine and in a separate data center.
3. **Increase throughput.** Drive-thru windows at your bank are convenient but if the line is long you know there are lots of bank tellers available inside to help you out and keep customers and productive.

This chapter discusses three primary reasons *why* we need to replicate. We'll offer strategies to resolve why **data changes all of the time and how we deal with it**.

If data never changed then replication would be easy. Since data grows and evolves over time, we employ various strategies based on our system's specific needs.

Leader replication

The first two methods involve **leader replication**. Each replica, or copy, has a role: `leader` or `follower`. Some schemes invoke a single leader while others leverage multiple leaders. The responsibilities of each are straightforward:

1. **Leaders ingest write data.** Only the leader can modify the database.

2. **Followers copy changes from the leader.** Followers must wait for a log file from the leader with changes they will make in the same order the leader made them.
3. **Anyone can offer read data.** All replicas can return data to a customer that asks for it. This is called *read-scaling architecture*.

   That last step is the interesting one. What if you ask for data from a replica that has not copied all of the latest changes yet? This is where replication strategies offer tradeoffs in their approach.

   From here, you can focus on two ways of getting changes to replicas:

1. **Synchronous replication.** As the name implies, you send changes to followers immediately. You then only inform customers when all changes are completed successfully. Practically, this doesn't work because if a follower replica hangs then the change fails which hurts durability. To be fair, **synchronous replication is the best strategy to eliminate the risk of data loss**. In reality, when systems use synchronous replication, they will follow a hybrid *semi-synchronous replication* strategy where one follower is synchronous while the others are asynchronous. Read-scaling architectures don't work with this scheme because followers have to wait for leaders to synchronize data which keeps them too busy to accept additional reads.
2. **Asynchronous replication.** In this scheme, changes are sent to followers immediately. The difference is you can inform customers when *some* changes complete successfully. You are not beholden to every follower sending a successful response. This is the most common scheme in production, high-scale application. This trades off speed for no guarantees in durability and eventual consistency.

   What if you want to add a new follower to the gang?

1. **Periodically take snapshots of the data.** This is good anyway for

backups. Send the latest backup to your new follower.

2. **Poll the leader again for changes since the snapshot.** Now you can assume the new follower is asking for changes as if this were an asynchronous replication strategy. Each database engine has its own scheme for issuing catch-ups such as log sequence numbers or binlog coordinates.

   What if things fail?

- **If the follower fails** then you execute the same catchup strategy for a new follower. Just read from the logs the last changes since the failure and then play catchup.
- **If the leader fails** then you have to elect a new leader. You can either do this *manually* or *automatically*. Manual promotion benefits from careful failover to a verified follower. The downside is that you have to invest developer time into failovers and can be cumbersome if they occur with some frequency. Automatic promotion relies on selecting a new leader, usually with the closest representation to the leader's data.

  Reaching consensus with the automatic strategy is fraught with problems:

- **There might be a gap in the data, even with the best follower chosen.** In this case, you violate strict durability and you incur data loss which is never good.
- **Multiple followers might believe they are the leader.** This can also lead to data loss if both start accepting writes to propagate to the other followers.
- **The heartbeat interval may be too long and miss the latest changes.** Even if you have one follower with a perfect copy of the leader's data, the leader may have been down long enough to miss some new writes. If the heartbeat to check for its health is in a long enough interval then new changes could have been requested

while the leader was down which will be lost for good.

- **The old leader may not relinquish their throne**. Assuming everything else was correct, you still might see a variant on the second issue where the old leader starts back up and still thinks it is the leader. Now you are in a similar scenario where two leaders are consuming writes and you incur data loss.

  Logs help with restoring the data in addition to helping new followers join in on the fun. Each database engine utilizes one of four replication log schemes:

1. **Statement-based.** These copy raw SQL commands. They are simple but require deterministic commands, which can be too strict for most databases.
2. **Write-ahead.** These are logs written before changes to the main disk are stored. This is a common append-only log that is used with things like SSTables and B-trees but requires all engines to follow the same format. If the format changes with breaking changes to an engine, then it will require downtime to upgrade. This is what plagues things like major version upgrades in tools like Postgres.
3. **Row-based.** This focuses on the values of rows instead of the commands like the previous two. This decouples the data from the storage engine used, which solves the problems from the previous strategy. New versions of Postgres use this, also known as *logical log replication*. The book doesn't seem to see a downside to this approach but, this Stack Overflow thread goes into this strategy in greater detail which I appreciated. In short, things take more time because one database is updated at a time methodically.
4. **Trigger-based.** This strategy leverages SQL triggers and stored procedures. While this benefits from functionality native to the language instead of the database engine, these triggers can be complex. Further, this log strategy does not work for NoSQL solutions or any database that doesn't leverage SQL, such as

graph databases.

Eventual consistency

As mentioned earlier, a read-scaling architecture can provide additional performance for applications that are heavy on reads. The downside of this strategy is that you may read outdated information from a follower node. When a leader node sends data asynchronously to a follower node, it cannot wait to know if the follower replication succeeded.

This is what is known as *eventual consistency*. At some point, *eventually*, the data will be synced across all replicas. There is no guarantee when that eventual time will be. But, it is meant to be a reasonably short amount of time so that there is no real compromise in reliability. Nonetheless, there are occasional complications with eventual consistency:

- **Reading right after writing.** Say you update your email. If you make those changes and refresh the page then you'll want to see your new email address saved. *Read-after-write consistency* is a strategy around this but is complicated in distributed systems.
- **Reading into the past.** Say you write a few checks to your bank account. Your bank account starts at $5, then you add two checks of $5 each. In theory, if the eventual consistency is delayed enough between replicas, the leader could show $15, one replica could show $10, and yet another replica could show the original $5. *Monotonic reads* ensure you only read from one replica so this problem doesn't occur. Of course, if the replica fails, you're back to square one.
- **Answers to questions you haven't asked.** In the previous scenario, you could see a later state before you see a previous state. Not only are you going into the past but you are messing up the causality of the data - you wouldn't expect an addition to lead to

subtraction of the balance. *Consistent prefix reads* to ensure that the writes happen in a certain order. If you have a sharded database, you can't guarantee this fix unless all of the data is affixed to the same shard. This becomes more complicated but does solve the issue.

All of these problems are difficult to solve with a single-leader replication strategy. They become less difficult when you invoke a multi-leader replication strategy.

Multi-leader replication

As the name implies, multi-leader replication allows you to assign multiple leaders at once. This solves nearly all of the problems you see above:

- **No need for manual promotion.** With multiple leaders, you don't have to worry about being without a leader at all. Restarting leaders assumes there is at least one leader already able to receive writes.
- **No gap in writes.** Unless you get incredibly unlucky, a leader will exist to ensure there is no downtime of a leader to accept writes. This can happen with a single leader. With multiple leaders, you guarantee there is always a place to accept writes.
- **No leader conflict resolution.** If you know who the leaders are, you can always create a backup scheme so you either fall back to an alternative leader *or* you have a simple election to keep things smooth.
- **Heartbeat intervals no longer matter (to a point).** Waiting a long time to see if the leader is still online is not a problem because you have multiple leaders to fall back on. As long as their heartbeats aren't synced and are checked in a staggered fashion, you can ensure that you're constantly aware of at least one online leader.
- **Multiple leaders don't necessarily require follower promotion**

at all. One problem with a single-leader scheme is the leader won't give up its role as leader when it comes back online. If you have other leaders available, there is no need to promote them. You always have leaders available to do all the work you need. If a particular leader goes offline, you just wait for it to return and push the additional load to your remaining leaders.

In addition to solving problems from single-leader replication, there are a few use cases where multi-leader replication really shines:

- **Datacenter replication.** Have multiple data centers? Assign a leader for each data center to ensure all locations have write coverage. The challenge is that you are back to single-leader replication per data center if you don't elect multiple leaders per site. Further, writes become far more complicated since you have multiple data centers that have to resolve data changes. That requires costly RPC calls to copy logs from one data center to the other.
- **Offline applications.** What if your mobile phone were a data center? When you're offline, you store changes to your local database. When you get back online, you sync your changes with a leader at a data center for persistence and backups. This is multi-leader replication taken to the extreme where every device acts like a leader.
- **Collaborative applications.** Tools like Google Docs allow you to edit concurrently with other users. Think of your browser as a leader where you publish changes locally and see them instantly in the browser. In the background, you take those changes and push them to a remote leader replica. This replica syncs your changes with the changes of another individual. They ensure they all match nicely.

Everything sounds great - so what's the catch? It should be obvious at this point, but the big issue with multi-leader replication is **write**

conflicts. With multiple leaders comes multiple nodes where you can write changes to your database. Who has the correct answer? Is there a leader of leaders?

The easiest way to mitigate write conflicts is to simply **avoid having them in the first place**. This may seem like a cop-out answer, but you can design your systems toward conflict avoidance. Having one leader per data center and routing users to specific data centers is a way to avoid conflicts. Specific updates in a particular range of data go to dedicated leaders. Of course, this reverts to being a microcosm of single-leader replication, so if a data center goes down, you are forced to reconcile concurrent writes on different leaders.

If you can't avoid conflicts, you can use some *tie-breaker heuristic* such as assigning UUIDs to data centers or records so that the highest numbered ID wins. Simple, but prone to data loss. In theory, this also allows you to merge conflicting values, but then you'll need some sort of user intervention to reconcile the concatenated data.

Of course, you could implement your own custom conflict resolution scheme. Amazon has done with this some surprising effects, such as having items reappear in users' shopping carts after they have purchased items. While a suboptimal user experience, it is still likely better than losing data.

How do leaders communicate with each other? The book identifies three specific *topologies* for communication, and I see there are only two:

- **Every leader talks to every other leader.** This ensures everyone gets the message from everyone else with a direct line of communication between every leader. The problem here is there is no causality to the writes because there is no order with which leaders are updated. As opposed to…

- **Leaders update each other in a chain.** An example of this would be a circular or star-shaped topology. In these schemes, there is an ordering to how leaders communicate with each other which solves the causality problem. The challenge is that if a replica fails during the update chain, it breaks the chain, and all downstream updates will stall out. If a leader is early enough in the chain, most other leaders could suffer from data loss and stale information.

As mentioned before, the best course of action is generally to avoid conflicts in the first place. Many database systems support some kind of conflict resolution scheme, but none are perfect.

Leaderless replication

The last major strategy with replication is leaderless replication, where **every node can accept reads and writes.** Amazon Dynamo is the most famous example of leaderless replication and has ushered in a resurgence of this strategy. Cassandra, Riak, and Voldemort have also adopted leaderless replication schemes. This scheme is ideal for applications that require **high availability and low latency** at the expense of a very loose definition of **eventual consistency and stale reads**.

When every database replica can write data, you open yourself up to lots of staleness. There are two primary strategies for dealing with this stale data:

1. **Use subsequent reads to detect stale data.** The next time a user reads data, read it from multiple replicas. If any of those are stale you can issue follow-up writes to ensure the stale nodes are brought back up to the latest versions of the data. This is commonly called *read repair*.
2. **Run a heartbeat to detect stale data.** Every update comes with a version number. Run a background process that pings every node to see what version of the data it has. If any of the nodes are less

than the latest version number, update the databases. Continue to ping all databases in regular intervals to ensure staleness is mitigated promptly.

Leaderless replication schemes often implement *quorums* on reads and writes to create consensus when there is a discrepancy. This is a tuneable threshold that allows developers to configure at exactly which point a vote passes to reconcile when replicas argue over which version of the data is the latest.

Quorums really only work if you have sufficient replica nodes to break a tie. This requires votes from read replicas, write replicas, and ensure those votes outnumber the number of replicas that debate the values. As long you have a majority vote on the state of reads and writes, you can proceed with quorum voting.

Further, **monitoring is difficult with leaderless replication**. This is because there is no universal order to how data is consumed. If a node goes down, or worse, several nodes go down, you risk destroying the quorum.

A concept known as **sloppy quorums** was designed to mitigate against this. It states it is better to **write the data even if a quorum is not achieved** when the required nodes for a quorum vote are down or offline. They still ask for the designated number of votes, but they may not come from the approved pool of original nodes that belong to the quorum. Without the approved set of nodes, you have fewer guarantees into the latest state of the data since these nodes are not as closely monitored.

This method *increases write availability* but require further mitigation like *hinted handoff* to restore the quorum node team. It also requires backup plans like read repair to ensure that restored nodes eventually receive the latest updates.

Another issue you can run into with leaderless replication is **concurrent writes**. A single change is propagated across multiple replicas. Each replica is subject to its own network latency and IO. There is no guarantee that any write will arrive at all replicas at the exact same time. How do you reconcile this case?

Last write wins is a simple strategy that dictates that the latest write is the one that will persist, and all other writes for the same data are considered stale duplicates that can be dropped. This sounds great in theory, but in practice, this is *bad for durability*.

Say your bank account reads $1. If you somehow issued 2 concurrent updates, mistakenly adding $2 instead of $4, you may run into trouble. Both updates will say they were completed successfully. But if the $2 change arrives after the $4 change, your bank account will update to $3 even if that isn't ultimately the correct answer. Cassandra mitigates this with a UUID attached to every write update to ensure seemingly concurrent operations are indeed differentiated.

There are numerous algorithms discussed in the book to determine concurrency. **Concurrency is not concerned with things happening at the same *time* but happening *without knowing about each other***. It's extremely difficult to say with certainty that two events occurred at the *exact* same time. But they could occur in a close enough interval to not know about each other based on the state of the application.

Riak knows how to merge concurrent values with its *sibling merge* functionality. Riak also uses *version vectors* to issue read repair so that clients can retrieve them on reads and return them back to the databases on writes. My takeaway here is if you are looking for **robust leaderless replication, look no further than Riak**.

That wraps up the chapter on replicating distributed data. Another way to distribute data is through partitioning. This is the topic of the next chapter so stay tuned!
