

Replication

Replication

September 7th 2019

To improve performance, you can duplicate data. The two big reasons for performance are:

- Keeping data geographically close, reducing latency
- Scale # of machines that can serve requests, increasing throughput.
- Provide fault tolerance via the backups.

Assume for now that the dataset can be stored on one machine.

Synchronous replication means that changes will be guaranteed to propagate, but real distributed systems can't generally make this guarantee. So the sync replication model guarantees neither, since nodes may be offline when changes are propagating.

Single-Leader Replication

One replica is designated as the leader, and all writes must go through it. Then, the write is propagated to following replicas (read-only replicas, from the clients perspective)

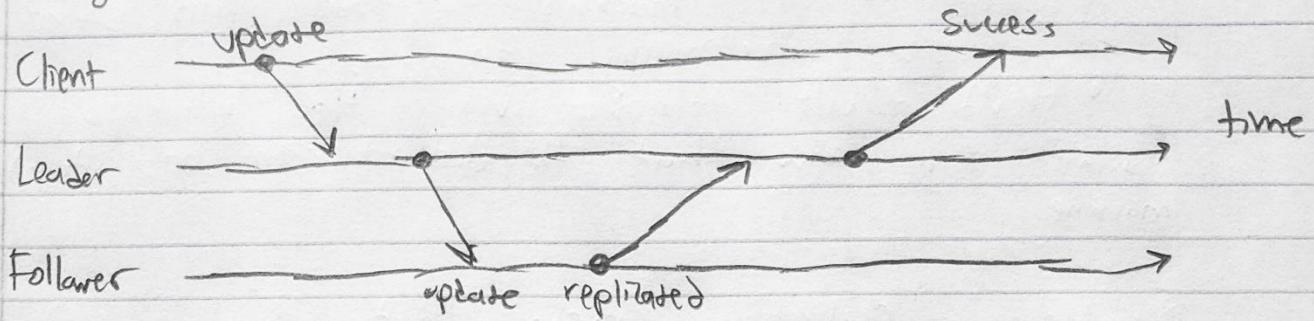
The leader sends it to the replicas usually via a log.

These services use this strategy:

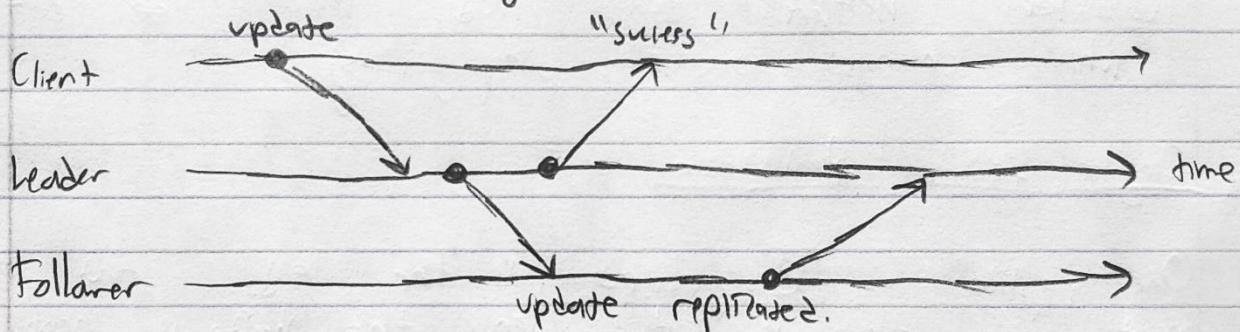
Relational DBs	Non-relational	Message Brokers
MySQL	MongoDB	Kafka
PostgreSQL	RethinkDB	RabbitMQ
Oracle Database	Espresso	

Synchronous vs. Asynchronous ReplicationSeptember 7th 2014

In synchronous systems, updates from the client need confirmation from the leader (and the followers) before being notified of success.



In asynchronous systems, there's no info about what the followers are doing from the client's perspective.



Updates are normally propagated pretty fast (≤ 1 second) but COULD take minutes if a follower is recovering from failure, or if the network is faulty, etc.

A node failure in a synchronous system causes the whole system to stall, so it's impractical to have everything be synchronous.

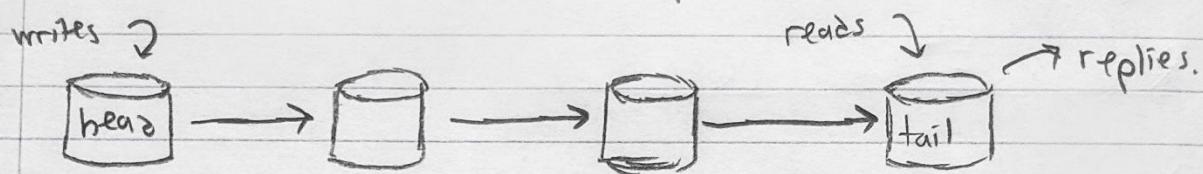
In semisynchronous systems, SOME of the followers (though usually one, in practice) are synchronous, and the rest aren't.

Asynchronous systems will:

- lose updates if the leader dies, as the replicas never receive them
- continue processing writes regardless of follower status.

Synchronous Replication: Chain Replication

The idea is to make the replicas into a "linked list":



The head failing means its successor becomes the new head.
The tail failing means its predecessor becomes the new tail

The internal nodes failing results in their removal, but the coordination required is complicated as not all messages from the failed node may have reached its successor.
(See the paper for more details).

→ The histories of updates needs to be kept, so the failed node's successor can determine which ones it needs.

Adding a node results in it becoming the new tail, and the previous tail propagates everything to it.

A downside is that while this increases fault-tolerance, load is not scaled as only one server handles each type of request

This is used in some systems, like Microsoft Azure Storage. It is ideal for low-demand high-availability systems.

RecoverySeptember 7th 2019

Node recovery has a lot of similarity to how nodes would scale up, since both imply that some node is not up-to-date (either by being new or offline for a while).

Depending on the DB, this can be any level of automated

Recovery: Followers (catch-up)

Since followers keep an update log, it can determine which updates it's missing by asking the leader.

If the node is brand new (by normal scaling up) or has been offline for way too long, it can do this:

- 1) Copy over a snapshot from the leader
- 2) Find all missing updates since the snapshot was taken.

The snapshots don't need to be fresh every time, backups can also work (and may already exist).

Recovery: Leaders (Failover)

The general process is as follows:

- 1) Detect leader failure: this can be done by any node
- 2) Elect new leader: usually want the new leader to be a more up-to-date one
- 3) Recconfigure routing: new writes need to go the new leader
- 4) Step down old leader:

August 12th 2019

Stuff that can go wrong:

- The new leader may be missing some updates, meaning writes from the old leader may be in limbo.

Most commonly, those writes will be discarded, which can cause correctness errors.

This problem is always present in asynchronous replication. A mitigating strategy would be to have some synchronous replicas (so semi-synchronous system) for backup.

- It may be possible for multiple leaders to exist at the same time (split brain). Without resolution, data can be lost or corrupted.
- Determining the timeout before re-electing leader is tricky:
 - too short \Rightarrow unnecessary failovers
 - too long \Rightarrow long recovery

There may also be scenarios where the source of stress is only temporary:

- load spikes, causing responses to go above timeout
- network failure, causing message delays.

These can trigger failovers that don't need to occur.

Replication Logs

When a leader makes a change, it will send the change to its replicas by a replication log.

August 12th 2019

Each replica maintains its own log of events it has processed.

Statement-Based Logs

Each event would be every write request (e.g. every SQL statement for a relational DB). Can be broken with:

- Non-deterministic functions like `NOW()`, which return different things depending on the replica.
- If statements depend on prior events (like `UPDATE ... WHERE ...`), then the events must be replayed sequentially.

This is very limiting when multiple transactions are executing

- Anything where the execution has downstream effects that aren't totally deterministic will break (like user-defined functions, stored procedures, etc.)

The edge cases are really tricky, and can be limiting overall, leading to many other log formats to be preferred.

Write-Ahead Logs

The WAL is an append-only sequence of byte blocks, recording all writes to the DB.

The leader sends the log to its followers, and you can rebuild the leader's state on them by

August 12th 2019

To allow crash recovery, the WAL is updated before modifying the main structure (like a B-Tree).

There are some disadvantages:

- Very low level, so it becomes dependant on the storage medium, due to details about disk blocks.
- Results in software updates that need downtime, as attempting to upgrade the followers followed by a failover for the leader is generally infeasible.

Used in PostgreSQL and Oracle DB.

Logical Logs (Row-Based Logs)

An attempt to decouple the logical information and the data format, row-based logs keep a record of row-level changes to the table.

- For insertions, log contains new values of all columns.
- For deletions, log uniquely identifies the row that's been deleted (usually the primary key, otherwise all columns need to be individually logged).
- For updates, log uniquely identifies the row with its new values (or at least the values that were changed).

Any transaction that modifies several rows will generate multiple records.

The decoupling of the storage medium means:

- Easy backwards compatibility
- Leader & followers can run different versions
- Heterogeneous storage.

Used by MySQL's binlog

September 15th 2019Trigger - Based Replication

A trigger is a user-given function that on data changes, will log it to a separate table.

That table can be read externally, and thus replicated.

Done at application level, so more overhead, (and more bugs and limitations) than the DB handling replication itself but very flexible.

Replication Lag/Latency

In asynchronous replication, the followers can become inconsistent with the master, if not completely up to date.

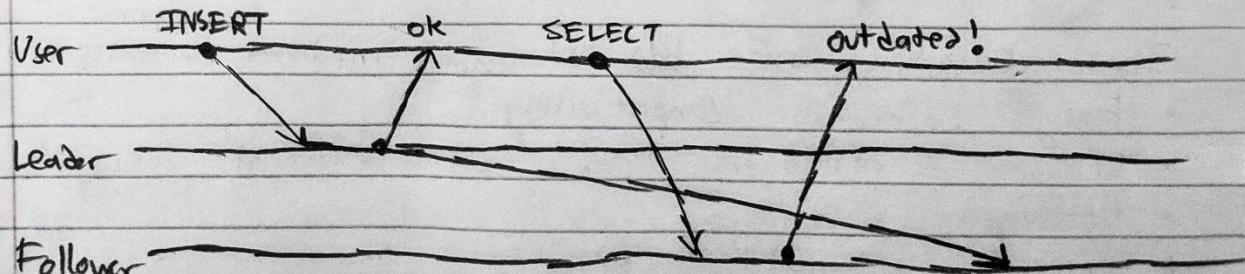
Eventual consistency states that the replicas will vaguely catch up to the master over time.

Is problematic if inconsistencies grow too large.

Read - Your - Writes

Successive read operations on a piece of data after a write on it will always return the result after writing.

If this isn't guaranteed, user can get data that looks like their update hasn't gone through:



September 15th 2019

Note that depending on the case, it may only be necessary to provide RYW consistency to that particular user. Other users' updates may not be visible until later.

Some techniques to ensure consistency are:

- When reading into that user may have modified, read it from the leader. This requires a way to know if a piece of data is modified without querying it.

For example, user account settings is only editable by that user, so all profile changes go to the leader.

- Other criteria for leader-reading can be used by tracking time after last update (like a minute) or redirect traffic from followers too far behind.
- Keep timestamp (either logical or system) of most recent write, and only query from replicas that are up-to or past that timestamp.

If the same user is using multiple devices, like a phone + desktop, RYW consistency must be cross-device.

This adds additional complexity:

- Timestamp-based ordering gets hard since the devices aren't directly synced. So metadata will need to be centralized.

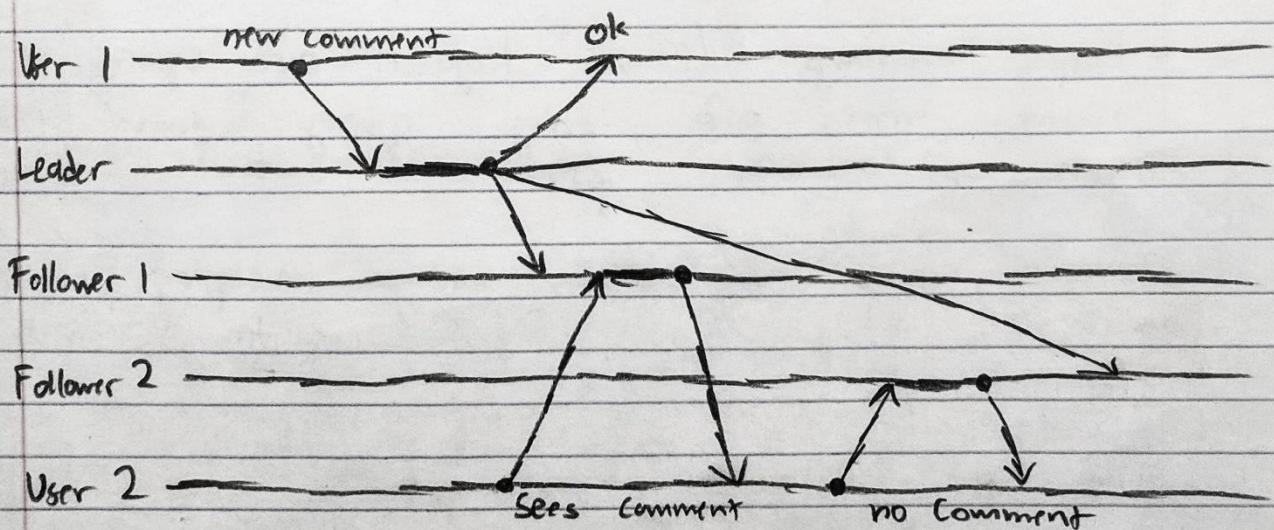
- If replicas are distributed in different datacentres, device connectors may go to different centres. So leader-only requests must go to the same datacenter, for all of the various devices.

Monotonic Reads

[Successive reads on a piece of data returns that state or successive states.]

If MR's are not guaranteed, it may look like you're going back in time.

Ex If User 1 comments, and User 2 sees that comment, User 2 may not see the comment after refreshing the page since the new request may go to an outdated replica.



Monotonic reads are less strong than full consistency, as it only guarantees that you won't go back in time. You may still read outdated data.

Transactions as a solution

If lag differs by several minutes to an hour and is okay for the user, weaker consistency is fine.

Otherwise, the system must be designed to avoid this issue.

Transactions serve as a way to help developers achieve stronger guarantees, albeit with some downsides.

Multi-Leader Replication

Same thing as single-leader replication, just that multiple nodes can process writes. All leaders must publish the change to all replicas.

Known as master-master or active/active replication.

Multi-Leader Use Cases

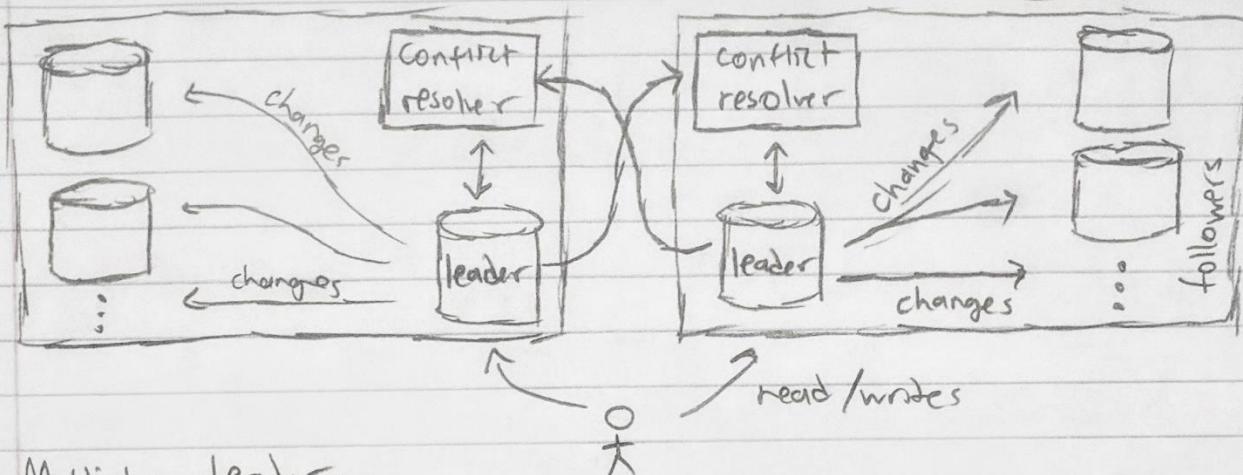
Rarely used within a single datacenter, but may have some scenarios where it is beneficial.

→ Multi Datacenter Operation

With only one leader, this is really hard as the leader can only be in one datacenter.

But you can instead have multiple leaders for each datacenter, where each datacenter follows a single leader approach.

Between datacenters, each leader replicates its changes to other leaders (like it would for its own followers).

September 15th 2019Multiple leaders

can help with the following when it comes to multiple datacenters compared to single leaders.

Performance Multiple leaders allow for localized writes, which may otherwise go to a datacentre far away if that datacentre contained the leader in a single-leader setup.

Outage Tolerance Datacentres can operate independently even if other datacentres fail.

Network Tolerance Inter-datacentre traffic over public networks are less reliable. Single leaders do this synchronously (why?) when linking to other datacenters.

Feature B is usually retrofitted in many DBs, so many subtle pitfalls and kinks may occur with the rest of the DB.

So multi-leader operation is usually considered only if absolutely needed.

Much of the time, multi-leader operation is provided with external tools, like BDR for PostgreSQL.

September 21st 2019

Offline clients

To allow apps to work offline for a while, every device can act as its own leader, with a local DB.

Between devices, copies of the same data (like between your phone & laptop), and will asynchronously update others just as with the multiple datacenter setup.

Collaborative Editing

Not exactly a database replication problem, but shares a lot of the same resolution conflicts. Like the offline clients solution above, its implemented as edits are written to the local DB immediately, before being asynchronously replicated to the server.

There are 2 general approaches:

- 1) Obtain a lock on the document (or section of the document) before a user can edit. This eliminates editing conflicts, and is analogous to a single leader with transactions.

But this is less performant than:

- 2) Make units of change really small (ex. keystrokes) and avoid locking. This will require write conflict resolution

Conflict Resolution

Unlike single leaders + transactions, multi-leaders will accept conflicting writes, only detectable later.

September 22nd 2019

If the system is synchronous, and writes must be fully replicated before accepting more, this is effectively the same as single-leader replication, and advantages of using multiple leaders is lost.

Conflict Avoidance

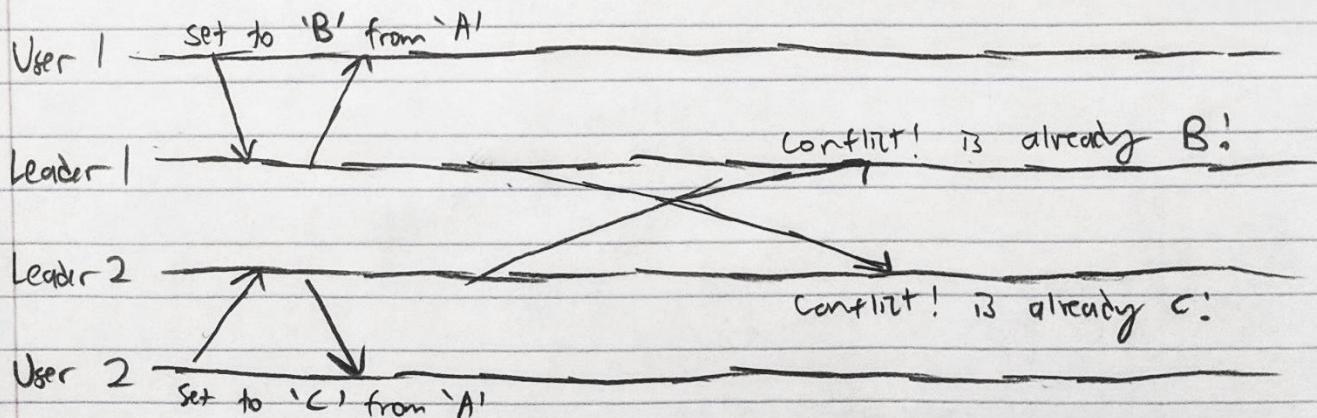
The easiest and simplest approach is to avoid them, which is commonly done.

A typical example is a user editing their data, and having them all sent to the same datacenter, effectively making it appear as single leader.

If the user is switched to another datacenter (by failure or load balancing) then conflicts may occur.

Conflict Resolution: Consistent State Convergence

For example let's say this happens:



... where 2 users edit the same info on the same page. When they first read the page, they saw the same thing, but now they both have conflicting changes, neither of which is more "correct."

September 22nd 2019

Solving this through convergence means that all replicas eventually arrive at the same final value. Some strategies include:

- 1) Give each write a unique ID (ex timestamp, UUID, key-value hash, long random number), and pick the highest one as the winner, discarding the other conflicting writes.

Popular, but can cause data loss. If a timestamp is used, it's known as last write wins (LWW).

Concept is similar to leader election, in the sense that the highest valued write is favoured over the others.

- 2) Merge the values (dependent on the application for specifics) like concatenating them, or something.
- 3) Record the conflict in an explicit data structure and send the conflicts back to the user for manual resolution.

Conflict Resolution: Resolution Log

A custom piece of code that gets executed on writes or reads.

- For writes, when a conflict is detected, the resolution logic is called in the background. Usually needs to resolve quickly to prevent more divergence, so user-level resolution is generally not an option.
- For reads, the recorded writes thus far are sent to the user for automatic or manual resolution. Used in CouchDB.

September 22nd 2014

Some patterns for conflict resolution logic have begun to emerge (as user given solutions are error-prone and less scalable).

- 1) Conflict-free Replicated Datatypes (CRDTs) are data structures that can be replicated and can resolve inconsistencies, much like the consistent state convergence strategy, through merges.

There are a few types of CRDTs, but the general goal is to not need expensive synchronization or consensus. Each one is pretty complex, so it will need to be investigated individually.

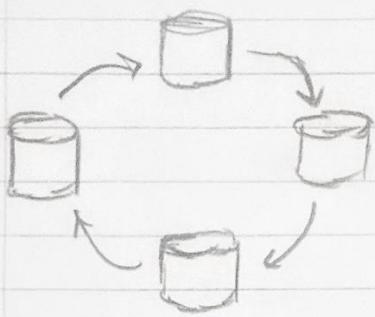
- 2) Mergeable Persistent Data Structures are similar to CRDTs, except they track the entire history explicitly, and uses a bit-style 3 way merge, instead of 2 like CRDTs.

- 3) Operational Transformation is the conflict resolution algorithm used in Etherpad and Google Docs. Its core purpose is to allow concurrent editing of ordered lists of items, like text.

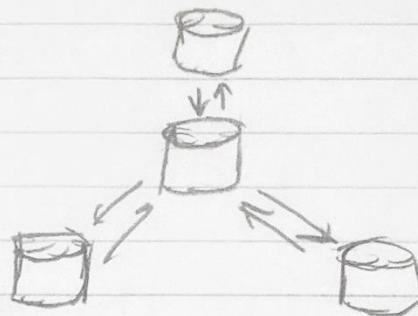
The use of these solutions will allow for more multi-leader systems in the future, but is still under research.

Multi-Leader Replication Topologies

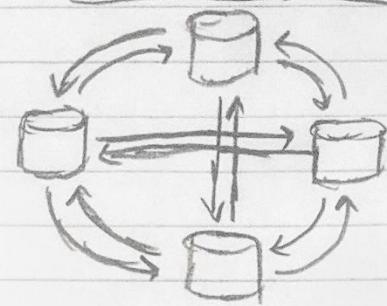
If you have more than two leaders, you can arrange them so that their communication is more or less restricted.

September 22nd 2019

Circular / Ring



Star.



All-to-All / Mesh

Different topologies have their own communication strategies, pros, and cons.

- In ring and star topologies, since information may need to pass through multiple nodes to reach everyone, there needs to be a way to prevent infinite replication loops.

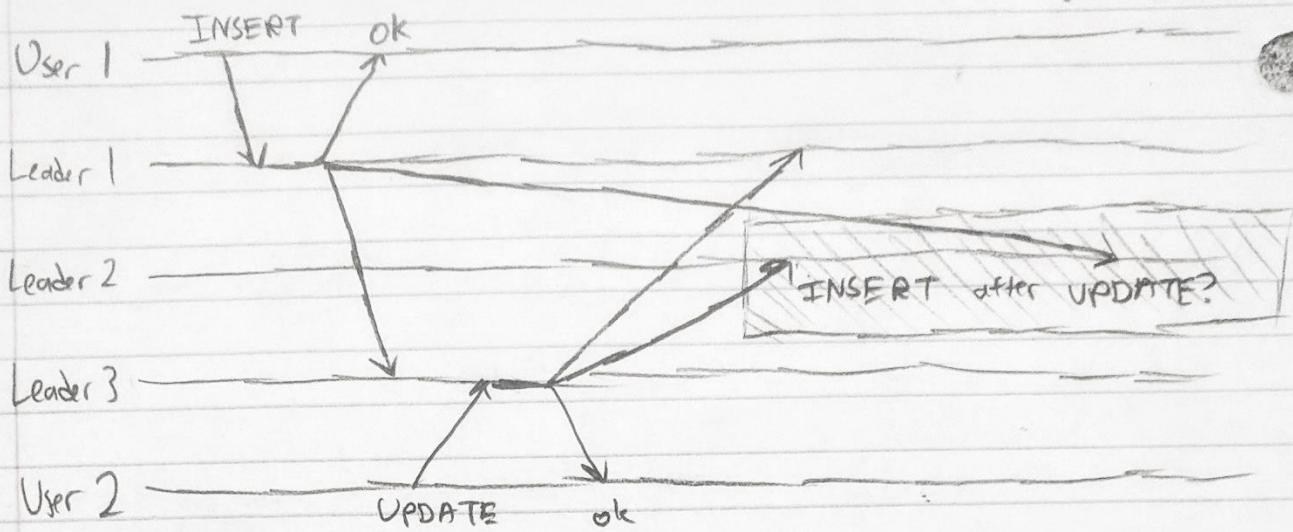
This is usually done by keeping a list of nodes it has seen so far (by some unique ID), and ignoring messages with its own ID on it as that means it's already been seen.

Node failures may also interrupt message flow, so fault tolerance is limited. Your only real option if it happens is reconfiguration, usually done manually.

- In mesh topologies, network links may have different latencies, and thus messages can arrive out of order.

Suppose an insert reaches a leader after the update for that insert does. This leads to a causality problem, and is generally fixed through version vectors, which are very similar to vector clocks.

(VV's are discussed later on).



This falls under "causality problems", and can be solved by using version vectors.

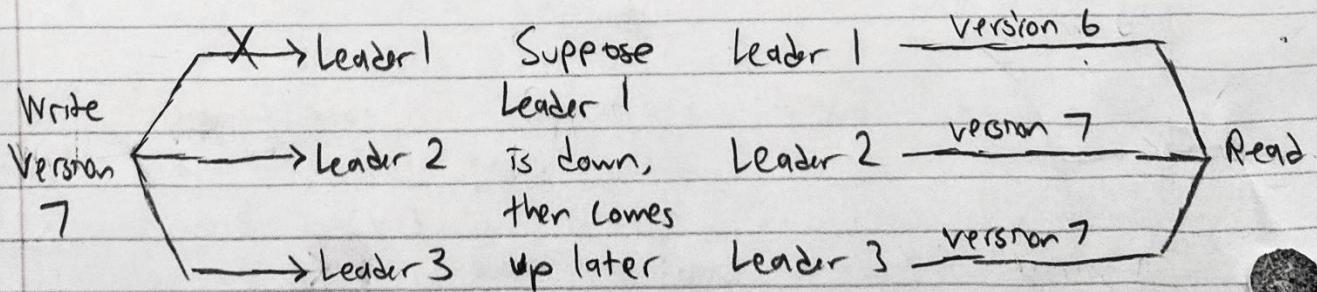
Leaderless Replication

Here, any replica can accept writes, making it leaderless.

There are two main strategies:

- Have client send directly to multiple replicas
- Have client send to coordinator, which forwards to multiple replicas. Different from leaders as write ordering is not enforced.

A cool property is that failover isn't necessary: by parallelizing reads and writes, you can see what the most common values are on a read, and assume it's the most up-to-date version.



The read would get an outdated value from Leader 1, but could deduce that it's likely out of date.

September 25th 2019

Node Recovery

For nodes that need to be updated after being offline, there needs to be a different approach from replication logs since no leader exists to be the source of truth.

Read Repair When the client sees a node that's outdated, it will send the new updates to the outdated.

Anti-Entropy Background processes look for discrepancies and repairs data between each other. No ordering is guaranteed between nodes, so delay may be high.

The most typical implementation is through anti-entropy gossip, along with Merkle trees to detect data discrepancy.

For read-heavy systems, read repair is good as the many reads trigger more updates.

For write-heavy systems, read repair may not occur often enough to keep decent consistency. Thus, anti-entropy is generally preferred.

Quorum Consistency

To be considered a successful read/write, there must be a minimum amount of nodes must confirm the operation.

The process would be to query a number of nodes and ask if they'll accept the operation before actually reading/writing.

To determine the minimum number of nodes that need to be contacted, consider that:

September 26th 2019

Suppose that to read, you need a read quorum, requiring N_r nodes. Similarly for writes, you need N_w nodes, and N is the # of nodes with this data.

Quorum Consistency solves two major consistency issues:

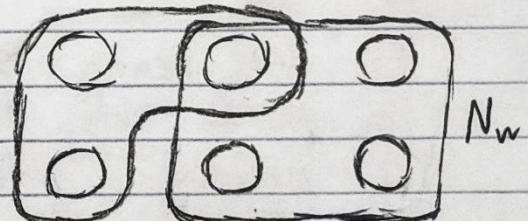
- 1) Read-Write Issues (both Read-Your-Writes, and Writers-Follow-Reads)
- 2) Monotonic Writes.

Read-Your-Writes and Writes-Follow-Reads

If $N_r + N_w > N$, then when reading after any write, one of the nodes are guaranteed to read the most recent write.

Same applies to writing after a read, as it will also have to apply on top of one of the nodes that were read from. Combined with read repair, this type of consistency can be achieved.

An example where $N_r = 3$, $N_w = 4$, $N = 6$. There will always be a node in the overlap if $N_r + N_w > N$.



Monotonic Writes

If $N_w > N/2$, then successive writes will always reach at least one other node that received the previous write.

Note that if less than N_r or N_w nodes are online, then no read/writes will process.

September 26th 2019Quorum Consistency Limitations

Normally, Quorums can support fault-tolerance. For example, if $N_r = 2$, $N_w = 2$, and $N = 3$, then one node could be offline and still support operations.

Fault-tolerance can be increased by lowering N_r and N_w , but at the cost of consistency, as it will be more likely to read stale values.

In theory, monotonic writes and read-your-writes and etc. should be guaranteed, but in practice, it isn't as there are some tricky edge cases:

- For concurrent writes, it may not be possible to order/prioritize them. A conflict resolution strategy will be needed to resolve the Rsp.
- For concurrent writes + reads, the writes may not have completed to all its replicas and thus the read is indeterminate.
- For partially successful writes (where some nodes couldn't process it for whatever reason) that aren't rolled back, subsequent reads may not return the correct value even when it's supposed to.
- For node recovery, if it restores data from an outdated node (where newer nodes are offline) then the N_w quorum condition may not be satisfied.
- Sloppy quorums may prevent proper read-write overlap
- Other various edge cases that occur due to bad timing.

October 1st 2019

To fully fix the consistency issues, transactions or consensus is needed.

Monitoring Staleness

For leader-based systems, metrics for replication lag are usually easy to keep track of as the lag can be measured by taking the difference between the leader's log position and the replica's.

For leaderless systems, if only read repair is used (so no anti-entropy), there are no limits to the staleness of the data, and even with anti-entropy, ordering is not guaranteed.

The CAP Theorem and Sloppy Quorums

If a network partition occurs, then a large chunk of nodes may not be accessible anymore. If a quorum can no longer be achieved, then according to the CAP theorem, you have to choose between availability or consistency.

For availability, you can continue with the writes anyway and attempt to write it to available nodes instead of the original Nw.

For consistency, errors are returned to all requests that require that quorum.

If you choose availability, what happens is a sloppy quorum. When the partition ends, nodes that temporarily accepted the writes hands it back through a hinted handoff.

Sloppy quorums allow you to accept writes as long as any N_w nodes in the entire system are online, instead of N_w nodes that hold the replicated data.

The tradeoff is that the value received on a read may not be the most up-to-date because it may be stored on a temporary node.

Multi-Datacenter Operation

There are 2 general strategies:

- 1) Simply spread out the N replica nodes across the datacentres. Writes go to all nodes, but the response from the quorum is returned from the local datacenter.

The inter-datacentre writes are done asynchronously to lower latency.

- 2) Keep all communication between one datacenter, and let cross-datacenter happen as an asynchronous background service.

Approach 1) is done by Cassandra and Voldemort, and Riak follows 2).

Concurrent Write Handling

In single-leader systems, concurrent writes are usually handled by ordering them so they're applied sequentially or locking certain rows/tables when

In multi-leader systems, conflict resolution is used.