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 async 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.

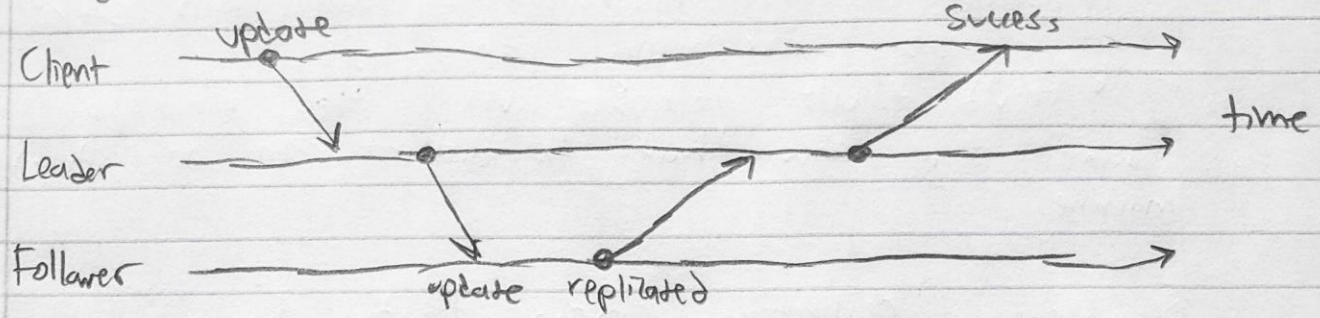
These services use this strategy:

Relational DBs	Non-relational	Message Brokers
MySQL	MongoDB	Kafka
PostgresQL	Rethink DB	RabbitMQ.
Oracle Data Guard	Espresso	

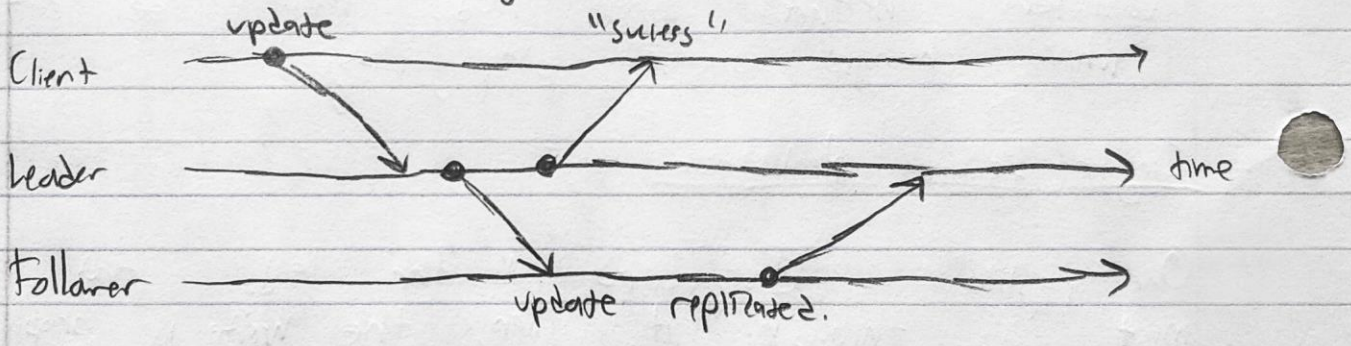
Synchronous vs. Asynchronous Replication

September 7th 2017

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 its impractical to have everything be synchronous.

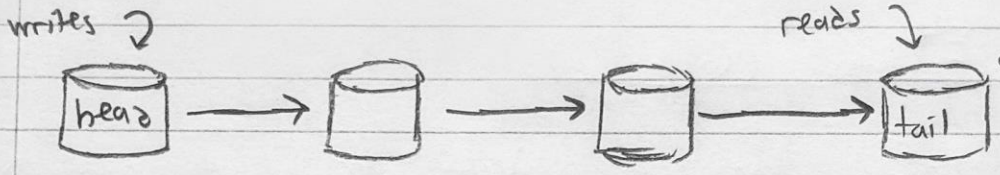
In semi-synchronous systems, SOME of the followers (though usually one, in practice) are synchronous, and the rest aren't.

September 7th 2019

Asynchronous systems will:
→ lose updates if the leader dies, as the replicas never require 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 becames 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.

Recovery

September 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) Reconfigure routing: new writes need to go the new leader
4) Step down old leader: