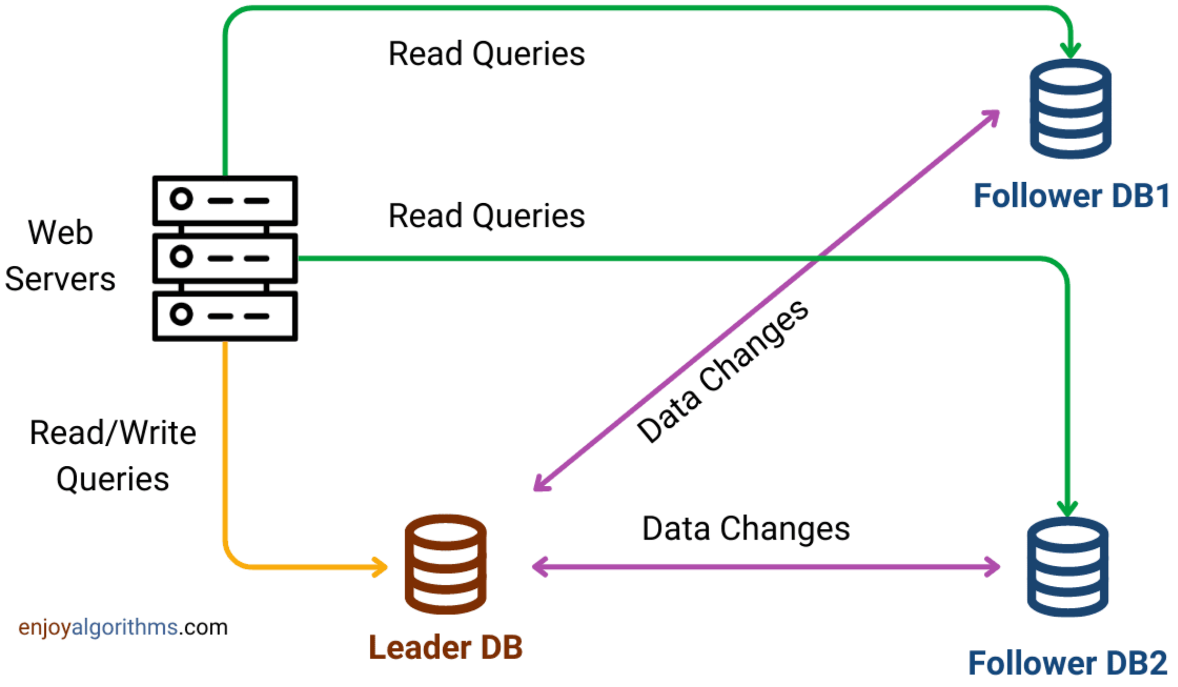
Master-Slave Replication (Single Leader Replication)

In database replication, we create copies of the same database across multiple nodes (replicas). However, with multiple replicas, a question arises: How do we ensure that all the data ends up on all the replicas after every write? The most common solution is master-slave replication, also known as **active/passive** or **single-leader replication.**

What is Master-Slave Replication?

There are two types of nodes in Master-Slave replication: Master and Slave. The single **master (leader)** node works as the primary database, while one or more **slave (follower**) nodes maintain copies of the Master's data.

- The master node is responsible for handling write queries while slave nodes are responsible for handling read queries. Note: If required, the master node can also perform read queries.
- Whenever a write operation is performed on the master node, it is replicated to all the slave nodes to keep the data consistent across the entire system.



The whole idea of master-slave replication works as follows:

- When clients want to perform a write query, they send their requests to the leader, which first writes the new data to its local storage.
- Whenever the leader writes new data to its local storage, it also sends the data change to all of its followers as part of a replication log. Each follower takes the log from the leader and updates its local copy by applying all write in the same order as they were processed on the leader.
- If only one slave database is available and it goes offline, read operations will be temporarily directed to the master database, and a new slave database will replace the old one.
- If multiple slave databases are available, read operations are redirected to other healthy slave databases, and a new database server will replace the old one.
- If the master database goes offline, a slave database will be promoted to be the new master. Now, all write operations will be executed on the new master database, and a new slave database will replace the old one for data replication immediately. **Note:** Promoting a new master is more complicated as the data in a slave database might not be up to date. How to update the missing data? We will

discuss this idea in the later section of this blog.

Use cases of master slave replication

The Master-Slave replication is used in situations where the workload is read-heavy, and there is a need to distribute the read requests across multiple nodes to improve system performance. Since slave nodes can handle read requests, they can offload read traffic from the Master node and allow it to focus on processing write requests.

- This architecture provides fault tolerance and allows the system to continue functioning even if the master node fails.
- Master-Slave replication is used in relational databases such as MySQL, PostgreSQL, and Oracle, as well as in NoSQL databases such as MongoDB, Cassandra, RethinkDB, Espresso, etc.
- Leader-based replication is not restricted to only databases. Distributed message brokers such as Kafka and RabbitMQ also use it to provide highly available queues.

Synchronous Vs Asynchronous Master-Slave Configuration

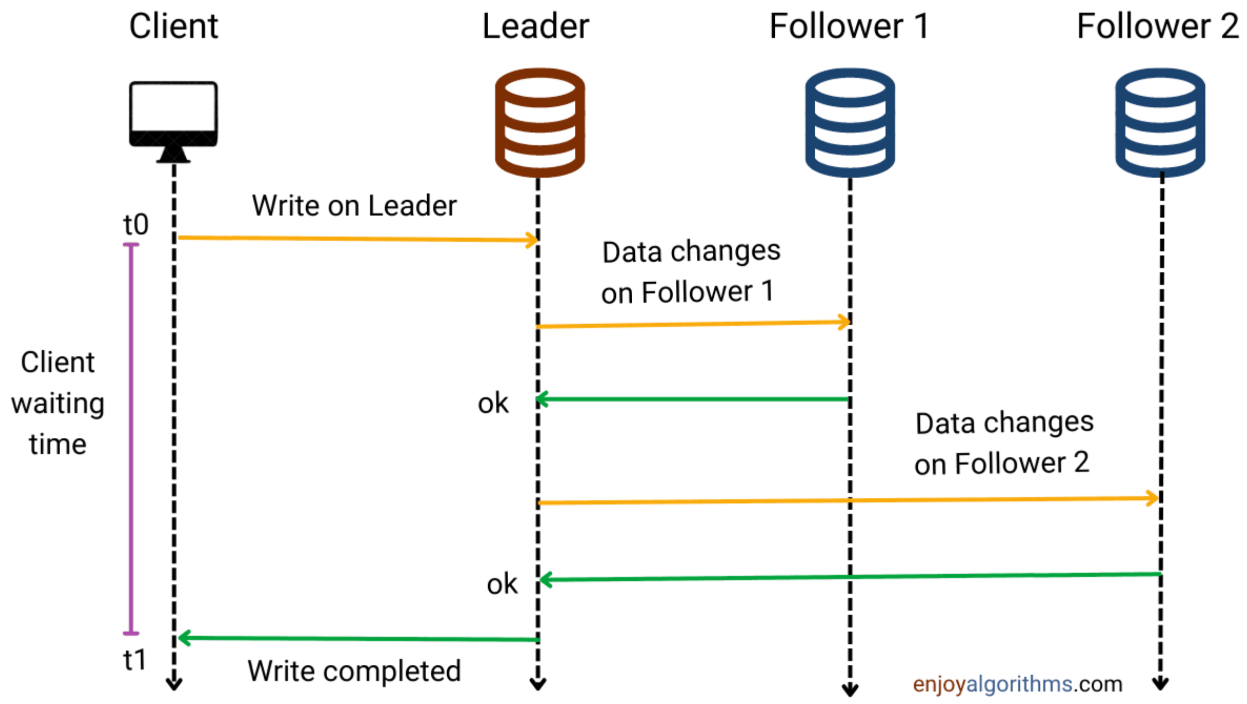
In master-slave replication, an essential aspect to consider is whether replication happens synchronously or asynchronously. To understand this, let's consider what happens when a user updates their profile image on a social media website: the client sends an update request to the leader. As soon as the leader receives the request, it forwards the data change to the followers. Now, the critical question is: How do the followers update their data?

In synchronous replication, the client waits for confirmation from the leader that the update has been applied to all followers before receiving the response. In contrast, asynchronous replication allows the client to receive the response before all followers have been updated.

Advantages and disadvantages of synchronous replication

Disadvantage: If the follower doesn't respond (due to a crash, network fault, or any other reason), the write cannot be processed.

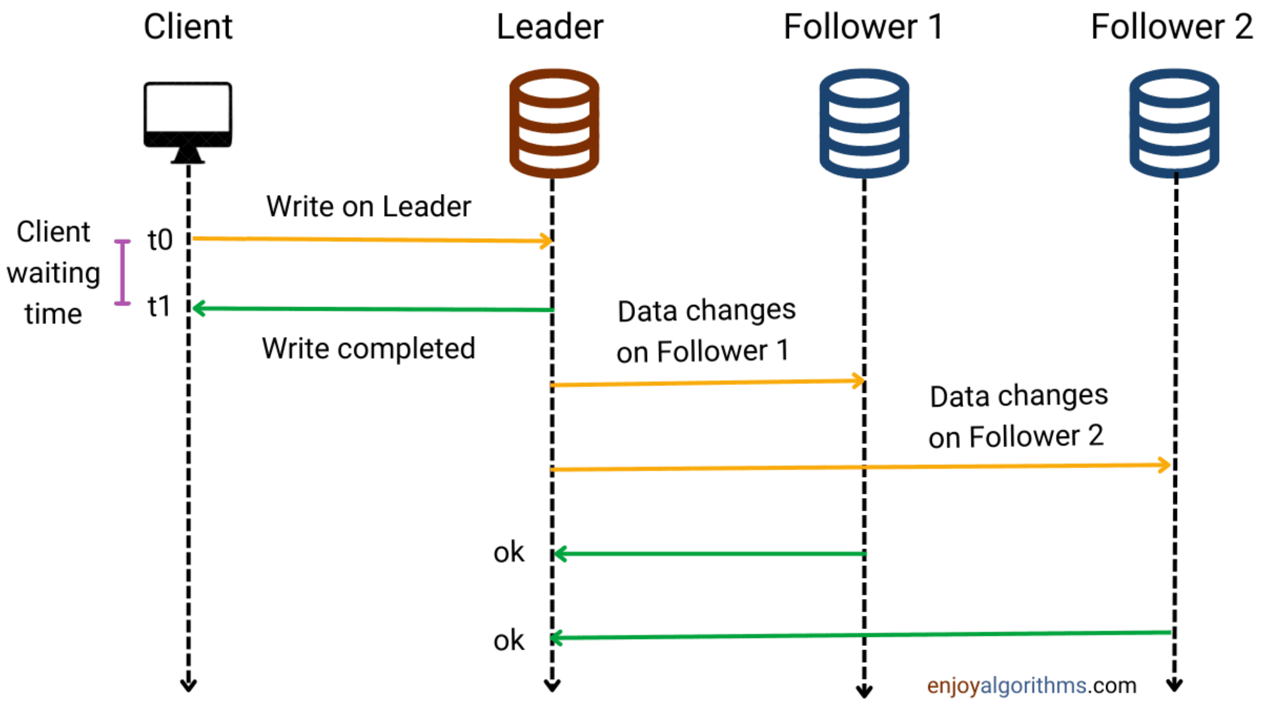
Advantage: The follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader. If the leader suddenly fails, we can be sure that the data is still available to the follower.



Advantages and disadvantages of asynchronous replication

Disadvantage: If the leader fails and is not recoverable, any writes that have not yet been replicated to followers are lost. This means that a write is not guaranteed to be durable, even if it has been confirmed to the client.

Advantage: A leader can continue processing writes, even if all of their followers have fallen behind. Weakening durability may sound like a bad trade-off, but asynchronous replication is nevertheless widely used, especially if there are many followers or if they are geographically distributed.



Overall, the choice of synchronous or asynchronous replication depends on the **trade-offs between consistency and performance**. Synchronous replication provides stronger consistency guarantees, but it can also result in longer response times. On another side, asynchronous replication provides faster response times but may compromise consistency.

Critical ideas to think in asynchronous replication

1. It can be a serious problem in asynchronous replication to lose data if the leader fails. For this reason, researchers have continued to investigate replication methods that provide good performance and availability without sacrificing data integrity. One example of such a method is **chain replication**, which is used in Microsoft Azure Storage.
2. When a user makes a write, the new data may not immediately reach all followers. If the user tries to read the same data immediately after making the write, it may not be available to the follower, leading the user to think that their data was lost.

To handle the above situation, one solution is to read from the leader when reading something that the user may have modified. Otherwise, read from a follower. For example, in a social media platform, profile information is usually editable only by the user. So, it's a good idea to always read a user's own profile from the leader and any other user's profile from a follower. Think!

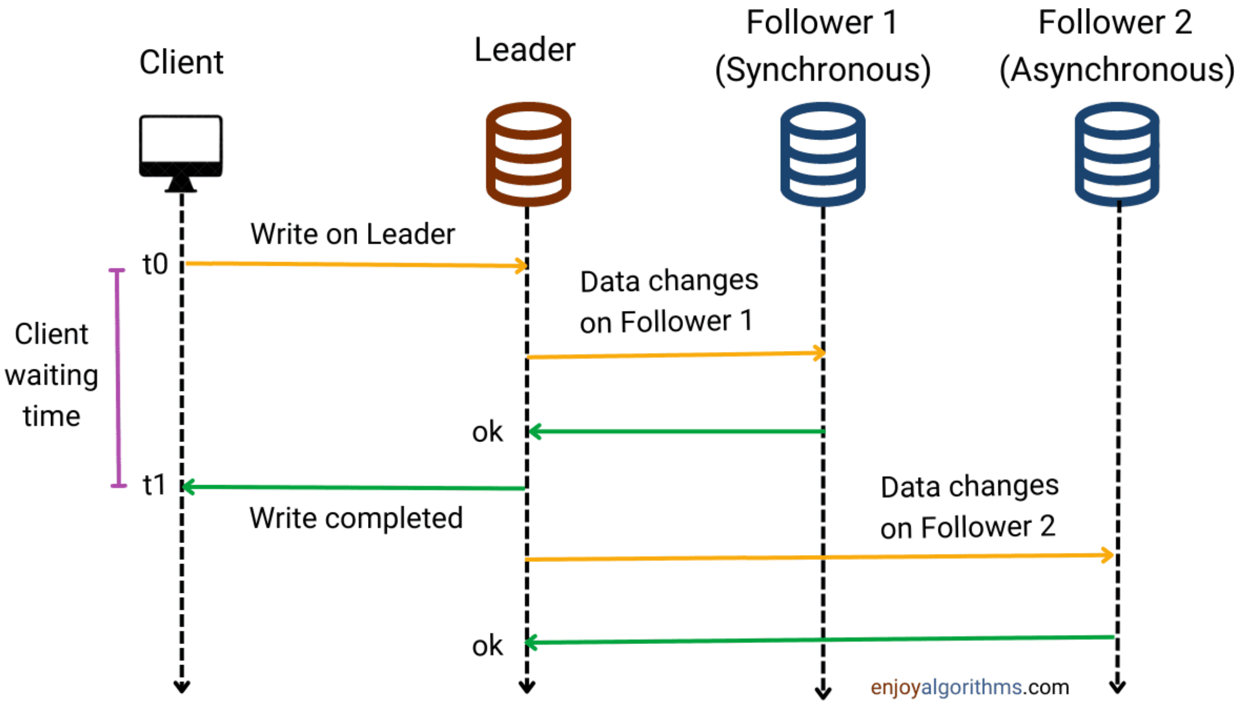
A common ground: Semi-synchronous replication

In most cases, replication is very fast, and changes made to the leader database are quickly propagated to the followers. But there are certain situations where replication can become delayed. For example, if a follower is recovering from a failure, the system is operating at maximum capacity, or there are network problems between the nodes.

On the other hand, synchronous replication requires the leader database to block all write operations until the followers have acknowledged that the data has been received and stored. This can cause significant delays and performance issues if all followers are synchronous. If one of the replicas goes offline, the entire system would be affected.

A good solution would be to use **semi-synchronous replication**. In this setup, one follower is designated as synchronous, while the others are asynchronous. Here synchronous followers will update all data changes in real time and all asynchronous followers will update data eventually in the background. If the synchronous follower goes offline or slows down for any reason, one of the asynchronous followers can be promoted to take its place.

This configuration ensures that at least two nodes, the leader and one synchronous follower, have up-to-date copies of the data at all times. Let's understand this via an example: Suppose a user updates their profile image on the website and there are one leader and two followers. Here replication to follower 1 is synchronous and replication to follower 2 is asynchronous.



- **In the case of follower 1:** The leader waits until follower 1 has confirmed that it received the write before reporting success to the user, and before making the write visible to other clients.
- **In the case of follower 2:** The leader sends the message but doesn't wait for a response from follower 2. The diagram shows that there is a substantial delay before follower 2 processes the message.

Setting up new followers in master slave replication

Suppose we want to increase the number of followers or replace a failed follower node. How can we do this? How can we ensure that the new follower has an accurate copy of the leader's data? Simply copying data from the leader node to the new follower node is not enough to guarantee data consistency because clients are continuously writing to the database. In other words, a standard data copy would see different parts of the database at different points in time.

One solution is to lock the leader or master database to ensure consistency during the copying process. However, this approach would not support the goal of high availability since it would make the leader database unavailable for writes during the copy process.

Fortunately, there is a way to set up a new follower without any downtime. Here are the steps:

1. Take a **consistent snapshot** of the leader's database at a specific point in time. Most databases have this feature, which allows for consistent snapshots without locking the entire database.
2. Copy the snapshot to the new follower node. This ensures that the new follower has a complete and accurate copy of the leader's data at the time the snapshot was taken.
3. Connect the follower to the leader and request all the data changes that have occurred since the snapshot was taken. The new follower must associate the snapshot with a specific position in the leader's **replication log** to request all the data changes since that point. This position is known by different names in different databases, such as the log sequence number in PostgreSQL and the binlog coordinates in MySQL.
4. Finally, the new follower will process the backlog of changes since the snapshot. This process is complete once the new follower has caught up with the leader and can continue processing data changes in real-time.

The practical steps involved in setting up a follower can vary significantly depending on the DBMS system. In some systems, the process is fully automated, while in others, it may require a multi-step workflow that must be manually performed by an administrator.

Handling node failure and outage in master slave replication

Due to faults or errors, any node in a system can go down. So our goal is to keep the system running despite individual node failures and to minimize the impact of a node outage. The critical question is: How can we achieve high availability and reliability with leader-based replication? Let's discuss this scenario separately in case of follower and leader failure.

Handling follower failure

Each follower maintains a log of the data changes it has received from the leader. This log helps the follower to identify the last transaction processed before the fault occurred.

So if a follower fails (event of a crash, restart, or temporary network interruption), it can connect to the leader and request all the data changes that occurred during the time when it was disconnected. Once it has applied these changes, it will have caught up to the leader and can resume receiving a stream of data changes as before.

Handling leader failure

This is a little trickier and requires three critical steps: 1) Detecting the failure of the leader node 2) Promoting one of the followers as a new leader 3) Configuring clients to send their writes to the new leader, and other followers to start consuming data changes from the new leader. This process is also called **failover.**

Step 1: Detecting leader failure: This can happen due to various reasons (crashes, power outages, network issues, etc). Since there is no foolproof way to detect the cause of failure, a timeout is used to assume that a leader node is dead if it doesn't respond for a certain period of time, typically less than 30 sec or 1 minute.

Step 2: Choosing a new leader: This can be done through an election process where a new leader is chosen by a majority of the remaining replicas. To minimize data loss, the replica with the most up-to-date data changes from the old leader is usually chosen as the new leader. But getting all the nodes to agree on a new leader is a consensus problem. **Note:** We will discuss the idea of **the consensus problem** in a separate blog.

Step 3: Reconfiguring the system to use the new leader: Clients need to send their write requests to the new leader. If the old leader comes back online, it might still believe that it is the leader, unaware that it has been replaced by a new leader. The system needs to ensure that the old leader becomes a follower and recognizes the new leader.

Note: In case of semi-synchronous replication, we make the synchronous slave as a new master since we know that it is the most updated one and no data will be lost.

Some critical challenges in handling leader failure!

- There can be a situation in asynchronous replication, where the new leader has not received all the writes from the old leader before it fails. When the former leader rejoins the cluster after a new leader has been chosen, there may be a possibility of conflicting writes. The critical question is: What should happen to those writes? Think and explore!
- In some cases, it is possible for two nodes in a distributed system to believe that they are the leader, which is known as a **split-brain scenario**. This situation can be quite tricky because both leaders may accept writes without any process for resolving conflicts, which can lead to data loss or corruption. How to prevent or handle this split-brain scenario?
- What is the right timeout before the leader is declared failed? A longer timeout means a longer time to recover in the case where the leader fails. However, if the timeout is too short, there could be unnecessary failovers. For example, a temporary increase in load could increase the response time of the node to more than timeout, or a network glitch could cause delayed packets. If the system is already struggling with high load or network problems, an unnecessary failover is likely to make the situation worse, not better.
- There are no easy solutions to these problems. For this reason, some operations teams prefer to perform failovers manually, even if the software supports automatic failover. So, is it better to use Multi-leader replication instead? Problems like node failures, unreliable networks, and trade-offs around replica consistency, durability, availability, and latency are fundamental problems of study in

distributed systems.

References

- Designing data-intensive applications by Martin Kleppmann
- Web Scalability for Startup Engineers by McGraw-Hill
- Understanding distributed systems by Roberto Vitillo

We will keep updating this blog with more insights on master-slave replication. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!