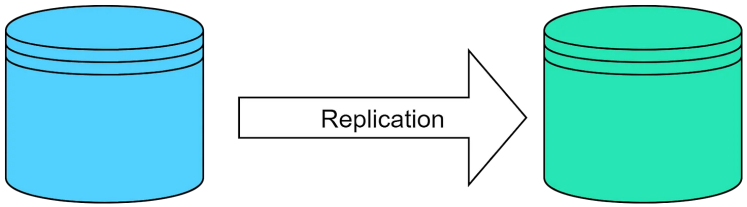
Data Replication: A Key Component for Building Large-Scale Distributed Systems

BYTEBYTEGO
AUG 31, 2023 · PAID

Data replication is critical for building reliable, large-scale distributed systems. In this issue, we will explore common replication strategies and key factors in choosing a suitable strategy.

Throughout this issue, we will use databases as examples. Note that they are not the only data sources where replication can be useful. Replication could apply to cache servers like Redis and even application servers for critical in-memory data structures.

So, what is replication? It's a method of copying data from one place to another. We use it to make sure that our data is available when and where we need it. It helps us improve the durability and availability of our data, reduce latency, and increase bandwidth and throughput.



But choosing a replication strategy isn't always straightforward. There are different strategies, each with its own benefits and drawbacks. Some strategies might be better for certain use cases, while others might be better for different situations.

In this issue, we'll explore three main replication strategies: Leader-Follower, Multi-Leader, and Leaderless. We'll break down what each strategy is, how it works, and where it's most effectively used. We'll discuss the trade-offs involved in each, so we can make informed decisions about the best strategy for our systems.

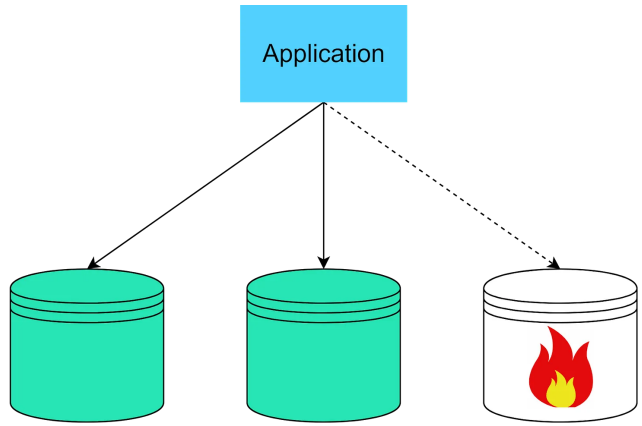
So, let's dive in and start exploring the world of data replication together.

A Primer on Replication

Let’s examine at a high level why replication is needed. As we mentioned earlier, we’ll use databases as examples throughout, but this applies to other types of data sources as well.

Improving Durability

Improving durability is perhaps the single most important reason for data replication. When a single database server fails, it could lead to catastrophic data loss and downtime. If the data is replicated to other database servers, the data is preserved even if one server goes down. Some replication strategies, like asynchronous replication, may still result in a small amount of data loss, but overall durability is greatly improved.



You might be wondering: Isn’t regular data backup sufficient for durability? Backups can certainly recover data after disasters like hardware failure. But backups alone have limitations for durability. Backups are periodic, so some data loss is likely between backup cycles. Restoring from backups is also slow and results in downtime. Replication to standby servers provides additional durability by eliminating (or greatly reducing) data loss windows and allowing faster failover. Backups and replication together provide both data recovery and minimized downtime.

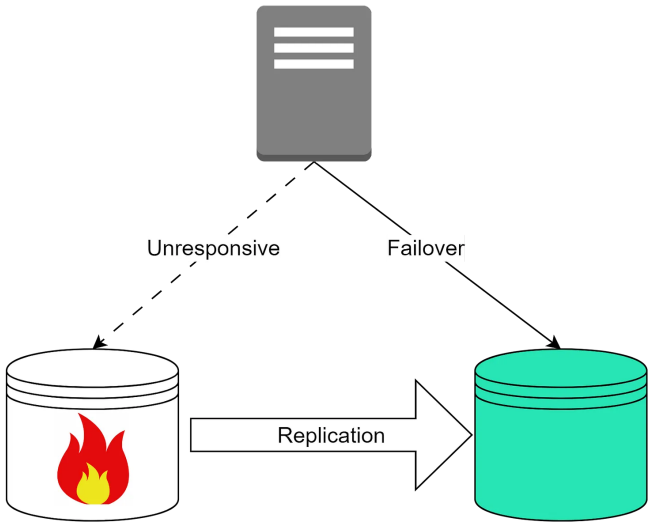
Improving Availability

Another critical reason to replicate data is to improve overall system availability and resilience. When one database server goes offline or gets overwhelmed, keeping applications running smoothly can be challenging.

Simply redirecting traffic to a new server is non-trivial. The new node needs to already have a nearly identical copy of the data to take over quickly. And switching databases behind-the-scenes while maintaining continuous uptime for applications and users requires careful failover orchestration.

Replication enables seamless failover by keeping standby servers ready with up-to-date data copies. Applications can redirect traffic to replicas when issues occur with minimal downtime. Well-designed systems automatically handle redirection and failure recovery via monitoring, load balancing, and replication configurations.

Of course, replication has its own overhead costs and complexities. But without replication, a single server outage could mean prolonged downtime. Replication maintains availability despite outages.



Increasing Throughput

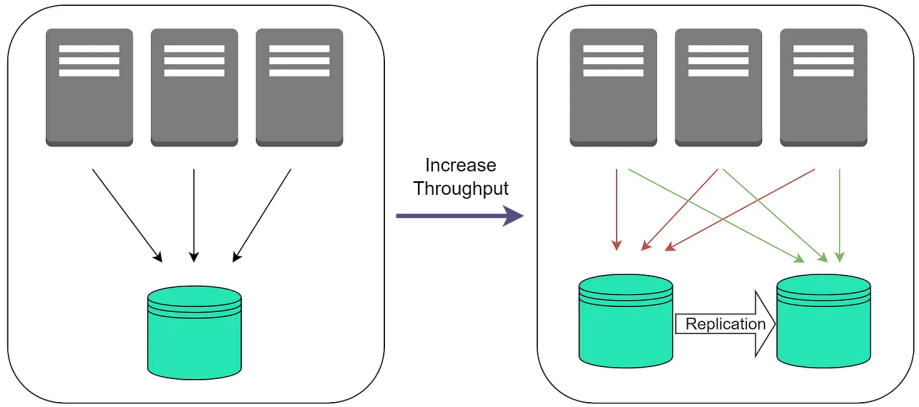
Replicating data across multiple database instances also increases total system throughput and scalability by spreading load across nodes.

With a single database server, there is a maximum threshold of concurrent reads and writes it can handle before performance degrades. By replicating to multiple servers, application requests can be distributed across replicas. More replicas means more capacity to handle load in parallel.

This sharding of requests distributes workload. It allows the overall system to sustain much higher throughput compared to a single server. Additional replicas can be added to scale out capacity even further as needed.

The replication itself has associated overheads that can become bottlenecks if not managed properly. Factors like inter-node network bandwidth, replication lag, and write coordination should be monitored.

But proper replication configurations allow horizontally scaling out read and write capacity. This enables massive aggregated throughput and workload scalability far beyond a single server's limits.



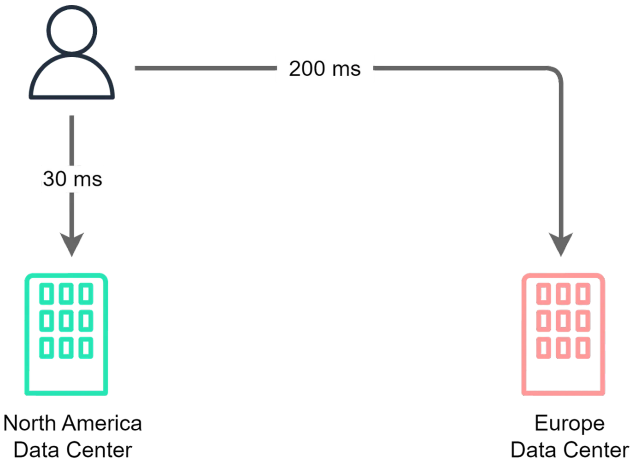
Reducing Latency

Data replication can also improve latency by locating data closer to users. For example, replicating a database to multiple geographic regions brings data copies closer to local users. This reduces the physical network distance that data has to travel compared to a single centralized database location.

Shorter network distance means lower transmission latency. So users' read and write requests see faster response times when routed to a nearby replicated instance versus one further away. Multi-region replication enables localized processing that avoids the high latencies of cross-country or intercontinental network routes.

Keep in mind that distributing copies across regions introduces complexities like replica synchronization, consistency, and conflict resolution with concurrent multi-site updates. Solutions like consistency models, conflict resolution logic, and replication protocols help manage this complexity.

When applicable, multi-region replication provides major latency improvements for geo-distributed users and workloads by localizing processing. The lower latency also improves user experience and productivity.



Replication Protocols

Certain replication strategies rely on protocols like Paxos and Raft to coordinate and reach consensus between nodes.

Paxos is a protocol that allows nodes in a distributed system to come to consensus about data values or the state of the system. It ensures all nodes agree on updates even in complex failure scenarios. Paxos elects a node as the leader to coordinate consensus. The leader sends proposed updates to replicas, who approve or reject them. Paxos is utilized in systems like CockroachDB.

Raft is another consensus protocol using an elected leader. It's considered simpler than Paxos. The leader sends heartbeat messages to maintain its leadership. If followers miss enough heartbeats, a new leader is elected. Raft is commonly used for leader election in systems like etcd.

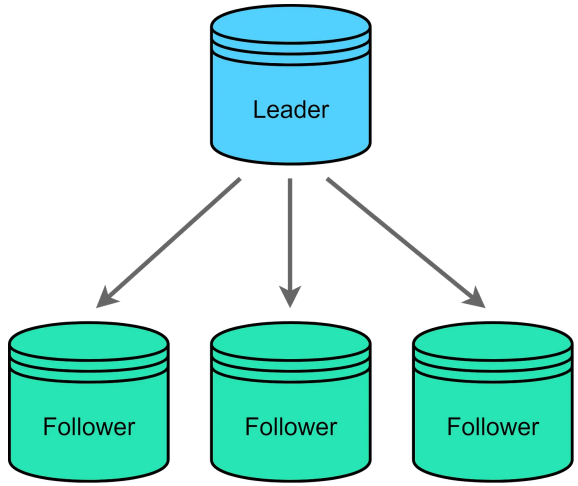
These protocols facilitate replication strategies like leader-follower by providing consensus mechanisms between nodes. Understanding how they enable coordination and agreement helps shed light on the inner workings of various replication models.

Now that we've covered the major motivations behind replication, let's examine some common replication strategies and how they help address durability, availability, scalability, and latency needs.

Leader-follower replication

The leader-follower model plays a significant role in data replication. It's a model we often use when we want to balance load between nodes.

In this model, the leader node handles all writes, while reads can go to any nodes, including the followers. This allows our system to handle more requests by parallelizing reads and writes across nodes.

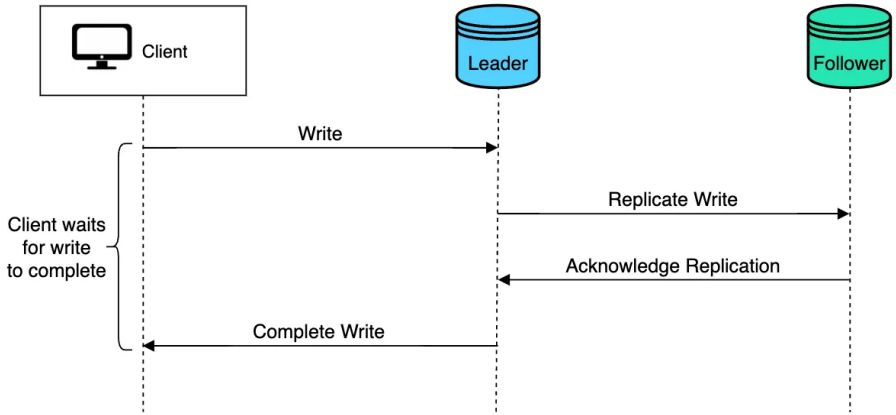


There are two broad types of leader-follower replication: synchronous and asynchronous replication.

Synchronous Replication

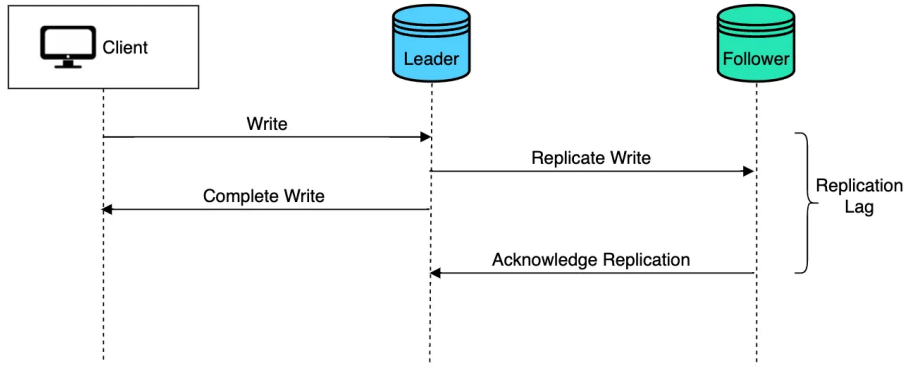
With synchronous replication, each write to the leader waits for acknowledgement that data is replicated to followers before completing. This ensures strong data consistency, as followers are guaranteed to have the latest writes before the next write occurs.

However, this safety comes at a cost. The extra network round trips for replication confirmation add latency to writes. This can reduce performance, especially with high write volume or if the nodes are geographically distributed.

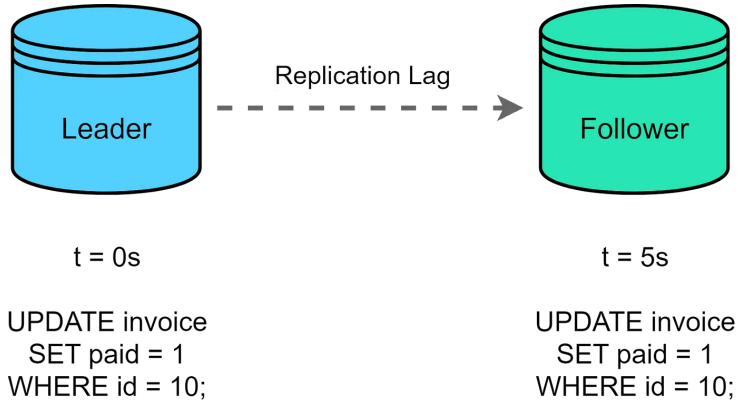


Asynchronous Replication

Asynchronous replication improves performance by allowing the leader to complete writes without waiting for replication confirmation. This reduces write latency significantly. Follower failures also don't impact write availability.



However, there's a risk here. If the follower node doesn't replicate the data fast enough, and the leader node fails, we could lose some data. Another disadvantage is that there is a small window where the data between the leader and the followers is inconsistent. The time it takes for the data to be replicated from the leader to the followers is called replication lag. The user might get a different result of a recent write due to this replication lag.



We need to carefully weigh tradeoffs between consistency and availability/performance. Synchronous replication offers stronger consistency. Asynchronous replication provides higher availability and performance.

In practice, asynchronous replication is common since performance is crucial. But replication lag requires additional strategies to mitigate consistency issues.

Consistency Models

So far we have focused on strong and eventual consistency models. But there are other consistency models that provide different guarantees and have implications for replication strategies.

Linearizability provides very strong ordering guarantees, ensuring reads instantly reflect the latest writes. This necessitates synchronous replication to keep replicas synchronized.

Sequential consistency guarantees ordering of writes across nodes, but allows some lag in propagating writes before reads. This enables some asynchronous replication flexibility.

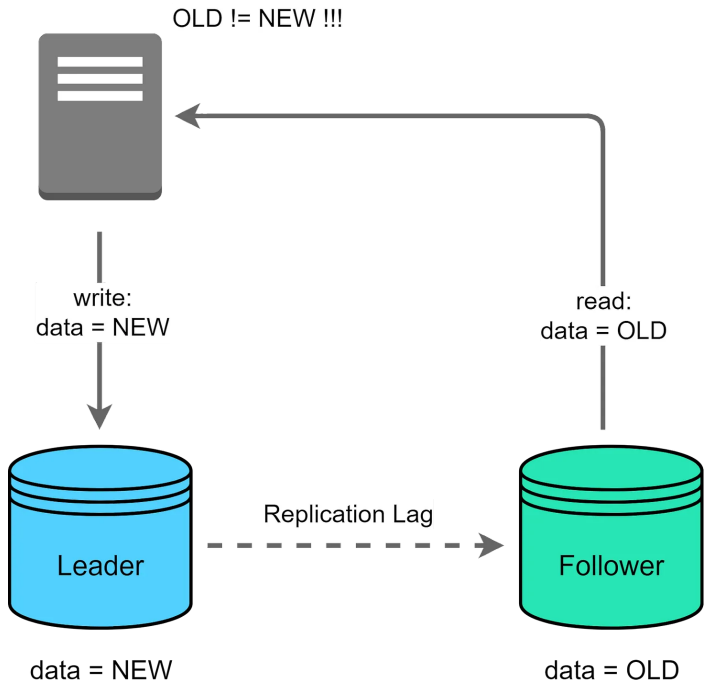
Causal consistency only enforces order between related writes. Unrelated writes can be seen out of order. This further relaxes synchronicity needs for asynchronous replication.

These stronger models add ordering constraints that limit how asynchronous replication can be used. Weaker consistency models provide more flexibility but increase the risk of stale reads. Understanding these trade-offs helps guide the choice of replication strategy.

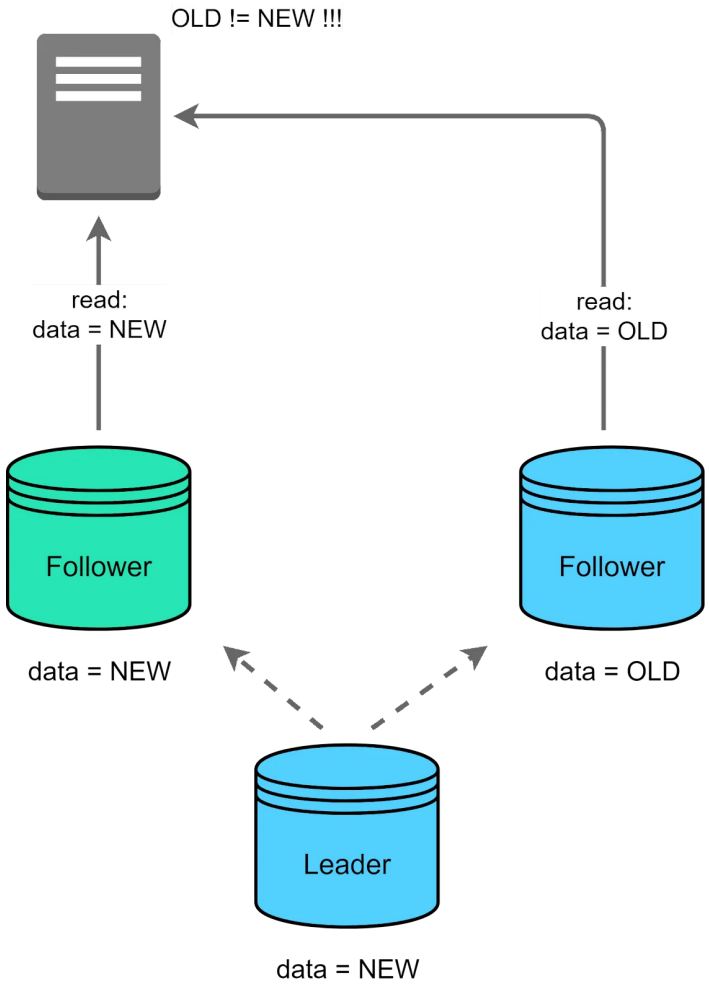
The flexibility of asynchronous replication comes at the cost of replication lag between leaders and followers. This lag can lead to a number of consistency issues that must be considered.

Consistency Issues

The first is “read-your-own-write” inconsistency. Imagine a user just uploads a new profile picture then immediately views their profile, they may still see the old picture due to replication lag. This delay is a result of the time it takes for the new picture (a write operation) to be replicated to the follower node (where the read operation is usually happening). This inconsistency can confuse the user and even make them question whether the upload was successful. Maintaining read-your-own-write consistency is crucial for a smooth user experience. One way to handle this is to direct reads after writes to the leader node to avoid replication lag issues.



Another potential problem is inconsistent reads from different replicas. This is not uncommon if there is a load balancer that sits in front of the read replicas to distribute load across them in a round robin fashion. This inconsistency might be acceptable for some applications but not for others.



Tunable Consistency

Many modern systems allow tuning the consistency level on a per-operation basis. This allows applications to balance strong consistency vs. high availability and performance depending on their needs.

For example, an application might use strong consistency for financial transactions to ensure accurate account balances. But for analytics queries, eventual consistency could be acceptable in order to improve performance and availability.

Similarly, synchronous replication might be used for a user's initial login to guarantee read-your-own-writes consistency. But asynchronous replication could handle less critical backend updates to maximize throughput.

The ability to tune consistency levels adds flexibility in balancing the tradeoffs between replication strategies. The optimal balance depends on the specific application's functional needs, performance requirements and tolerance for inconsistency.

Handling Failures

Let's examine strategies for handling node failures with leader-follower replication.

Follower Failures

We could use periodic health checks to detect if a follower has failed. Once we've detected the failed node, we take immediate action and redirect the traffic that was going to the failed follower to a healthy one. At the same time, we work to replace or restart the failed follower to get it back online.

However, redirecting traffic to near-capacity replicas runs the risk of overloading and crashing the entire cluster. Available capacity must be considered carefully before shifting traffic.

Leader Failures

The situation becomes more complex if the leader node fails. We can't simply redirect traffic because writes can only happen on the leader.

Heartbeat checks help detect a failed leader. When the leader fails, we either manually promote a follower to become the new leader, or use automated leader election protocol like Raft to elect a new leader from among the follower nodes.

The downside is that replacing a leader, either manually or through an election process, takes time, and our system might experience a temporary disruption to write operations. Once the new leader is in place, normal operations can resume. This delay might not be acceptable for some systems.

This covers the basics of leader-follower replication. It's a robust and popular model, but it's not the only one. In the next part of our series, we'll go beyond leader-

follower and explore two additional models - multi-leader and leaderless replication.

We'll compare their unique designs, examine use cases where they excel, and provide guidance on choosing the right strategy.

175 Likes · 5 Restacks

2 Comments

Write a comment...

Ray Li Nov 29, 2023

One question needs help,

For asynchronous replication part, it writes 'If the follower node doesn't replicate the data fast enough, and the leader node fails, we could lose some data', I have a question about it, how can this happen? If the leader finishes writing data and finish writing binlog, and then it crashes without sync data to followers, the followers may loss this data, but if we can check the binlog of leader manually and sync the missing binlog to followers to repair the data loss?

LIKE REPLY SHARE

Bahdan Sep 11, 2023

Good stuff!

LIKE REPLY SHARE