04 - Data Replication

Why Replicate Data?

Data replication involves storing copies of the same data on multiple machines to enhance performance, availability, and reliability. Distributed systems commonly use replication to manage high data loads efficiently.

Benefits of Data Replication

1. Scalability & High Throughput – As data volume and read/write requests grow, a single machine cannot handle the load. Replication allows multiple machines to distribute the workload.
2. Fault Tolerance & High Availability – If a machine fails, another machine with a copy of the data can take over, ensuring system reliability.
3. Low Latency for Global Users – Replicating data across geographically distributed servers reduces access time for users worldwide.

Challenges of Distributed Data

While replication offers advantages, it introduces several challenges:

- Consistency – Changes made on one machine must be propagated across the network to all copies.
- Application Complexity – Managing reads and writes across multiple replicated nodes often falls to the application.
- Partitioning – Ensuring data is properly segmented without conflicts.

Scaling Strategies: Vertical vs. Horizontal

Vertical Scaling ("Scaling Up")

- Uses a single, powerful machine with more CPU, memory, or disk space.
- Some fault tolerance exists through hot-swappable components (replaceable parts without shutting down the system).
- Example: A single high-end database server handling increasing traffic.
- Limitations: Expensive, hardware limitations, and a single point of failure.

Horizontal Scaling ("Scaling Out")

- Uses multiple machines, each with its own CPU, memory, and disk.
- Coordination happens via the application layer over a network.
- Geographically distributed, improving reliability and performance.

- Example: A NoSQL database cluster like MongoDB or Cassandra, where each node handles a portion of the data.
- Benefits:
    - Better fault tolerance (failure of one machine doesn't bring down the system).
    - Easier cost management (commodity hardware is cheaper than high-end servers).

Replication vs. Partitioning

- Replication: Multiple machines store the same copy of the data.
- Partitioning: Each machine stores only a subset of the data.

Example:

- A global e-commerce platform might replicate customer accounts across all servers for high availability, while partitioning order history by region.

Common Replication Strategies

Distributed databases typically use one of the following replication models:

1. Single Leader Model
    - All write operations go to a single leader node.
    - The leader sends replication updates to follower nodes.
    - Clients can read from either the leader or followers.
    - Example: MySQL, PostgreSQL, SQL Server.
2. Multiple Leader Model
    - Multiple nodes act as leaders, allowing writes on different nodes.
    - Requires conflict resolution strategies if nodes receive conflicting updates.
    - Example: Google Spanner, DynamoDB.
3. Leaderless Model
    - Any node can receive write requests.
    - Conflict resolution happens during reads (Eventual Consistency).
    - Example: Amazon DynamoDB, Cassandra.

Leader-Based Replication

How It Works:

1. Clients send write requests only to the leader.
2. The leader sends replication updates to followers.
3. Followers apply the updates.
4. Clients can read data from either the leader or followers.

Common Implementations:

- Relational Databases: MySQL, Oracle, SQL Server.
- NoSQL Databases: MongoDB, RethinkDB (for real-time apps), Espresso (LinkedIn).
- Messaging Brokers: Kafka, RabbitMQ.

How Replication Information Is Transmitted

Replication can be performed using different methods:

- Statement-based replication: Sends SQL statements (INSERT, UPDATE, DELETE).
    - Downside: May cause errors with non-deterministic functions (e.g., NOW()).
- Write-Ahead Log (WAL) replication: Copies byte-level changes.
    - Downside: Requires all nodes to use the same storage engine.
- Logical (row-based) replication: Transfers only modified rows.
    - Advantage: Decoupled from storage engine, making parsing easier.
- Trigger-based replication: Uses database triggers to log changes.
    - Advantage: Application-specific flexibility but can introduce overhead.

Synchronous vs. Asynchronous Replication

Synchronous Replication

- Leader waits for a response from followers before committing changes.
- Ensures strong consistency (every node has the latest data).
- Downside: Slower writes due to waiting.

Asynchronous Replication

- Leader does not wait for confirmation from followers.
- Faster writes but may result in temporary inconsistencies (eventual consistency).
- Advantage: Better performance, suitable for high-availability applications.

Handling Leader Failures

When a leader fails, a new leader must be chosen. Challenges include:

- Selecting a new leader:
    - Use a consensus algorithm (e.g., nodes vote for the most up-to-date node).
    - Assign a controller node to manage leadership changes.
- Ensuring data consistency:
    - If using asynchronous replication, some writes may be lost.
    - Strategies include replaying logs or discarding lost writes.

- Preventing split-brain scenarios:
    - If multiple nodes act as leaders, conflicting writes may occur.
    - Requires conflict resolution mechanisms.

Replication Lag

Replication lag is the delay between a write on the leader and the update appearing on followers.

- Synchronous replication:
    - Minimal lag, but slower writes as followers must confirm every update.
- Asynchronous replication:
    - Higher lag, but better performance (eventual consistency).

The delay in consistency caused by replication lag is known as the inconsistency window.

Read-After-Write Consistency

Scenario:

You comment on a Reddit post. After submitting, you expect your comment to be visible immediately, but it's less critical for others to see it instantly.

Methods to Ensure Read-After-Write Consistency:

1. Always read recent updates from the leader.
2. Dynamically switch to leader reads for recently updated data (e.g., for 1 minute after an update).

Monotonic Read Consistency

- Issue: If a user reads data from multiple followers, they might see older data after reading newer data.
- Solution: Monotonic Read Consistency ensures that users never see outdated data after previously reading updated data.

Consistent Prefix Reads

- Issue: If different partitions replicate data at different speeds, reads may return out-of-order updates.
- Solution: Consistent Prefix Reads guarantee that updates appear in the correct order, maintaining logical consistency.

Example:

- A user posts Comment A, then Comment B. If replication delays occur, followers might see Comment B before A, causing confusion.
- With consistent prefix reads, users always see A before B as intended.

Key Takeaways

- Replication improves availability, fault tolerance, and scalability.
- Different replication models balance consistency, performance, and complexity.
- Leader failures require careful handling to maintain data integrity.
- Read consistency mechanisms prevent users from seeing outdated or out-of-order data.