Data Replication in Distributed Systems
Briefing Document: Data Replication in Distributed Systems
Source: Excerpts from "04 - Data Replication.pdf" by Mark Fontenot, PhD, Northeastern University
Date: October 26, 2023
Overview: This briefing document summarizes the key concepts and challenges associated with data replication in distributed systems, as presented in the provided lecture notes. The document outlines the benefits of distributing data, the fundamental strategies for replication, the various methods for propagating replication information, and the crucial considerations around consistency, fault tolerance, and leader election.
Main Themes and Important Ideas/Facts:
1. Benefits and Challenges of Distributed Data:
•
Benefits: Distributing data offers significant advantages, including:
◦
Scalability / High throughput: Enables systems to handle growing data volumes and read/write loads beyond the capacity of a single machine. As stated, this addresses situations where "Data volume or Read/Write load grows beyond the capacity of a single machine."
◦
Fault Tolerance / High Availability: Ensures application continuity even if some machines fail. The source highlights that this is crucial when "Your application needs to continue working even if one or more machines goes down."
◦
Latency: Improves performance for geographically dispersed users by allowing data to be closer to them. The note mentions the need to provide "fast performance" to users in "different parts of the world."
•
Challenges: Distributing data introduces complexities:
◦
Consistency: Maintaining data consistency across multiple replicas is a primary challenge, as "Updates must be propagated across the network."
◦
Application Complexity: The responsibility for managing data reads and writes in a distributed environment often shifts to the application layer.
2. Scaling Architectures:
•
The document briefly contrasts different scaling architectures:
◦
Vertical Scaling (Shared Memory & Shared Disk): While offering some fault tolerance (hot-swappable components in shared memory) or suitability for high-read workloads (shared disk for data warehouses), they face limitations in scalability due to centralized nature or contention and locking overhead. The exorbitant cost of high-end vertical scaling is also highlighted with the example of a "$78,000/month" AWS instance.
◦

Horizontal Scaling (Shared Nothing): This architecture, where "Each node has its own CPU, memory, and disk," is presented as a solution for geographic distribution and utilizing "Commodity hardware." Coordination is managed at the application layer.

3. Replication vs. Partitioning:

- The document clearly distinguishes between replication and partitioning:
    - **Replication:** Involves maintaining copies ("Replicates have same data as Main") of the entire dataset across multiple nodes.
    - **Partitioning:** Involves dividing the dataset into subsets ("Partitions have a subset of the data") and distributing these subsets across nodes.

4. Common Replication Strategies:

- Distributed databases commonly adopt one of the following replication strategies:
    - **Single leader model:** All write operations are directed to a single leader node, which then propagates the changes to follower nodes. Clients can read from either the leader or the followers. This is described as a "Very Common Strategy" and examples include relational databases like "MySQL, Oracle, SQL Server, PostgreSQL" and NoSQL databases like "MongoDB, RethinkDB (realtime web apps), Espresso (LinkedIn)," as well as messaging brokers like "Kafka, RabbitMQ."
    - **Multiple leader model:** Allows writes to be accepted by multiple leader nodes, which then need to coordinate and resolve potential conflicts. (This is implied but not detailed in the excerpt).
    - **Leaderless model:** Writes can be accepted by any replica, and mechanisms like quorums are used to ensure consistency. (This is implied but not detailed in the excerpt).

5. Methods for Transmitting Replication Information:

- The document outlines various techniques for propagating write operations from the leader to the followers:
    - **Statement-based:** Sends the SQL `INSERT`, `UPDATE`, and `DELETE` statements to the replicas. While "Simple," it is considered "error-prone due to non-deterministic functions like now(), trigger side-effects, and difficulty in handling concurrent transactions."
    - **Write-ahead Log (WAL):** Transmits a byte-level log of every change to the database. This requires that "Leader and all followers must implement the same storage engine and makes upgrades difficult."
    - **Logical (row-based) Log:** For relational databases, this involves logging "Inserted rows, modified rows (before and after), deleted rows." This method is "decoupled from the storage engine and easier to parse."

    - **Trigger-based:** Changes are logged to a separate table whenever a trigger fires. This is "Flexible because you can have application specific replication, but also more error prone."

6. Synchronous vs. Asynchronous Replication:

- This section highlights the trade-offs between different consistency guarantees and performance:
    - **Synchronous Replication:** The leader waits for confirmation from one or more followers before acknowledging the write to the client. (The excerpt notes "Leader waits for a response from the follower").
    - **Asynchronous Replication:** The leader does not wait for confirmation from followers. While providing higher availability and lower latency for writes, it introduces the possibility of data inconsistency.

7. Leader Failure Challenges:

- The failure of the leader node presents several significant challenges:
    - **Leader Election:** A new leader must be chosen, potentially using a "Consensus strategy – perhaps based on who has the most updates?" or a "controller node to appoint new leader?"
    - **Client Reconfiguration:** Clients need to be updated to direct their write operations to the new leader.
    - **Data Loss (Asynchronous Replication):** If asynchronous replication is used, the new leader might not have received all the writes from the failed leader. Decisions need to be made on how to "recover the lost writes?" or whether to "simply discard?" them.
    - **Split Brain:** Recovering the old leader can lead to a "Split brain: no way to resolve conflicting requests" scenario where two nodes believe they are the leader.
    - **Leader Failure Detection:** Determining when a leader has truly failed requires setting an "Optimal timeout" which is "tricky."

8. Replication Lag and Consistency:

- **Replication Lag:** Refers to the delay between a write on the leader and its reflection on the followers.
    - Synchronous replication can slow down writes and increase system brittleness with more followers.
    - Asynchronous replication maintains availability "but at the cost of delayed or eventual consistency. This delay is called the inconsistency window."

- **Read-After-Write Consistency:** Ensures that after a user performs a write, their subsequent reads will reflect that write.
    - **Method 1:** Always read modifiable data from the leader.
    - **Method 2:** Dynamically switch to reading from the leader for "recently updated" data (e.g., within one minute of the last update).
    - However, these methods can create challenges by requiring reads of modifiable data to be routed to potentially distant leaders, negating the benefit of having proximal followers.
- **Monotonic Read Consistency:** Prevents a user from seeing older data after having already seen newer data in a sequence of reads from multiple followers. This addresses "monotonic read anomalies: occur when a user reads values out of order from multiple followers." It "ensures that when a user makes multiple reads, they will not read older data after previously reading newer data."
- **Consistent Prefix Reads:** Guarantees that if a series of writes occur in a specific order, any reader will observe these writes appearing in the same order. This is important because "Reading data out of order can occur if different partitions replicate data at different rates. There is no global write consistency." The guarantee "ensures that if a sequence of writes happens in a certain order, anyone reading those writes will see them appear in the same order."

Conclusion:

The provided material offers a foundational understanding of data replication in distributed systems. It highlights the essential trade-offs between scalability, fault tolerance, latency, and consistency. The document underscores the complexity introduced by replication, particularly in managing leader election, handling failures, and ensuring various levels of read and write consistency. Understanding these concepts and challenges is crucial for designing and operating reliable and performant distributed applications.