Vidyalankar Institute of Technology, Accredited A+ by NAAC	**DEPARTMENT OF COMPUTER ENGINEERING**

Assignment No. 09

Semester	B.E. Semester VIII – Computer Engineering
Subject	Distributed Computing Lab
Subject Professor In-charge	Dr. Umesh Kulkarni
Assisting Professor	Prof. Prakash Parmar
Academic Year	2024-25

Student Name	Deep Salunkhe
Roll Number	21102A0014

Title: Balancing the Trade-off Between Strong Consistency and Performance in a Global System

---

In a global distributed system, achieving **strong consistency** while maintaining **high performance** is challenging due to network latency, partitioning, and availability constraints. Systems must carefully balance these two opposing goals to ensure usability while adhering to real-world constraints such as **CAP theorem** and **PACELC theorem**.

---

1. Trade-off Between Strong Consistency and Performance

A. Why is Strong Consistency Expensive?

Strong consistency requires that all replicas in a distributed system see the same data at any given time. This typically involves mechanisms such as:

- **Synchronous replication** (e.g., Two-Phase Commit, Paxos, Raft)
- **Global locks or coordination mechanisms**
- **Serialization of requests across distributed nodes**

These methods introduce **higher latency** and can degrade system responsiveness, especially in **geo-distributed systems** where network delays are unpredictable.

B. Performance Considerations

To achieve high performance, systems often:

- Use **asynchronous replication** to avoid blocking writes.
- Employ **eventual consistency**, where updates propagate over time rather than instantly.
- Utilize **geographically distributed caching** and **edge computing** to reduce access latency.

Thus, there is an inherent trade-off:

- **Strong consistency** ensures correctness but increases response time.
- **Weaker consistency** (such as eventual consistency) improves responsiveness but may cause **stale reads** and **temporary inconsistencies**.

---

2. Strategies to Balance the Trade-off

Several approaches exist to strike a balance between strong consistency and performance:

A. Hybrid Consistency Models

- **Tunable Consistency:** Systems like Cassandra and DynamoDB allow users to configure read/write quorum levels, enabling a trade-off between latency and consistency.
- **Session Guarantees:** Implementing **session consistency** ensures that users see their own updates, even if strong consistency is relaxed across multiple users.
- **Read-Your-Write Guarantees:** A weaker form of consistency ensures a user sees their own updates without enforcing global ordering.

B. Geo-Replication with Conflict Resolution

- **Primary-Backup Model:** A leader node handles writes, and replicas sync asynchronously to improve performance.
- **Conflict-Free Replicated Data Types (CRDTs):** Used in systems like Riak and Redis to merge updates without strong coordination.
- **Vector Clocks & Versioning:** Helps detect conflicting updates and apply resolution strategies.

C. Multi-Version Concurrency Control (MVCC)

- Used in databases like **PostgreSQL** and **Spanner** to allow concurrent access without blocking reads, providing snapshots of data while maintaining transactional consistency.

D. Partitioning and Localized Consistency

- **Partition-Tolerant Databases (e.g., DynamoDB, Cassandra):** Use eventual consistency within a partition while minimizing cross-region consistency constraints.
- **Geo-Partitioning:** Assigns users to a specific region to reduce cross-region latency while maintaining stronger consistency within each region.

---

3. Client-Centric Consistency Model for Usability

When dealing with **user-facing inconsistencies**, **Client-Centric Consistency Models** ensure that **individual users** experience a smooth and predictable interaction with the system, even if global consistency is relaxed. A suitable model for usability is:

A. Read-Your-Writes Consistency (RYW)

- Guarantees that a user always sees their most recent updates.
- Example: A user posts a comment on a social media platform, and the system ensures they see their comment immediately, even if other users see it with a delay.

B. Monotonic Reads

- Ensures that once a user sees a version of data, they will not see an **older version** in subsequent reads.
- Prevents **time-travel anomalies** where stale reads cause confusion.
- Example: If a user checks their bank balance and sees ₹10,000, they should never see ₹9,500 later unless a transaction occurs.

C. Monotonic Writes

- Ensures that a user's updates are applied in the correct order.
- Example: If a user updates their email, the system ensures that the new email is not overwritten by an older version due to delayed propagation.

D. Session Consistency

- Guarantees that within a session, the user experiences a consistent view of data.
- Example: In a cloud document editing system, a user should see their latest edits across multiple requests without inconsistencies.

4. Practical Implementation Example

A global e-commerce system like **Amazon** may implement:

- **Eventual consistency for product inventory** (fast performance, minor inconsistencies).
- **Read-your-writes for user carts and order history** (ensures correctness per user).
- **Geo-replicated databases with partitioning** to balance strong consistency in order processing but weak consistency in product availability.

5. Conclusion

The **optimal balance** between strong consistency and performance depends on the **use case**:

- **Financial systems** (e.g., banking) → Strong consistency is critical.
- **Social media** (e.g., Twitter, Facebook) → Eventual consistency with client-centric guarantees suffices.
- **E-commerce** (e.g., Amazon) → Hybrid models with tunable consistency.

By applying **client-centric consistency models** like **Read-Your-Writes and Monotonic Reads**, usability issues can be addressed while still leveraging **eventual consistency** to maintain system scalability and performance.