Strong vs Eventual Consistency

Consistency in distributed systems refers to the guarantee that all nodes in the system will have the same data at the same time.

When a system is consistent, it means that every read returns the most recent write for a given piece of data.

Strong Consistency

Strong consistency ensures that any read operation returns the **most recent write** for a given piece of data. This means that once a write is acknowledged, all subsequent reads will reflect that write.

How It Works

- **Atomic Writes**: Every write operation is atomic and immediately visible to all nodes.
- **Read-After-Write**: After a successful write, any subsequent read will return the updated value.
- **Synchronous Replication**: Data is replicated to all nodes before the write is considered complete.

Pros

- **Simplicity for Developers**: Developers do not need to handle stale reads or data conflicts.
- **Data Integrity**: Ensures the highest level of data integrity and reliability.
- **Predictable Behavior**: Guarantees that all nodes have the same data at the same time, which is critical for certain applications.

Cons

- **Latency**: Synchronous replication can lead to higher latencies as the system waits for all nodes to acknowledge the write.
- **Availability Trade-offs**: In a partitioned network, the system may become unavailable to maintain consistency (as per the CAP theorem).
- **Scalability**: Can be challenging to scale due to the overhead of ensuring all nodes are in sync.

Use Cases

- **Financial Systems**: Where accurate and up-to-date information is crucial.

- **Inventory Management**: Where real-time stock levels must be maintained to avoid overselling.
- **Booking Systems**: Such as airline reservations where double-booking must be prevented.

Eventual Consistency

Eventual consistency ensures that, given enough time, all nodes in the system will converge to the same value. However, there are no guarantees about when this convergence will occur.

This model is often used in distributed systems where high availability and partition tolerance are prioritized over immediate consistency.

How It Works

- **Asynchronous Replication**: Writes are propagated to other nodes asynchronously.
- **Temporary Inconsistency**: Reads may return stale data until all nodes are updated.
- **Conflict Resolution**: Mechanisms are needed to handle conflicts that arise from concurrent writes.

Pros

- **Low Latency**: Asynchronous replication can lead to lower latencies for write operations.
- **High Availability**: The system remains available even during network partitions.
- **Scalability**: Easier to scale as nodes can operate independently and eventually converge.

Cons

- **Complexity for Developers**: Developers must handle stale reads and data conflicts.
- **Data Integrity**: There is a risk of temporary inconsistencies and potential data loss.
- **Unpredictable Behavior**: Applications must be designed to tolerate eventual consistency.

Use Cases

- **Social Media**: Where real-time updates are less critical, and the system prioritizes availability and partition tolerance.

- **Content Delivery Networks (CDNs):** Where low latency is crucial, and eventual consistency is acceptable.
- **E-commerce:** For non-critical data like product recommendations and user reviews.

Feature	Strong Consistency	Eventual Consistency
Guarantee	Immediate consistency	Consistency over time
Latency	High (due to synchronous replication)	Low (due to asynchronous replication)
Availability	Lower during network partitions	Higher during network partitions
Scalability	Challenging due to synchronization overhead	Easier due to independent node operations
Developer Experience	Lower (simpler data model)	Higher (conflict resolution needed)
Use Cases	Financial systems, inventory management, booking systems	Social media, CDNs, non-critical e-commerce data