Consistency

Consistency is a fundamental concept in system design that ensures all clients see the same data at the same time.
It is one of the three properties described in the CAP theorem, which states that distributed systems can only provide two of the following three guarantees simultaneously:

Consistency: Every read receives the most recent write or an error.
Availability: Every request receives a response, without guaranteeing that it contains the most recent write.
Partition Tolerance: The system continues to operate despite network partitions.

Types of Consistency

1. Strong Consistency:
Guarantees that all the reads will reflect the most recent write, no matter where the data is requested from.
Example: A Banking system ensuring that an account balance update is immediately visible to all the users.
Stock trading platforms: Ensures trades are processed in the correct sequence.

Implementation Technique:
Distributed systems use quorum based protocols or consensus algorithm like Paxos or Raft to achieve strong consistency
Example: A write must be acknowledged by the majority(quorum) of replicas before it is considered commited.

Trade-offs:
High latency due to synchronization across replicas.
Reduced availability during network partitions.

2. Eventual Consistency:
Ensures that if no new updates are made, all replicas will eventually return the last updated value.
Example:
DNS Systems: Updates may take time to propagate, but eventually, all servers reflect the correct state.
NoSQL databases(Cassandra, DynamoDB): Often prioritize high availability and allow temporary inconsistency.

Implementation Techniques:
Conflict resolution mechanisms(e.g. Last write wins or vector clocks)
Background synchronization processes to reconcile replicas.

Trade-offs:
Allows temporary inconsistency(e.g. stale reads)
High availability and low latency

3. Causal Consistency:
Ensures that causally related updates are seen by all clients in the same order, but unrelated updates may be seen in different orders.
Example: A social media platform ensuring that comments appear after the corresponding post.

Implementation Techniques:
Track causal relationships using vector clocks or logical timestamps

Trade-offs
More complex to implement than eventual consistency.
May not guarantee strong consistency.

4. Read-your-writes Consistency:
Guarantees that a user always sees the results of their own updates.
Example: A blogging platform showing an author their updated article after submission.
User dashboards: A user sees the latest changes they made to their profile, even if replicas are still syncing.

Implementation Techniques:
Client-side caching or session consistency mechanisms

Trade-offs:
Does not guarantee that other users will see the same updates immediately.

5. Monotonic Read consistency:
Guarantees that if a user reads a value, any subsequent reads will reflect the same or more recent values.
Example: Email applications ensuring users never see older email states after checking a newer one.

Implementation Techniques:
Ensure that requests from a client are routed to the same replica(sticky sessions).

Trade-offs:
Requires careful routing but is easier to implement than strong consistency.

Read-World Example: Distributed Database

Imagine a ride-hailing application (e.g. Uber).

Scenario 1: Strong consistency
* When a driver accepts a ride, that ride should immediately be marked as "not available" for other drivers.
* Ensuring strong consistency avoids scenario where multiple drivers are assigned the same ride.
* Trade-off: Increased latency because the system needs to synchronize the latest state across all nodes before responding.

Scenario 2: Evetual Consistency
* When a passenger's trip is completed, their trip history may take a few seconds to update on all servers.
* This allows for higher availability and after responses during the trip but sacrifices real-time consistency.
* Example: The trip completion status might be instantly available to the passenger, but take a few moments to appear in the driver's dashboard.

When to choose consistency over availability

Banking Transactions: Strong consistency is critical to ensure that balances are updated in real-time and correctly reflect transactions.
E-commerce inventory: Strong consistency ensures that customers cannot over-purchase limited stock items.
Social Media: Eventual consistency is often sufficient because slight delays in reflecting comments, likes or shares do not critically impact user experience.

Key Discussion Points in Interviews

Trade-off: Discuss latency, fault tolerance and availability trade-offs when designing for consistency.
Use cases: Explain when to use strong vs eventual consistency based on requirements.
Implementation: Suggest techniques like quorum reads/writes, consensus algorithms: Paxos or Raft) or caching strategies.

Techniques for Implementing Consistency

1. Replication Protocols:
Master-slave Replication: Writes go to a single master node, and reads are served from replicas.
Leaderless Replication: Each replica can accept writes, and conflict resolution happens later.

2. Quorum-Based Systems:
Reads/Writes Quorums: Ensure consistency by requiring a subset of replicas to confirm an operation.
Formula: R+W>N, where R is the number of replicas involved in read, W is number of write and N is total number of replicas.

3. Consensus Algorithms:
Protocols like Paxos and Raft ensure strong consistency by agreeing on a single value among distributed nodes.

4. Conflict Resolution:
Last Write wins: Overwrites data with latest timestamp.
Application Logic: Allows custom conflit resolution based on business requirements.

Replication Protocols in Distrubuted Systems

Replication is the key strategy in distributed system to ensure high availaiblity, fault tolerance and scalability.
Replication protocols dictate how data is copied and kept consistent across multiple nodes in a system.

Replication protocols can be broadly classified into two main types:
1. Synchronous Replication.
2. Asynchronous Replication.

Within these categories, various strategies are employed based on system design requirements.

1. Synchronous Replication
In synchronous replication, a write operation is only considered successful when the data has been replicated to all required nodes. This ensures strong consistency but comes with increased latency.

How it works:
* A client writes data to primary node.
* The primary node sends the write operation to all replicas.
* The system waits for an acknowledgement from all replicas before confirming the write to the client.

Advantages:
* Guarantees data consistency across all replicas.
* Ensures no data loss if the primary node fails after acknowledgement.

Disadvantages:
* High latency because the system waits for replication acknowledgements.
* Reduced availability during network partitions(as writes fail if some replicas are unavailable).

Use Cases:
* Banking Systems: Ensures account balances and transactions are always consistent.
* Stock Trading Systems: Guarantees that orders are processed correctly and consistently.

2. Asynchronous Replication
In asynchronous replication, the primary node sends write requests to replicas but does not wait for acknowledgements.
The write operation is considered successful as soon as the primary node processes it.

How it works:
* The client writes data to the primary node.
* The primary node processes the write and immediately responds to the client.
* Replication to secondary nodes happens in the background.

Advantages:
* Low latency for write operations
* High availability since writes can continue even if some replicas are unavailable.

Disadvantages:
* Risk of data inconsistency between replicas during failures.
* Potential for data loss if the primary node fails before replication is completed.

Use cases:
* Content Delivery Networks (CDNs): Allows fast propagation of updates to edge servers.
* Social Media Platforms: Permits eventual consistency for likes, comments or shares.

Strategies for Data Replication

1. Leader-Based Replication(Master-Slave)
How it works: A designated leader(Master) handles all write requests and propagates changes to follower(slaves). Reads can be served by the leader or followers.

Advantages:
* Centralized control simplifies conflict resolution.
* Strong consistency is easier to achieve.

Disadvantages:
* The leader is a single point of failure(unless leader election is implemented).
* Potential bottlenecks at the leader.
Example: Traditional RDBMS(MySQL with master slave setup).

2. Leaderless Replication
How it works:
Any node can accept writes, and data is replicated to other nodes. Conflict resolution happens later, often based on timestamps.

Advantages:
* No single point of failure; highly available.
* Suitable for systems requiring high availability over strict consistency.

Disadvantages:
* Requires complex conflict resolution.
* May result in inconsistency reads temporarily.

Examples: DynamoDB, Cassandra.

3. Quorum-Based Replication:
How it works: A write is successful when a quorum(majority) of nodes acknowledge it.
Reads can also be performed with a quorum of nodes to ensure consistency.
Formula: R+W>N.

Advantages:
Balances consistency and availability..
Fault tolerant during network partitions.
Disadvantages:
Increased latency due to quorum calculations.

Example: Apache cassandra, Riak.

4. Chain Replication
How it works: Nodes are organized in a chain, with writes processed by the head of the chain and reads served by the tail. Updates propagate sequentially along the chain.
Advantages:
Guarantee strong consistency
Efficient for sequential data processing.

Disadvantages:
Latency increases with chain length.
A single failure in the chain disrupts the system.
Example: used in some distributed file system.

5. Multi-Leader Replication
How it works: Multiple nodes act as leader and accept writes. Changes are synchronized between leaders async.

Advantages:
* High availability and fault tolerance
* Supports geographically distributed writes.
Disadvantages:
* Conflict resolution is complex when multiple leaders handle conflicting writes.
Example: Postgre SQL multi-master setup.

6. Peer-to-Peer(P2P) Replication
How it works: Every node is equal, and all nodes can accept writes and propagate changes to others.
Advantages:
No single Point of failure
High availability.
Disadvantages:
Complex synchronization and conflict resolution.
Example: Git(distributed version control).

Practical Considerations for Choosing a Replication Protocol.

1. Consistency Requirements:
Use sync replication or leader based replication if strong consistency is critical. (ex: Banking)
Use async replication or eventual consistency for system prioritizing availability (ex: Social media)

2. Write and Read Latency:
Low latency systems favor async replication or leaderless replication.

3. Failure Handling:
Systems requireing resilience to node failure benefit from leaderless replication or quroum- based replication.

4. Conflict Resolution:
User conflict-free replication strategies(ex: last write wins) for simple scenarios.
Employ custom conflict resolution logic for complex systems (ex: merge strategies in multi-leader replication).

5. Geographical Distribution:
Mutli-leader replication or asynchronous replication is suitable for distributed systems spanning multiple regions.

Quorum-Based Technique for Implementing Consistency.

The quorum-based replication technique is a widely used method in distributed systems to achieve consistency while balancing availability and partition tolerance, as described in the CAP theorem. It ensures that a majority(or quorum) of nodes agree on the state of the system for reads and writes to be considered valid.

Key Concepts

1. Replication Factor (N):
The total number of replicas(nodes) that store copies of the data.
2. Write Quorum (W):
The minimum number of nodes that must acknowledge a write operation for it to be considered successful.
3. Read Quorum (R):
The minimum number of nodes that must respond to a read request.
4. Quorum Rule:
To ensure consistency, the sum of R and W must be greater that N:
R+W>N

This ensures that there is at least one overlapping node between reads and writes, guaranteeing that a read always retrieve the most recent write.

How it Works

Write Operation:
The client sends a write request to the sytem.
The system forwards the write request to all N replicas.
The write is considered successful if at leaset W replicas acknowledge the operation.

Read Operations:
The client sends a read request to the system.
The system queries all N replicas.
The read is successful if responses from at least R replicas are obtained.
If there are conflicts, the system resolves them(by choosing the most recent write).

Conflict Resolution

When nodes are queried during a read operation, replicas might return conflicting data. The quorum-based system typically resolves conflicts using:
1. Last Write Wins: Use timestamps to determine the most recent write.
2. Versionn Vectors: Track causal relationship between updates.
3. Application specific Logic: Let the application decide how to merge conflicting data.

Advantages of Quorum-Based Replication

1. Flexibility:
Tunable parameters allow balancing between consistency, availability and latency.
2. Fault Tolerance:
The system can tolerate failure of N - min(R,W) nodes and still function.
3. Scalability:
Works well in distributed environments with a large number of nodes.

Dis-advantages of Quorum-Based Replication

1. Latency:
High latency if R or W requires responses from a majority of nodes.
2. Complexity:
Requires careful tuning to balance consistency and performance.
3. Inconsistencies During Failure:
Temporary inconsistencies can occur if a read quorum overlaps with a partial write quorum.