

Consistency, CAP Theorem & Idempotency Notes (Day 15–16)

I. Distributed Consistency Models

In a distributed system, data is replicated across multiple nodes. Consistency defines how and when these nodes agree on a single "truth."

1. Strong Consistency

After a write completes, any subsequent read will return that value. It feels like there is only one copy of the data, even if there are many.

- Trade-off: It requires a "consensus" (like Paxos or Raft), which increases latency because the system waits for all nodes to agree.

2. Eventual Consistency

The system guarantees that if no new updates are made to a data item, eventually all accesses to that item will return the last updated value.

- Trade-off: High availability and speed, but users might see "stale" (old) data for a few milliseconds or seconds.

Interview Talk Track:

"I choose the consistency model based on the business impact of stale data. For a banking system, strong consistency is non-negotiable; we cannot allow a user to spend money they just transferred out. However, for a YouTube comment count, eventual consistency is perfectly fine—if one user sees 100 likes and another sees 102, the world doesn't end, and the system stays fast."

II. The CAP Theorem

The CAP Theorem is a fundamental principle for distributed data stores. It states that you can only provide two out of three guarantees at any given time.

1. The Three Pillars

- Consistency (C): Every read receives the most recent write or an error.
- Availability (A): Every request receives a non-error response (without the guarantee that it contains the most recent write).

- Partition Tolerance (P): The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network between nodes.

2. The "Choice"

In the real world, network failures (partitions) will happen. Therefore, P is mandatory. The real choice is between CP and AP.

- CP (Consistency/Partition Tolerance): If the network fails, the system shuts down to avoid returning wrong data.
- AP (Availability/Partition Tolerance): If the network fails, nodes keep responding, even if they can't talk to each other (leading to inconsistent data).

Interview Talk Track:

"In any distributed system, Partition Tolerance (P) is a requirement because networks are unreliable. So the real design decision is: when a partition occurs, do we stop the system to maintain data integrity (CP), or do we keep serving users even if the data is slightly out of sync (AP)? For a checkout service, I'd lean toward CP; for a discovery feed, I'd choose AP."

III. Idempotency: The "Retry" Safety Net

An operation is idempotent if it can be performed multiple times without changing the result beyond the initial application.

1. Why it matters

In distributed systems, the network often fails *after* a server has processed a request but *before* the client gets the "Success" response. The client will naturally retry.

- Non-idempotent: \$N\$ retries = \$N\$ charges to the user's card. ❌
- Idempotent: \$N\$ retries = 1 charge to the user's card. ✅

2. Implementation: The Idempotency Key

The most common way to handle this is by using a unique Idempotency Key (usually a UUID) generated by the client.

- The server checks its database (e.g., Redis) for that key.
- If the key exists, the server returns the previous result without re-processing.
- If not, it processes the request and saves the result.

Interview Talk Track:

"I treat Idempotency as a core requirement for any write operation in a distributed environment. Since we cannot guarantee 'Exactly-once' delivery in networking, we must design our APIs to handle 'At-least-once' delivery gracefully. By using Idempotency Keys, we ensure that retries—whether from a buggy client or a network timeout—don't result in duplicate side effects like double payments."

Practice Interview Questions (Day 15–16)

1. "You are building a 'Live Cricket Score' app. Which CAP trade-off would you make?"

- Answer: AP (Availability + Partition Tolerance). Users want to see a score immediately. It's better to show a score that is 2 seconds old (available) than to show an error message because the servers are busy synchronizing the exact millisecond of the last ball (consistency).

2. "How do you achieve Strong Consistency in a system that is physically distributed across the globe?"

- Answer: You use a Synchronous Replication or a global consensus protocol like Spanner (Google). However, the trade-off is high latency, as light can only travel so fast between data centers in different countries.

3. "Is a DELETE request idempotent?"

- Answer: Yes. If you delete User ID #5, the first time it's deleted. The second time you try to delete it, it's still gone. The *result* (User #5 is gone) is the same, even if the HTTP status code changes from 204 No Content to 404 Not Found.

4. "What is PACELC theorem, and how does it extend CAP?"

- Answer: (Bonus point question!) PACELC says: P (if there is a Partition), choose between A (Availability) and C (Consistency); E (Else/otherwise), choose between L (Latency) and C (Consistency). It explains that even when the system is healthy, you still have to choose between being fast (L) or being perfectly consistent (C).

5. "If a POST request is not idempotent by default, how do you make it idempotent?"

- Answer: By using an Idempotency Key. The client generates a unique `request_id` and sends it in the header. The server ensures that it only processes a POST with that specific `request_id` once.

