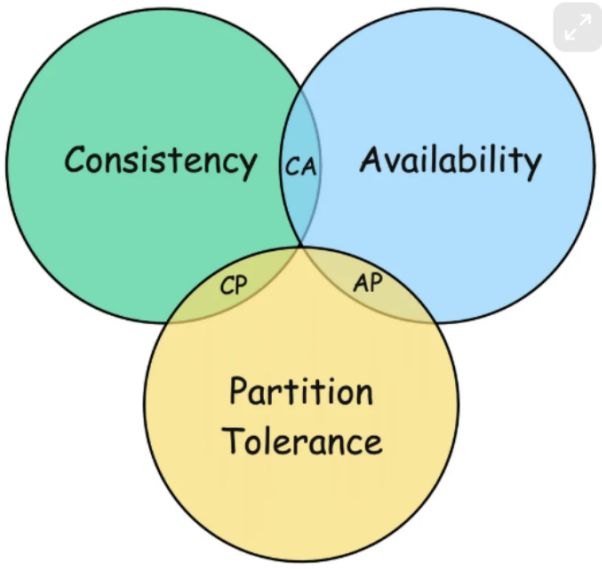
[CAP Theorem]



CAP stands for **Consistency**, **Availability**, and **Partition Tolerance**, and the theorem states that:

> **It is impossible for a distributed data store to simultaneously provide all three guarantees.**

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

In this article, we will explore the the 3 pillars of CAP theorem, trade-offs, and practical design strategies for building resilient and scalable distributed systems.

When Partition happens

⇓

choose between

* What to choose depends on type of system

E.g -> (1) Bank transaction -> Consistency preferred

(2) Seat Booking -> Consistency

(3) Social Media Likes -> Availability

Practical Applications of CAP

CP Systems (Consistency + Partition Tolerance):

- Prioritize data accuracy, even if the system becomes unavailable during failures.
- **Use Cases:** Banking systems, payment gateways.

AP Systems (Availability + Partition Tolerance):

- Prioritize availability and responsiveness, even if some data is stale or inconsistent.
- **Use Cases:** Social media, real-time messaging.

CA Systems (Consistency + Availability):

- Theoretically impossible in distributed systems with partitions because network failures will force a trade-off.

* In Distributed System, only two is possible

1. Eventual Consistency

For many systems, strict consistency isn't always necessary.

Eventual consistency can provide a good balance where updates are propagated to all nodes eventually, but not immediately.

> **Example:** Systems where immediate consistency is not critical, such as DNS and content delivery networks (CDNs).

2. Strong Consistency

A model ensuring that once a write is confirmed, any subsequent reads will return that value.

> **Example:** Systems requiring high data accuracy, like financial applications and inventory management.

3. Tunable Consistency

Tunable consistency allows systems to adjust their consistency levels based on specific needs, providing a balance between strong and eventual consistency.

Systems like Cassandra allow configuring the level of consistency on a per-query basis, providing flexibility.

> **Example:** Applications needing different consistency levels for different operations, such as e-commerce platforms where order processing requires strong consistency but product recommendations can tolerate eventual consistency.

4. Quorum-Based Approaches:

Quorum-based approaches use voting among a group of nodes to ensure a certain level of consistency and fault tolerance.

Beyond CAP: PACELC

While CAP is foundational, it doesn't cover all scenarios.

Daniel Abadi proposed the PACELC theorem as an extension by introducing **latency** and **consistency** as additional attributes of distributed systems.

- If there is a partition (P), the trade-off is between availability and consistency (A and C).
- Else (E), the trade-off is between **latency** (L) and **consistency** (C).

This theorem acknowledges that even when the system is running normally, there's a tradeoff between latency and consistency.

In conclusion, the CAP Theorem is a powerful tool for understanding the inherent trade-offs in distributed system design. It's not about choosing the "best" property, but rather about making informed decisions based on the specific needs of your application.

By carefully evaluating the CAP trade-offs, you can architect robust and resilient systems that deliver the right balance of consistency, availability, and partition tolerance.