**CAP Theorem**

**Distributed System:**

It’s a group of machines working in coordination which appear as a single coherent system to end user.

**Consistency (C)**

If any read is happening after a write, all nodes should return the latest value of write. (**Fresh** Data)

**Availability (A)**

All available nodes **must respond** in non-error format to all requests sent by user, without any guarantee that response will contain consistent (fresh) data.

**Partition Tolerance (P)**

If communication channel between any two nodes (which help in maintaining consistency) is **broken (partitioned**), then each node must be able to serve user’s request as an **independent** system.

CAP theorem states that, the above 3 properties are desirable for any distributed system. In any real-world system, having all 3 of them is impossible. We need to give up on any one of these properties.

Partition tolerance occurs due to network failure, over which we do not have full control. It can happen anytime anyhow. Hence, we need to build systems to handle partition tolerance.

Thus, we cannot give up partition tolerance.

Example

Suppose there are 2 systems well connected with each other.

B

A

If we perform a write (x = 2) on A.

A is updated, B is updated (A receives acknowledgement).

Middleware

Hence, system is consistent.

Both, A and B are available, If a read( ) is performed on A or B, both will send a response in a non-error state with latest (fresh) value. There is no guarantee of freshness (as per definition) but in this case, both return fresh value. (Consistency and Availability Working).

Suppose, the middleware breaks, I perform a write(x = 3) on A.   
I then do read( ) on A and B, both will return different values.  
In this case, A and B both behaved as independent systems even after **partition**.

Both were **available** and returned non-erroneous values.

But, they were **not consistent** as B returned old (stale) value.

Hence, this system followed only **AP** (Availability and Partition Tolerance) .

Example 2:

In same scenario, if I decide that my distributed system should strictly follow consistency.

As the middleware was broken and we performed a read( ) on B after a write on A (x = 3).

B decided, not to give any response, or throw erroneous response.

Since, consistency must be followed, it will never give stale values. Fresh values or nothing.

In this case, both systems behaved independently and fulfilled **partition tolerance**.

Both were **consistent**, but B was **not available** at all time.

Hence, the system followed **CP**.

But, it’s not true that we need to give up on Availability or Consistency 100%.

There is a degree up to which we can achieve all Consistency and Availability.

B

A

Middleware

**Availability to some extent example**

If we wish to follow consistency.

When middleware beaks, B will not respond to any read( ) operations. Hence will not be available.

But, it can take write operations. Hence, it can be **available to some extent**. It will take only write(), A will take both read() and write().

When middleware is joined and both systems are connected together, all writes which happed at different timestamps on A and B will be merged.

And again, both will start giving fresh values.

Hence instead of shutting down B completely, we allowed it to serve write() requests.

Consistency was also maintained during that time, as B did not respond to read().

C – a – P

**Consistency to some extent example**

Suppose, a system needs to guarantee high availability and consistency to some extent is fine.

In above example, when middleware breaks, we will allow both read() and write() operations on A and B.

When both systems were disconnected from each other, some write() operations happened on both of them.

System logged all write() operations. If user wanted read(), this system will allow user to read() data which was freshly written by latest write() on that node A or B.

It is unaware of any write that might have happened on other system during that time span. But it guarantees **high availability** hence allows **dirty read**, by giving whichever latest value present on that system at that time.

When systems are connected again, all logs of A and B will be merged and both will start giving fresh values.

This way of consistency is called **eventual consistency**.

By definition, Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

c – A – P

How to decide?

Depends on Use case:

Always ASK interviewer the use case.

Suppose it’s a **Banking Application**, we cannot allow eventual consistency as well.  
We need **100% consistency on nothing**. Hence many a times we see, banking applications are down. Same for Seat Booking, Ticket Booking System.

Suppose, it’s a feature on YouTube, FB or Instagram which shows **likes on your video**, here **eventual consistency** can be considered, as its fine if we see different numbers at any given time, eventually, it will result in one particular number after some time, but system needs to be **highly available,** loading of webpage is more important.

**Tweaking partition tolerance**

We can also have **backup network** between any two nodes so that, even if one connection fails, we can switch to other network and guarantee that all 3 properties are achieved.

But, implementing this will be **costly** hence, we must consider the infrastructure supported as well.

Always ASK interviewer what is expected.