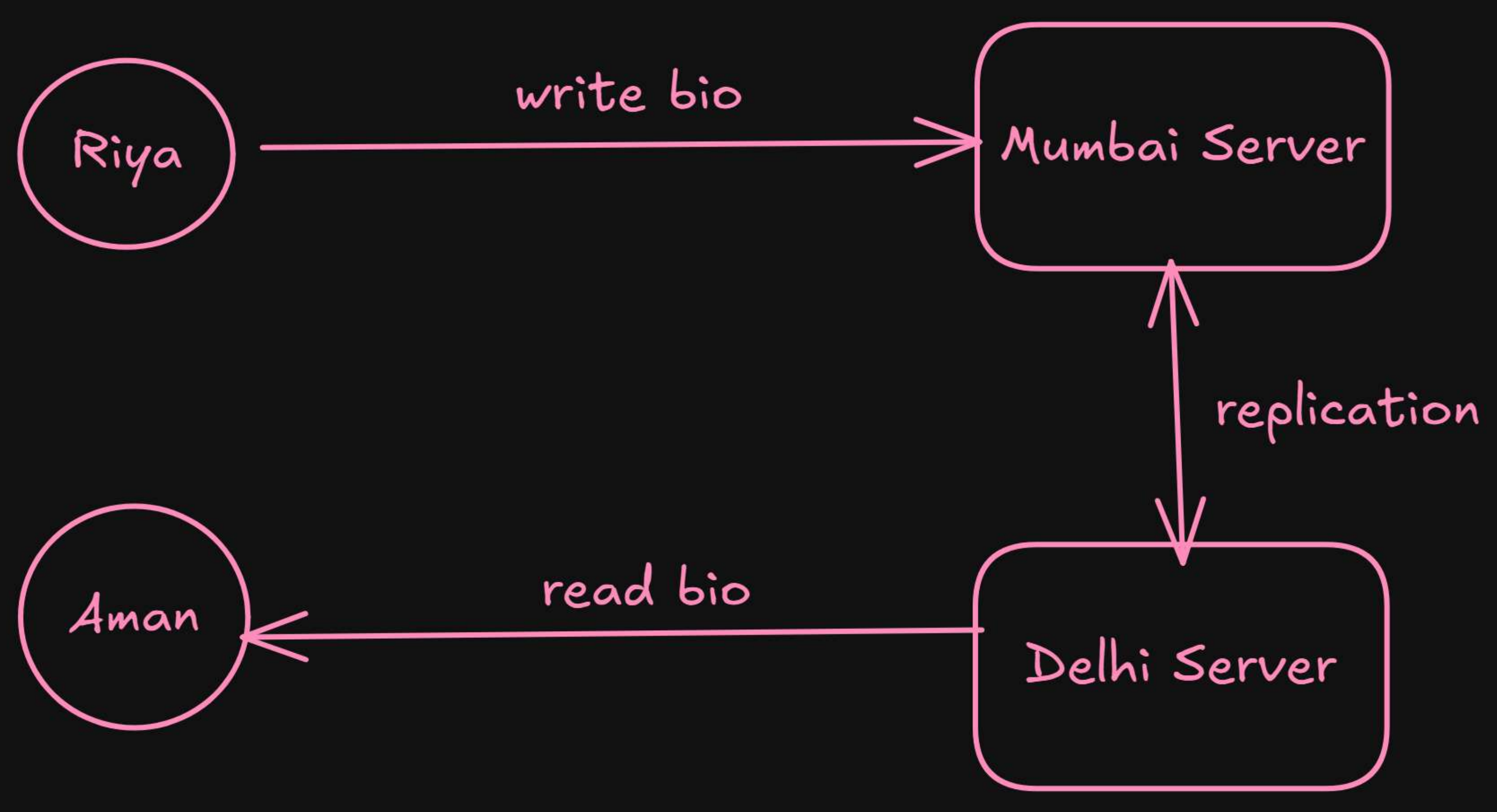
Think of Instagram running across India

Instagram has two major data centers:
-> Mumbai Data Center (serves West/South India)
-> Delhi Data Center (serves North/East India)

Now imagine this scenario:

Normal Situation
-> A user from Pune, Riya, updates her Instagram bio on the Mumbai server.
-> That update quickly gets replicated to the Delhi server.
-> A user in Delhi, Aman, opens Riya's profile and sees the updated bio.



Everything works perfectly.

Then a network issue hits
-> A routing problem, fiber-cut, or regional outage causes the connection between Mumbai ↔ Delhi to fail.
-> Now the two Instagram servers can't sync data.

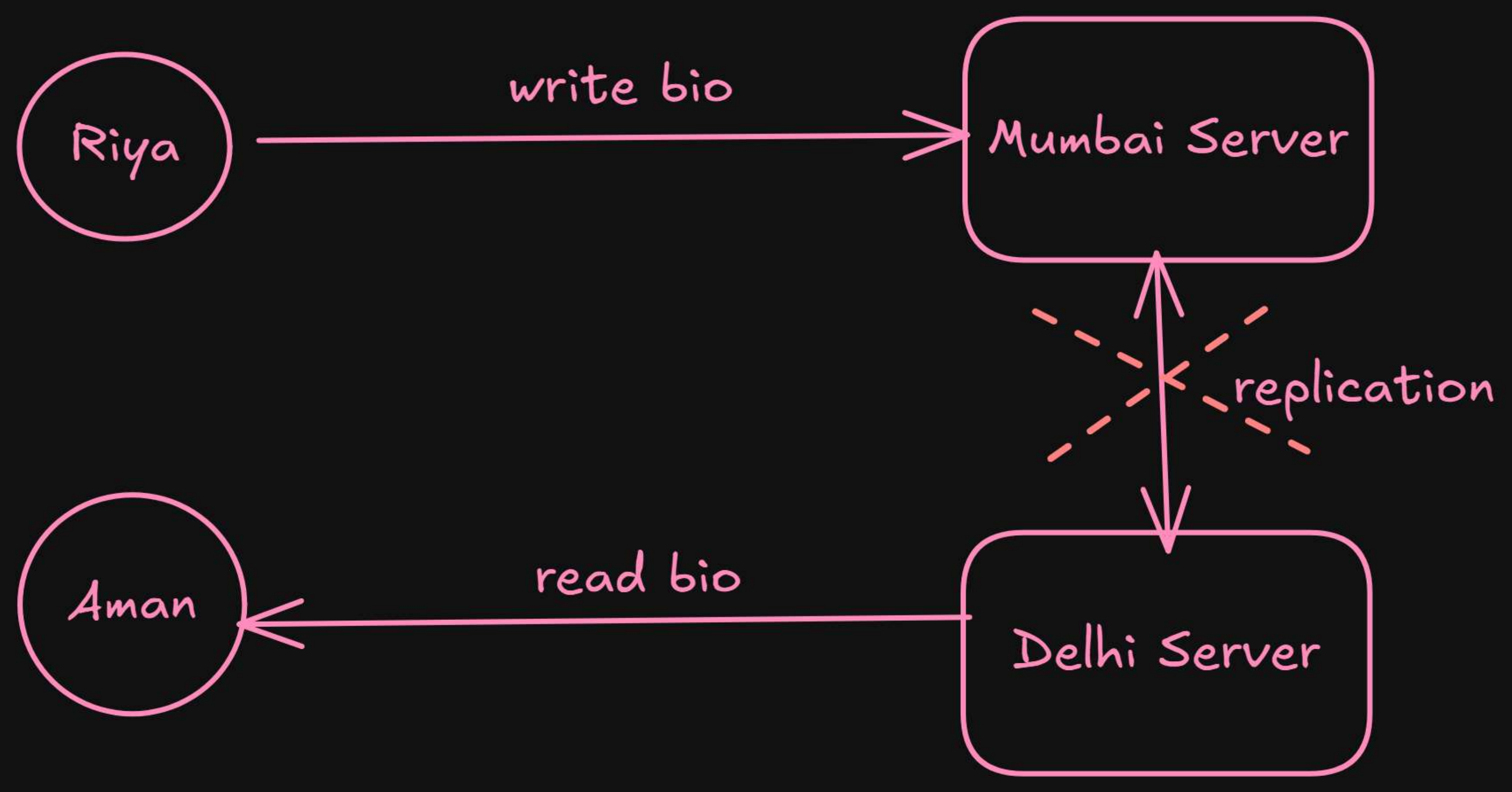
This is where things get tricky.

What happens when Aman opens Riya's profile now?

We must choose one of the two behaviors:

Option A — Consistency (Correct but unavailable)

Don't show Riya's profile because the Delhi server can't confirm whether the new bio from Mumbai is up to date.
So Aman gets an error or a blank profile.



Option B — Availability (Available but maybe outdated)

Delhi server still shows the old bio it already has.
Aman sees Riya's older information, but at least the profile loads.

What should Instagram do?

For a social media app like Instagram, the answer is obvious:

It's better to show an old bio than show an error.

A slightly outdated profile is harmless.
A "Profile cannot be loaded" message is frustrating.

CAP theorem in real life

Network problems will happen, so partition tolerance is mandatory.
That leaves one important decision:

During a network partition, should the system favor:

Consistency → show error if uncertain

Availability → show old data but keep working

For Instagram profiles,
availability is the better choice.

What is CAP Theorem?

In a distributed system, when a network partition occurs,
you must choose between Consistency and Availability.

You cannot guarantee all three: Consistency,
Availability, and Partition Tolerance.

In a distributed system, network issues will definitely happen.
So the system must handle broken connections.
That means CAP is really asking one simple thing:

If the network breaks, what do you want more — correct data or a working service?

The CAP theorem explains a fundamental limitation of distributed systems.
It says that when your data is spread across multiple machines,
the system can guarantee only two of the following three characteristics at any moment:

1. Consistency

Every part of the system shows the exact same data.
If one machine accepts a write, then any read from any other
machine should immediately return that new value.

2. Availability

Each healthy machine must always reply to a request.
However, the response may not always contain the latest update.

3. Partition Tolerance

The system must keep running even if the network between
machines is disrupted, messages are lost, or some nodes get isolated.

When Availability Should Be the Priority

In many applications, having the system remain online is far
more important than having every piece of data updated instantly.

These systems work perfectly well even if different users
momentarily see different versions of the data.

This approach relies on eventual consistency, meaning the
data will sync and become correct shortly, even if there's a small delay.

Examples Where Availability Matters More

Social Platforms:

If someone changes their display picture, it's not a big issue
if another user sees the older picture for a short while.
The app should continue loading smoothly.

Streaming Services (like Netflix, JioHotstar):

When a show's description or details are updated, some users
might continue seeing the previous text for a little time.
This doesn't disrupt the watching experience.

Food Delivery Apps (like Zomato):

Imagine a restaurant updates its menu, prices,
or temporarily switches off delivery.

Even if some customers still see the older menu
or slightly outdated prices for a minute or two,
the app should continue loading normally.
Showing stale information for a short while is
far better than showing an error page or
failing to load the restaurant altogether.

When a System Must Prioritize Consistency

There are situations where showing outdated or
conflicting information can cause serious problems.

In such cases, the system must ensure that every user
sees the exact same, correct data —
even if that means rejecting some requests during a network issue.

BookMyShow Seat Booking

-> Suppose someone from Delhi reserves Seat A12 for a
movie on BookMyShow.
-> If the servers are unable to synchronize due to a
network partition, another user in Bengaluru might
still see A12 as vacant.
-> If both are allowed to book, two people arrive expecting
the same seat — an unacceptable situation.

This is why strict consistency is essential here.

Flipkart Inventory Control

Imagine Flipkart has only one unit left of a popular smartphone.
If the system can't sync inventory updates, multiple users might be shown that the item is still in stock.

This leads to overselling, cancellations, and a poor customer experience.
To avoid this, consistency must be maintained.

Banking Transactions

Consider someone transferring ₹5,000 from their savings account.

If the system shows outdated balances across different servers, one side might think the money is still available while another has already deducted it.

This could let a customer spend more than they truly have or result in mismatched ledgers.

Banks must always maintain accurate, synchronized data — consistency is non-negotiable here.

A simple way to decide is to ask:

> "Will it cause serious problems if users momentarily see outdated or mismatched information?"
>
> If the situation becomes harmful or unacceptable, then consistency must be enforced.
>
> If a short delay in accuracy doesn't really matter, then availability should take priority.

Consistency in CAP vs Consistency in ACID

1. Consistency in CAP (Distributed Systems Consistency)

Refers to how up-to-date and uniform the data is across multiple nodes in a distributed system.

-> Every node should return the same, latest value.
-> If one node gets a write, all other nodes must reflect that write before answering reads.
-> CAP consistency is basically linearizability or read-after-write correctness across replicas.

Think of CAP consistency like this:
"If I write something, will every server immediately show the same value?"

Instagram Example (CAP Consistency)

If Riya updates her bio on Mumbai server, but Delhi server still shows the old bio →
CAP Consistency is broken.

But the data (bio text) is still valid. No rule is violated.

2. Consistency in ACID (Database Consistency)

Refers to maintaining valid data according to rules, constraints, and schema.
This is internal to a single database transaction, not multiple servers.

ACID consistency ensures:

no foreign-key violations
no constraint violations
no invalid data
database remains in a "valid state"

Think of ACID consistency like this:
"Does the transaction keep the database logically correct?"

> Remember:
>
> ACID consistency = correctness of data
>
> CAP consistency = same data across distributed nodes
>
> They solve completely different problems.

Banking Example (ACID Consistency)

If a banking transaction makes an account balance negative when it shouldn't be allowed →
ACID Consistency is broken.

But this might happen in only one database, not across replicas.

Types of Consistency Models

1. Strong Consistency

-> You always read the latest, most up-to-date value immediately after a write.
-> The moment data is updated anywhere, every node reflects that update.
-> Behaves like a single, perfect machine.
-> Guarantees correctness but often adds latency.
Example:
If you change your Instagram bio, every user — anywhere — sees the updated bio instantly.

2. Bounded Staleness

-> Reads may return slightly old data, but only within a guaranteed limit.

-> The system promises that data will not be older than X seconds or Y versions.

-> You know exactly how stale the data might be.

Example:
A social app might guarantee that profile updates become
visible everywhere within 5 seconds max.

3. Read-Your-Write Consistency

-> The user who made the update will always see their latest change,
even if others see it later.
-> Prevents confusion for the writer.
-> Good for user-facing apps where people expect instant changes to reflect for themselves.
Example:
You update your profile picture — you will always see the new picture,
even if others still see the old one for a short while.

4. Eventual Consistency

-> There is no guarantee about how old the data may be at the moment — but:

-> All replicas will eventually become consistent, as long as no new writes occur.

-> Prioritizes availability and speed.

-> Most AP systems (Cassandra, DynamoDB default mode) use this model.

Example:
If you update your status, some friends may see the old one for
a while, but eventually everyone sees the same final value.

5. Causal Consistency

-> If one operation logically depends on another, every user must see them in that same order.
-> Order of related actions is preserved.
-> Unrelated operations may still appear in different orders.
Example:
On Instagram, you comment on a post:
"This café looks amazing!"

A friend sees your comment and replies to it:
"Yes! It's in Mumbai —you should try it."

Since the friend's reply only makes sense after your comment, every user must see:

-> Your comment

-> The friend's reply

If the reply appears before the original comment,
the conversation becomes confusing.
Causal consistency prevents this by ensuring the logical
order of related comments is maintained.

But likes or comments from other users can appear in different sequences.

How CAP Theorem Fits Into System Design Interviews

After you clarify what the system needs to do (the functional requirements),
the next step is understanding:

reliability expectations
response time targets
scaling goals
failure tolerance

While discussing these qualities, a key question naturally arises:
"If the network breaks between nodes, should the
system preserve accuracy or remain responsive?"

If the System Must Stay Consistent

Consistency-first systems ensure all components see the
same data before performing any operation, even if this slows things down.

Design techniques you might consider:

Coordinated Writes:

All updates must be confirmed by several machines before they are accepted.
This avoids conflicting values but increases the time needed for writes.

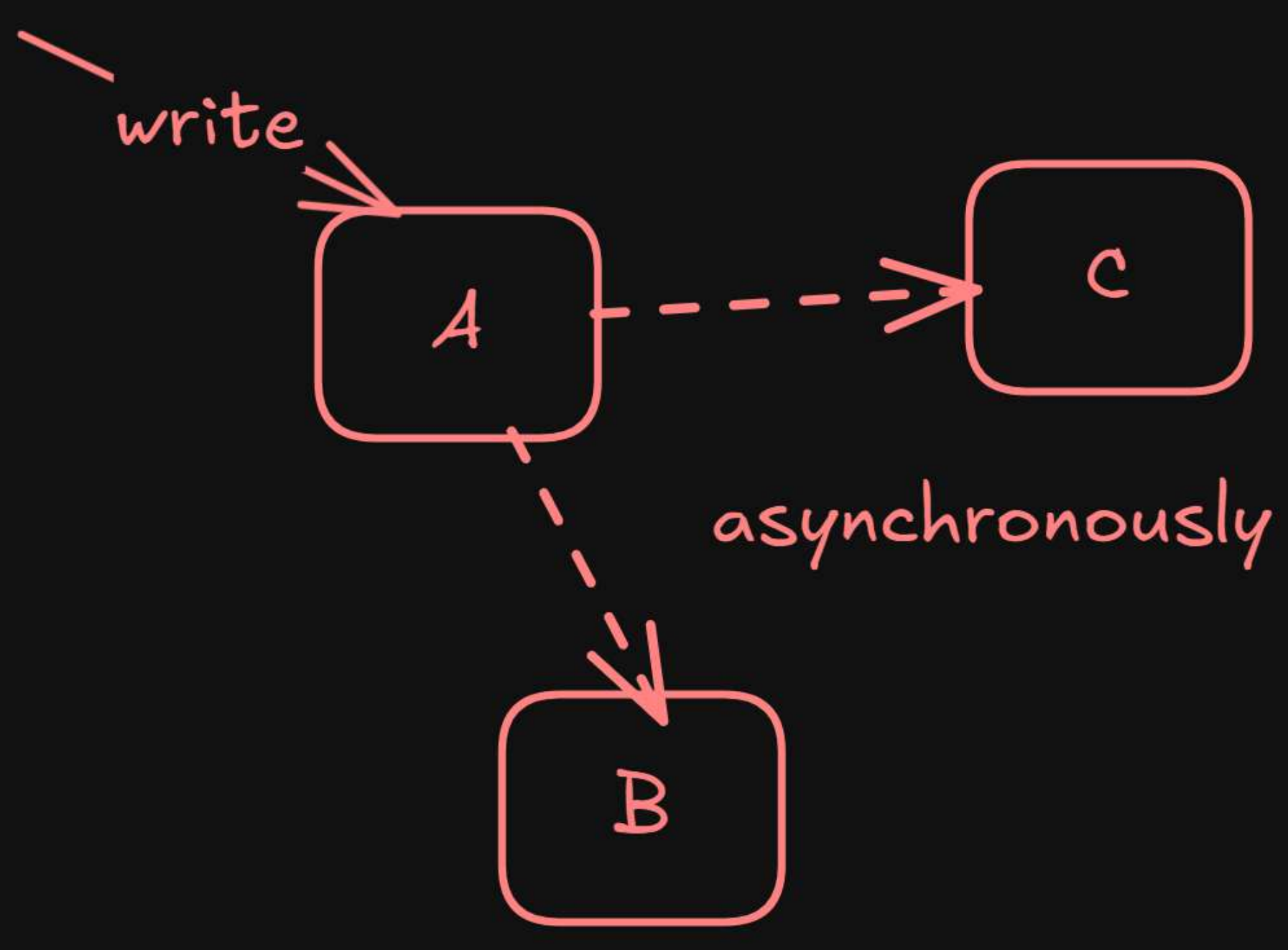
Centralized Datastore:

You might run everything through one main database instance
so that there's always one definitive truth.
Scalability becomes harder, but correctness becomes easier.

Tools typically used for strong consistency:

-> PostgreSQL / MySQL with a single primary
-> Google Spanner (global consistency via TrueTime)
-> DynamoDB in strongly consistent read mode
-> Zookeeper / Etcd for coordination

If the System Prioritizes Availability



For many applications, it's far more important that the system
keeps answering requests even if some parts of it briefly fall behind.

Architecture patterns often chosen:

Asynchronous Replication:

Writes go to a primary node and are pushed to replicas later.
This keeps the system fast but may momentarily show older data.

Distributed Read Nodes:
Multiple replicas handle read traffic independently.
Even if one replica hasn't caught up, the system
continues serving users without interruption.

Event Streams / CDC Pipelines:

Changes from the main database are captured and forwarded to caches,
analytics systems, and search indexes as background tasks.
The core system stays online even during spikes.

Common high-availability technologies:

-> Cassandra
-> DynamoDB in multi-AZ / multi-region setups
-> Redis clusters with replicas

Expert CAP Thinking

As products grow larger, the decision isn't simply "consistency vs availability."
Instead, you choose the right consistency model for each feature depending on how the business uses it.

Let's look at how this plays out in practical systems.

BookMyShow

A platform like BookMyShow doesn't treat all operations the same way.
Different features demand different consistency levels:

-> Seat Reservation → Needs Strong Consistency
-> Viewing Event Information → Availability is Fine

Instagram

-> Messaging (DMs) → Needs Consistency
-> Viewing a Profile → Availability Is More Important

PACELC Theorem

CAP theorem only talks about what happens during a network partition.
But what about the times when the network is perfectly fine?

That's where PACELC comes in.

PACELC says:

If there is a Partition (P), a system must choose Availability (A) or Consistency (C).
Else (E — when there is no partition), it must choose Latency (L) or Consistency (C).

So the full idea is:

**P → A or C

Else → L or C**

Why PACELC Exists

CAP focuses only on one rare event:
network partitions.

But most of the time, systems operate without partitions.

PACELC adds another dimension:

What trade-off do we make during normal operation?

This makes PACELC much more realistic for modern distributed systems.

Examples of PACELC in Real Systems

Example 1: Amazon DynamoDB

DynamoDB is generally PA/EL :

During a partition: prioritizes Availability over consistency (AP design).

Else (normal conditions): prioritizes low Latency over strict consistency.

This is why DynamoDB is super fast and highly available
but uses eventual consistency by default.

Example 2: Google Spanner

Spanner is a PC/EC system:

During a partition: chooses Consistency over availability (CP design).

Else: still chooses Consistency, even though it increases
latency because of TrueTime and global coordination.

This gives you global strong consistency but at the cost of higher latency.

> "CAP tells you what happens
> when the network fails.
>
> PACELC tells you what
> happens the rest of the time."

Example 3: Cassandra

Cassandra generally behaves like PA/EL:

Prioritizes availability during partitions.

Under normal operation, prefers low latency, using asynchronous replication.