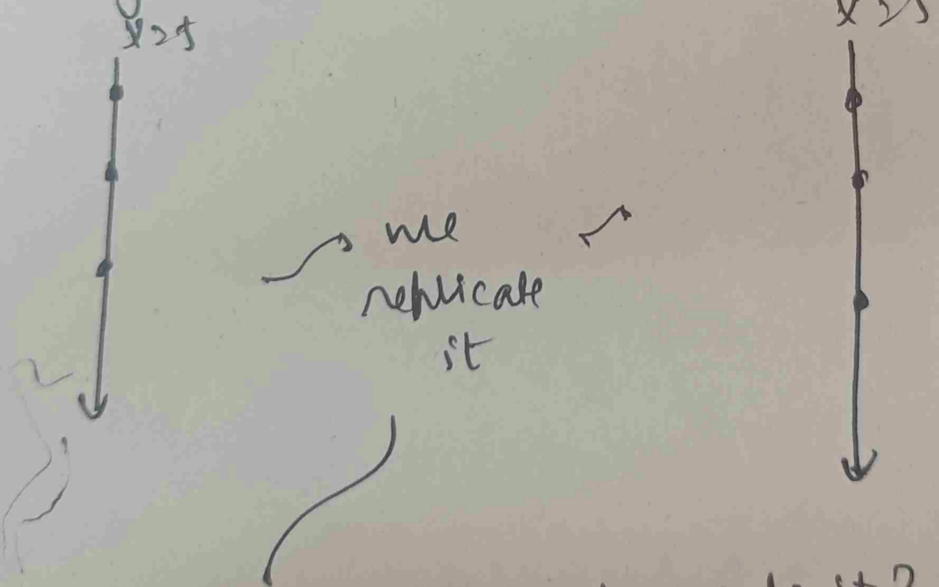
Lec-4 :- Replication, total order v/s determinism

So, last time we began to talk about replication

It's often the case that we mitigate loss of something by making copies.

- → So we mitigate message loss by making copies of messages
- → we mitigate loss of state by making copies of state.

Let's say we have a process with some events. and memo



and why would we do it?

→ If one these process crashes, then the fact that X=5 isn't lost forever.

→ That's one reason to do replication

mitigating data loss

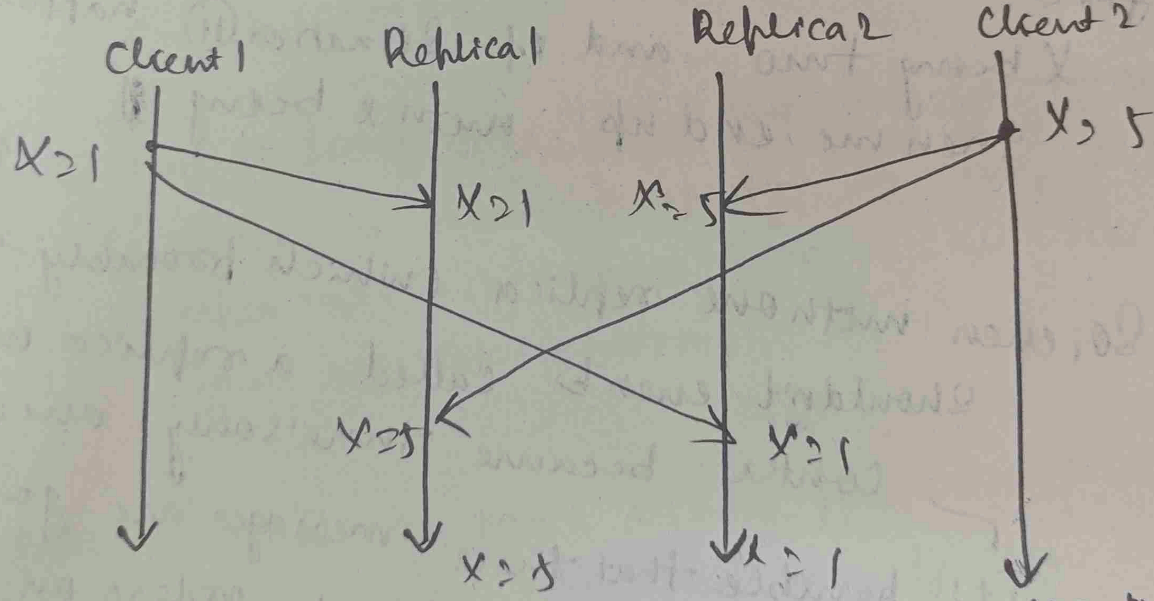
Reasons to do replication:

- mitigating data loss (fault tolerence)
- data locality (you want the copies to be close to clients, for faster response)
- Scalability (server more requests)

Downsides of replication:

→ Having to keep replicas consistent is huge problem

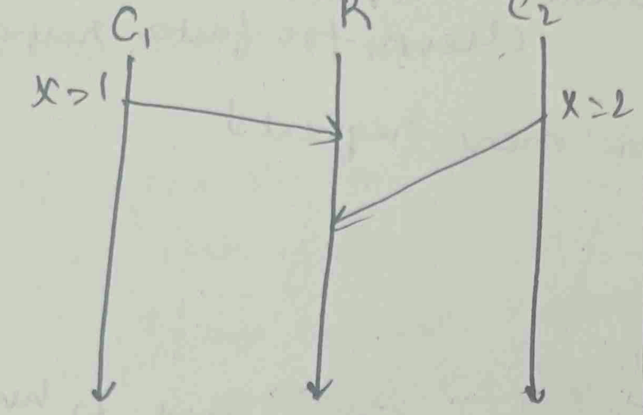
→ Its expensive



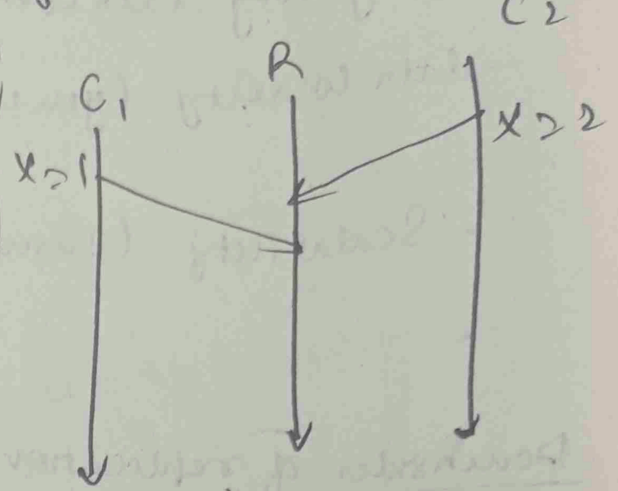
This illustrates one of the challenges with replication. Issue here is that different events happened in different order on two replicas

∴ Replicas end up being inconsistent

Let's just consider the case of one replica



Situation (I)



Situation (II)

If first situation happens, then we end with X being two and if Situation (II) happens then we end up with x being 1

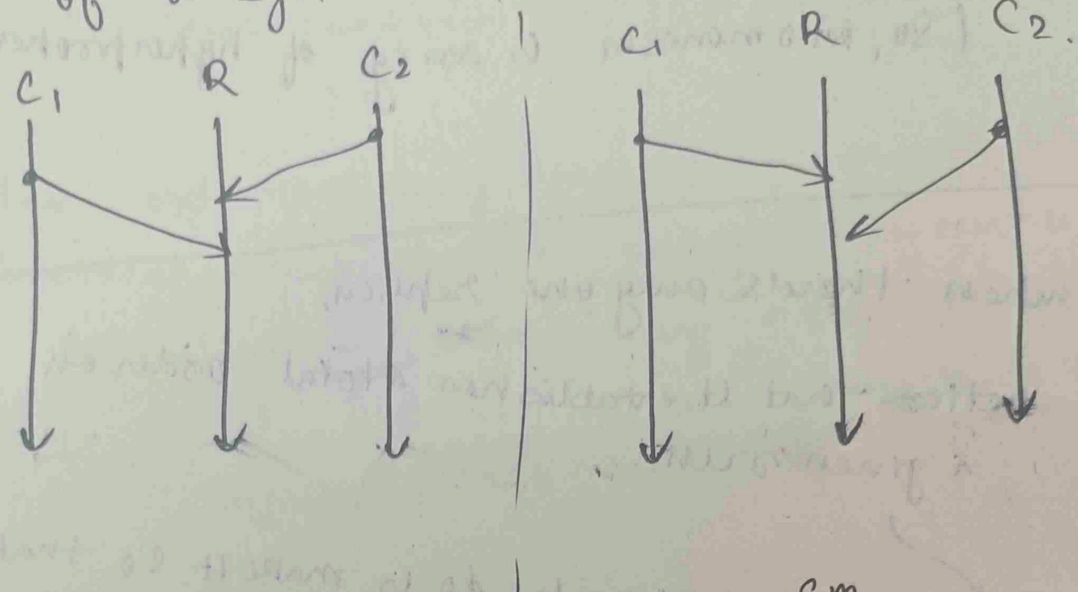
13:10.

So, even with one replica (which probably shouldn't even be called a replica in this context because there's only one of it) It's possible that these messages are going to arrive in different orders on different runs.

Situation (I) ya (II) mai koi galat ni keh sakta naa hi dono mai koi Total Order Delivery ko violate kar raha.

Determinism

- It is a property that relates multiple runs of a system to each other.



It is a violation of determinism,

It is not a violation of Total Order.

It is imp to point out, that even though total order delivery is a nice property to have, it's important to be aware that it still doesn't give you determinism.

Note: The totally order delivery property is a property that can only be true or not true of a single run.

But determinism is a property that relates multiple runs.

So, you can't look at a single run and say is the system deterministic. You have to be looking at multiple runs.

A property that relates multiple runs to each other is called a hyperproperty.

(So, determinism is an eg of hyperproperty)

So, when there's only one replica,

notice that it establishes a total order on a given run.

what we would like to do is make it so that there can be multiple replicas but this nice Total ordered delivery property.

In other words, we want clients to think that they are only dealing with one replica,

We want them to think that they are dealing with a system that does not have multiple replicas.

Strong consistency (Informal def):

→ A replicated storage system is strongly consistent if clients can't tell that it is replicated.

It turns out that every strongly consistent replication protocol, that we are going to discuss is going to work by establishing a total order on events, but they all are going to do in different ways.

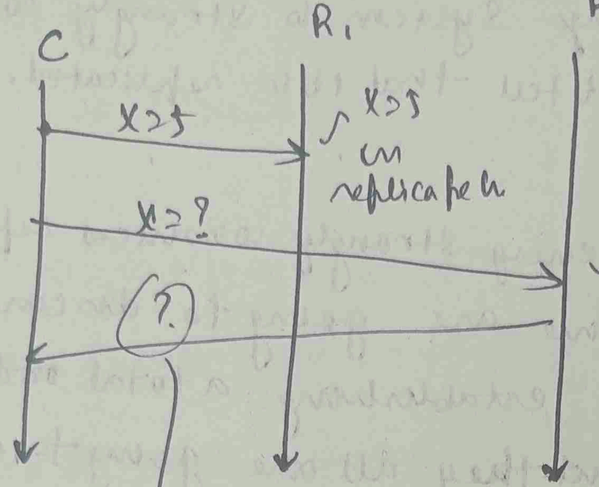
Before that, let's get into particular approaches for establishing this total order and implementing strong consistency.

Let's first talk about some of the ways where a client might be able to tell that data has been replicated.

In other ways, different ways in which replicas might disagree.

One of the worst cases that can happen is:

Case 1



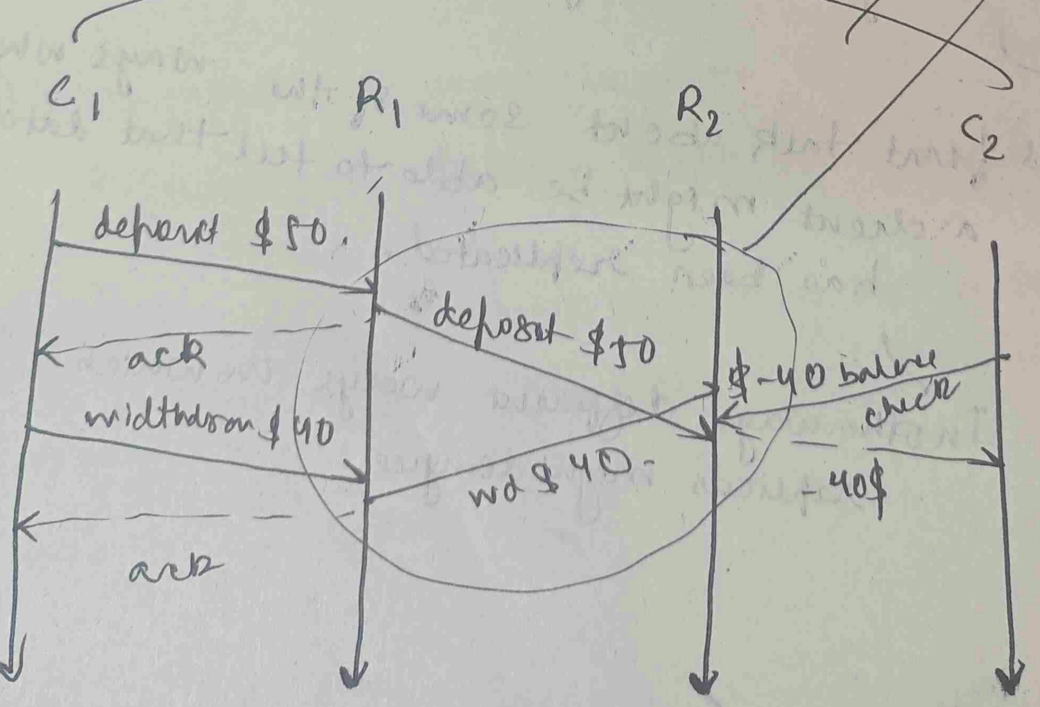
leken x=? ka req R2 pe gaya

fir to bhagwan jaane kya hua.

Read your writes anamoly.

Another thing that can go wrong is:-

Case II



Same person alag alag device se acc dekh raha

Case ① . mai to R₁ bothered hi ni tha
R₂ ko update
karne ko.

Case (ii) mai R₁ ne update kiya leken
galat order mai

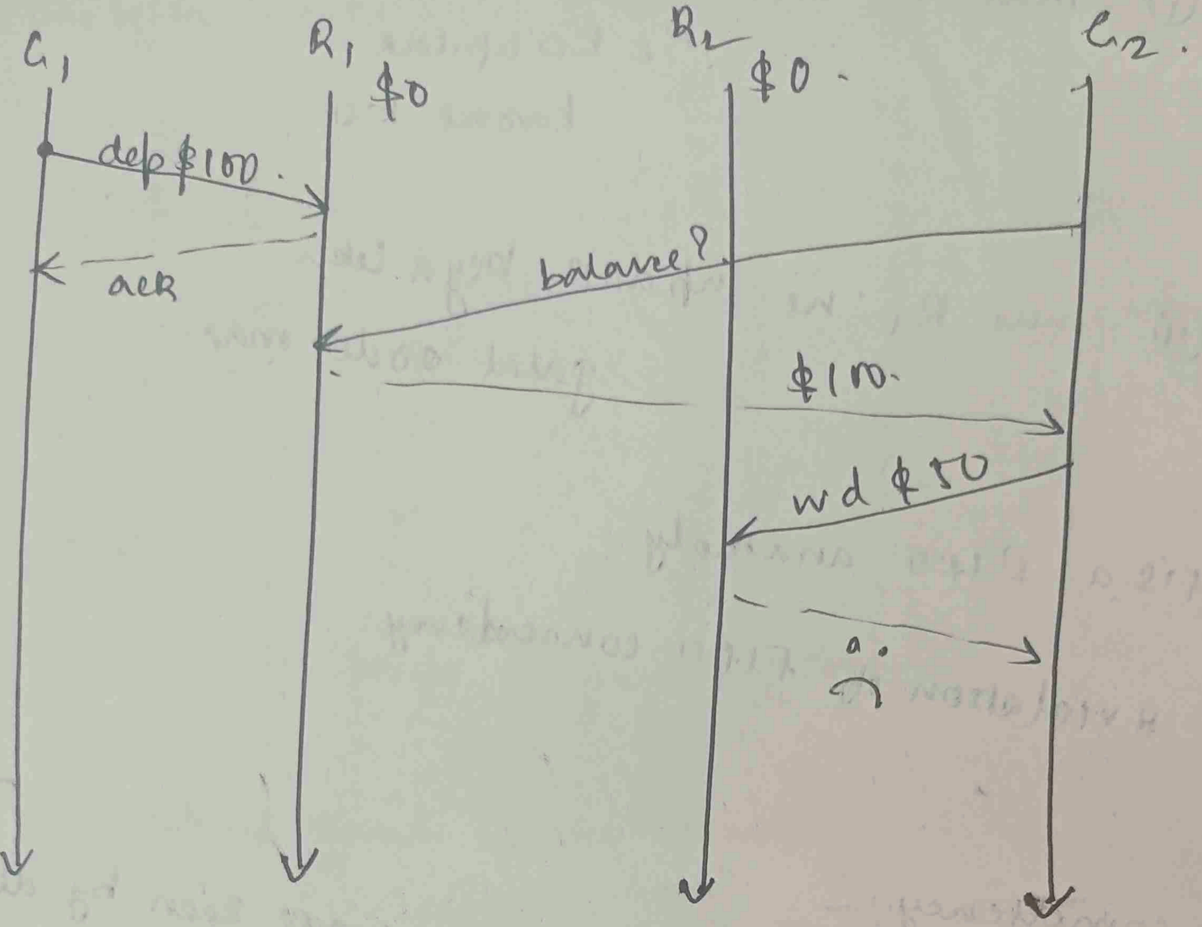
→ It is a fifo anamoly.

violation of FIFO consistency.

FIFO consistency :-

writes done by a single process are seen by all processes in the order they were issued.

Another bad thing that can happen : PTO



why did client ② think that they can do withdrawal?

→ Withdraw is being done because of the earlier deposit

and Client ② find out about this ~~bator~~ deposit because of "balance?".

So, If I follow chain of events

deposit happened before withdrawal.

So client 2 is not doing anything wrong.

The problem is that Replica ② sees the withdrawl but doesn't see the deposit which was in the causal history of ~~wd~~ withdraw.

So, because R② sees wd but doesn't see everything that was in the causal history of wd, this is called violation of causal consistency.

Causal consistency:

writes that are potentially related (i.e. related by →) must be seen by all processes in the same order.

Consistency Hierarchy

Read-your-writes consistency

↓

FIFO consistency

↓

Causal "

↓

Strong consistency

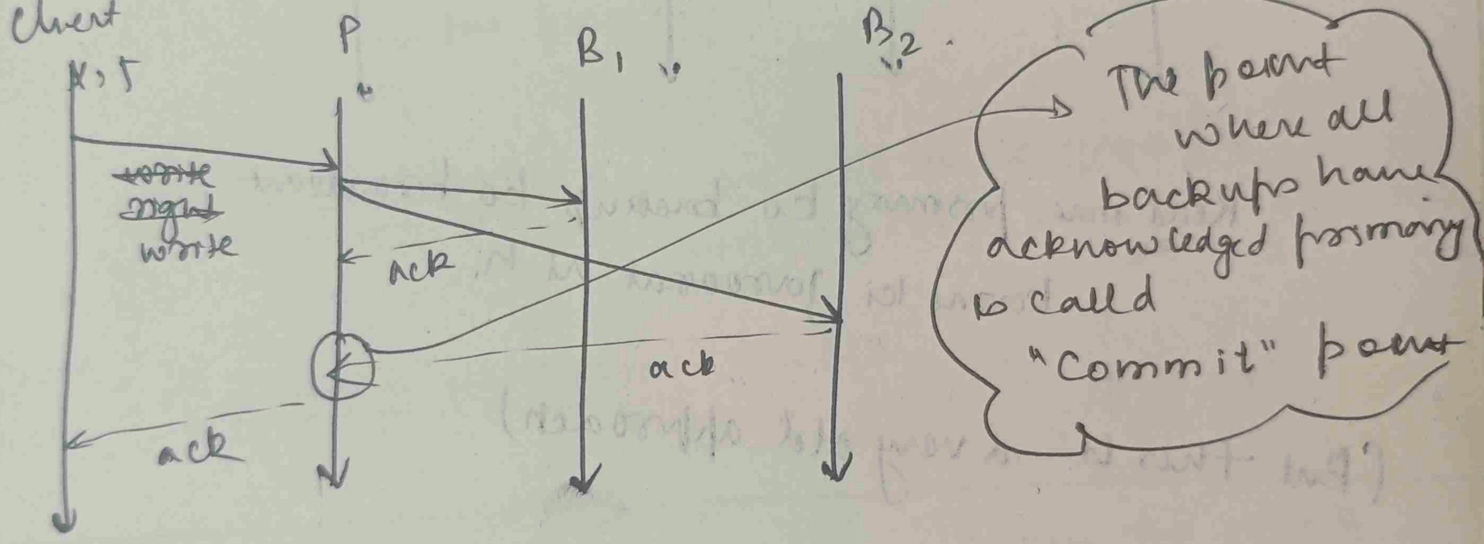
Bad ↓

Primary- backup replication

So, the idea is pretty, straight forward:

we pick a particular replica to be what's called the primary, and other replicas are backups,

let say → 1 primary
and
2 backup



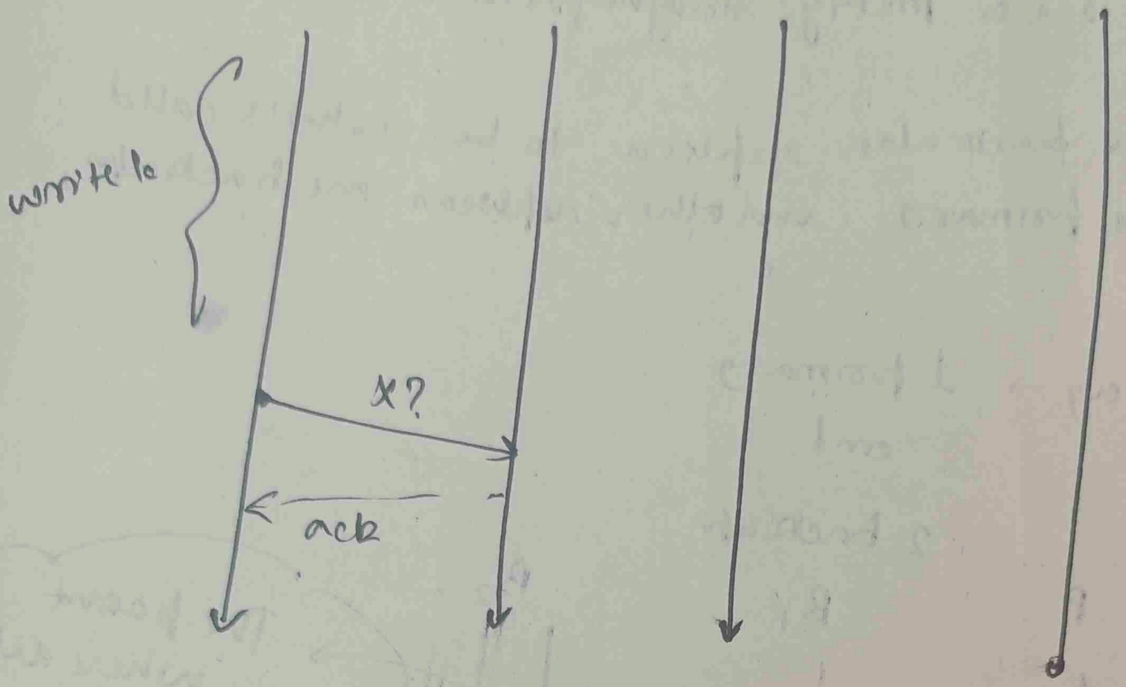
The point where all backups have acknowledged primary is called "commit" point

clients only interact with the primary.

So, when the primary get a ~~right~~ write from a client, it then broadcast that ~~right~~ to all (P) the backups, and all backups send an ack to the primary,

At the point when all of the backups have acknowledged the right to the primary, then the the primary can tell the client that the right succeeded.

What about read operation ?



Read mai primary ko backup ko broadcast karne ki jaroorat ni h.

(But this is a very old approach)

Drawback of this :-

— It's slow

Does Primary backup replication help with:

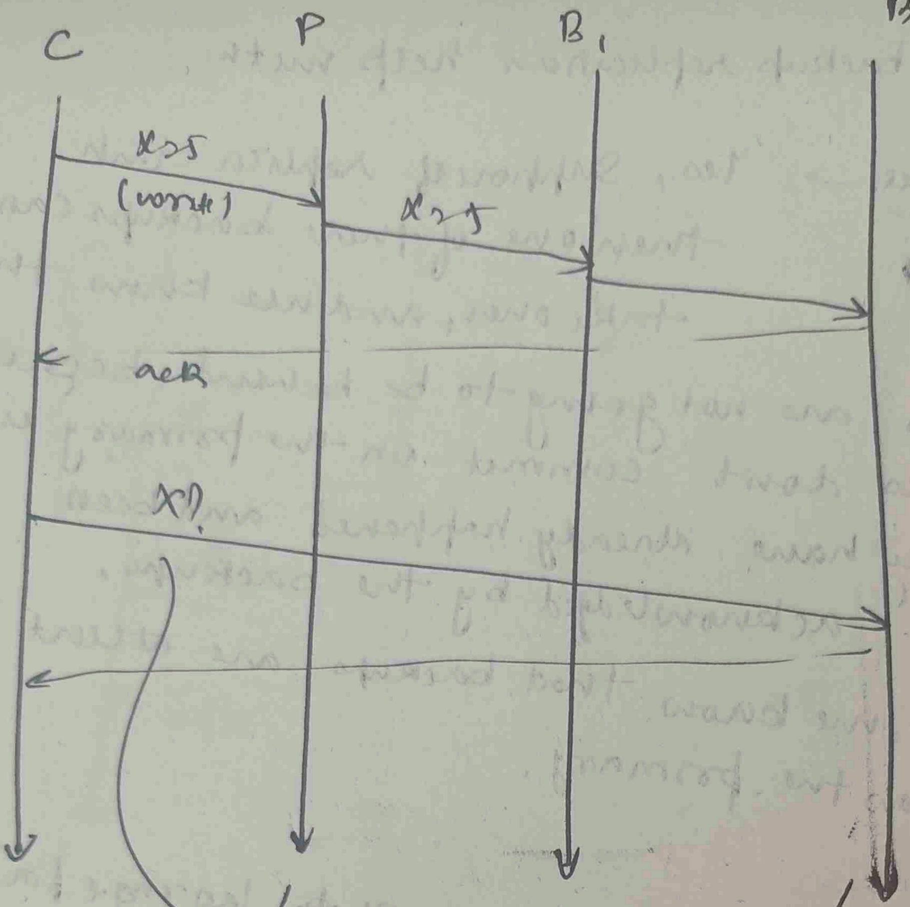
(a) <u>Fault Tolerence</u> → Yes, Suppose if replica crash then one of these backups can take over, and we know that backups are not going to be behind, because writes don't commit on the primary until they have already happened and been acknowledged by the backups.

So, we know that backups are atleast uptodate as the primary.

(b) <u>Data Localcity</u> :- No, ~~as read to~~ karoge hai write hi karh naa hi read.

(c) <u>Dividing up the work</u> → No

Could we do better?, Could we spread out the work any better?



#Chain replication.

Paper → Chain replication for Supporting High throughput and Availability

For read operation we choose one of the backup

Head

that only handles writes.

Tail.

that only handles read.

How are they Claiming High Throughput?

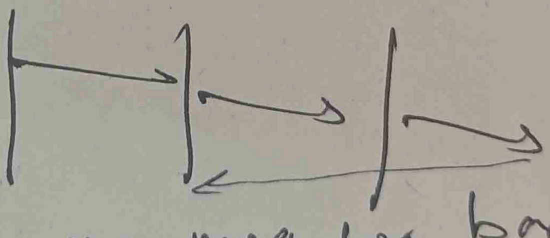
[Throughput :- no. of actions per unit of time.]

If we have a more or less equal mix of writes and reads, then in theory, chain replication is going to be a better choice than primary backup replication.

As we have one node handling writes and one handling reads then we can process more requests in a given amt of time.

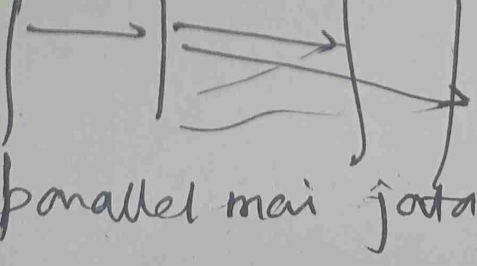
Downside of Chain Replication

→ Latency →



Yes msg for backup seq mai jata h.

Lekin primary ma



parallel mai jata tha, to faster tha.