L:10 ! Implementing Reliable delivery | idempotence.

Reliable delivery

→ Let P_1 be a process that sends a message 'm' to process P_2.
If neither P_1 nor P_2 crashes *, then P_2 eventually delivers m.

[* → and not all messages are lost]

Do we need that part?

→ well it depends,
If we are working under the crash model
~~or the~~ then the only kind of failures that can occur are processes crashing.

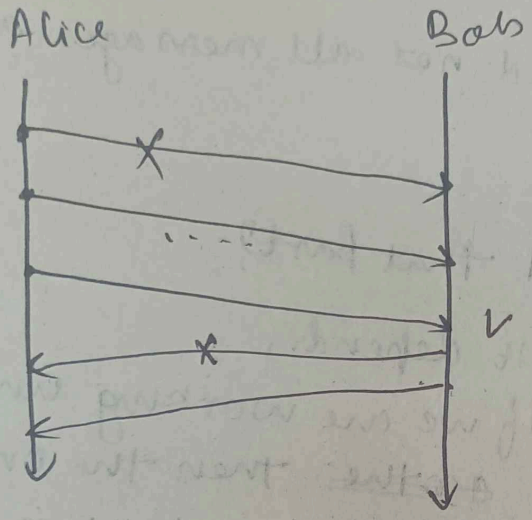
on the other hand
If we had omission model then we could have the crashing or we can have messages being lost, then we need * in definition too

[omission models more realistic than crash model]

How can we Implement Reliable delivery?

P.) Alice has to get a message to Bob how can she do so reliably?

→ Just keep sending messages over and over until you get an ACK

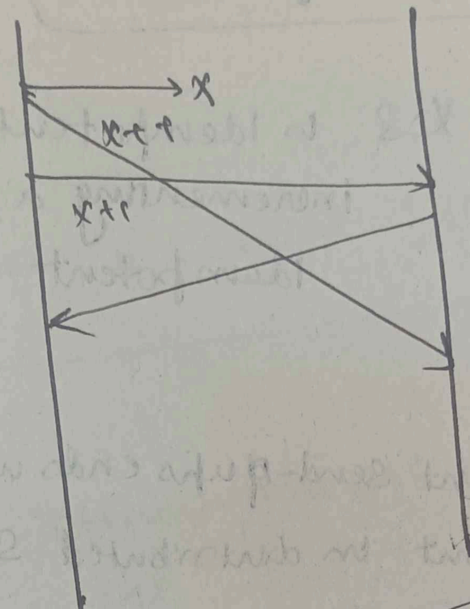


what is the problem with this

→ This doesn't solve the 2 general problems.

Alice can be sure that Bob got the message because she get the acknowledgement, but there's no certainty on Bob's side that Alice got acknowledgement.

→ Aur dousri baat ye ki
haan ek interval he
msg bhejne rahne
mei kuch dekkat h.



agar aisa
kuch might
B kya
kare waare
mai

Lekin wo alag case tha
ki msg agar X? type hota,
to multiple req jane se
chalta

So, the word for a message that it is okay
to receive for more than once or a
function that it's okay to run more than
once is "idempotent".

Idempotent

A function f is idempotent if

$$f(x) = f(f(x))$$

eg $X = 3$ is idempotent, but incrementing a variable is not idempotent

And Idempotent send-ups ends up being really important in distributed systems because generally speaking a good way to make sure something gets done is to try to do it a bunch of times until atleast one attempt succeeds.

And it's going to be useful if we can design systems in such a way that messages are idempotent

but this isn't going to be always possible. because sometimes we need operation. that aren't idempotent.

But all this thing has to do with reliable delivery?

→ In order to implement reliable delivery we typically need to deliver a message more than once sometimes.

So reliable delivery is called "atleast once delivery"

But how can you get "atmost once delivery"?

→ send a message
or just don't do anything

what about exactly-once delivery?

→

Reliable Broadcast

Unicast Communication

→ one sender and one receiver.

broadcast messages are one-to-all

(one sender, everyone receives)

eg Alice, Bob, Carol wala.

multicast messages one-to-many:

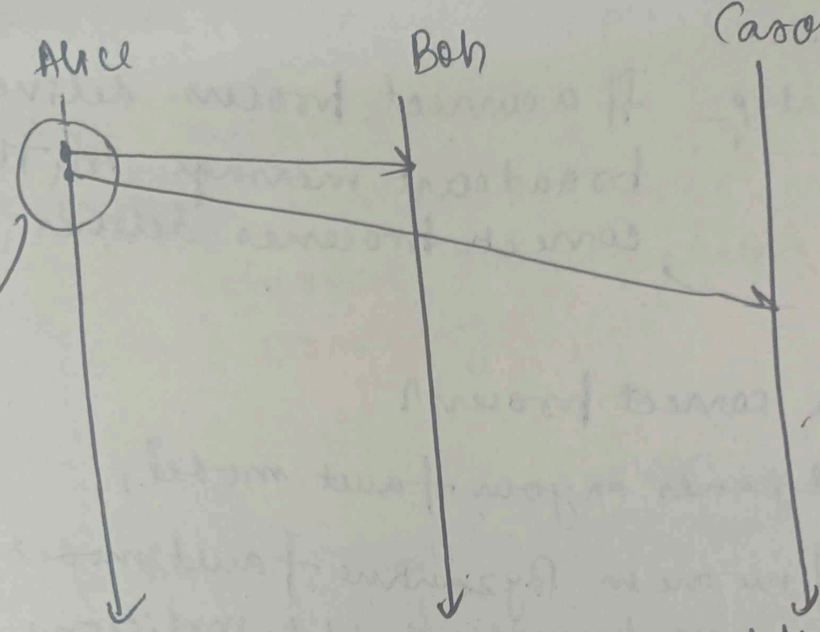
(one sender, many receives)

eg:- Total order wala eg.

So, we aren't going to talk about how to implement unicast messages in this class, we are going to assume that there is a built in primitive to begin with that lets you send a message from one sender to one receiver.

→ assuming that you had that built in primitive for implementing unicast messages how could you go about for implementing broadcast and multicast ??

→ You can just implement by sending a bunch of unicast messages.



I can send it to Bob and then almost at the same time I can send it to carol

So, under the hood these are actually seperate unicast messages.

but the way I want to think about them is as having the same send event and infer that's the way that we have been thinking of them all along

~~(phele saaf ek dot se dikhate, hum actually anal)~~

How do we define and implement reliable broadcast?

P.T.O

Reliable broadcast :- If a correct process delivers a broadcast message m, then all correct processes deliver m.

What is a correct process?

→ It depends on your fault model,

if we are in Byzantine fault model where processes could be malicious then you would certainly need to guarantee that they are not malicious

If we are in crash model, then a correct process would be something that didn't crash.

Reliable delivery (reduction):

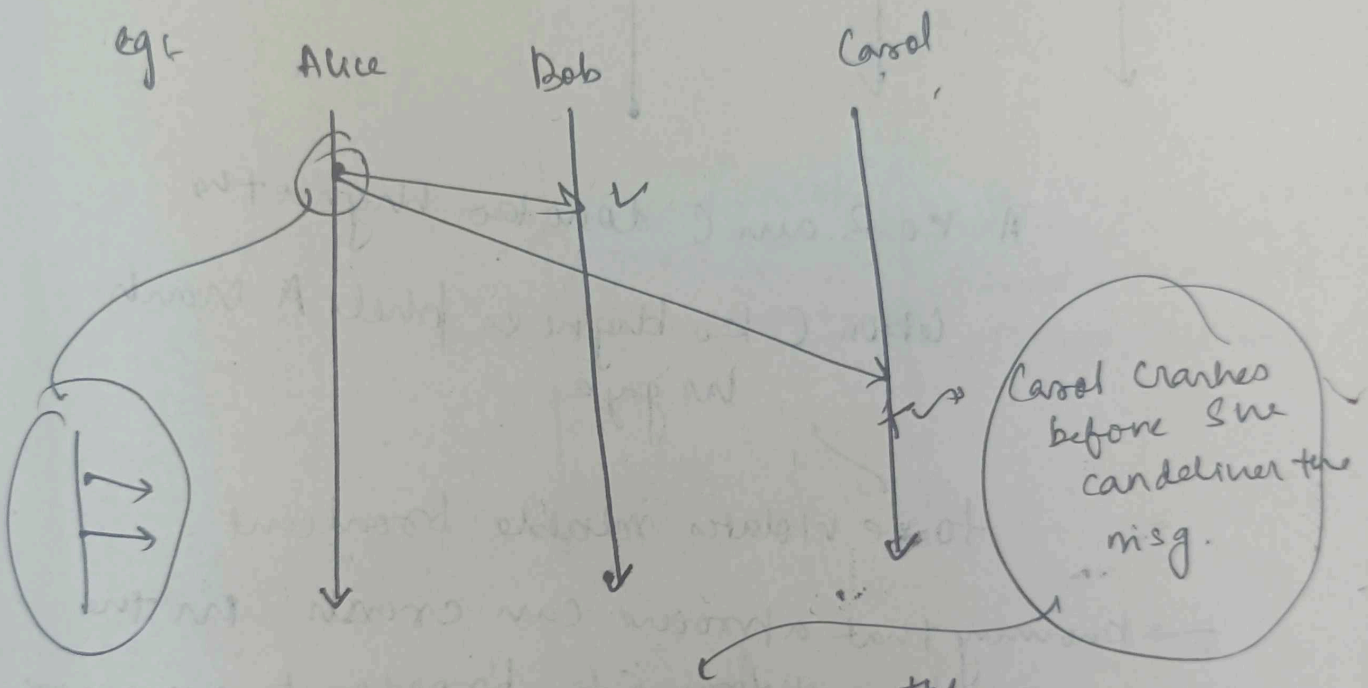
If a correct process P_1 sends a message m to correct process P_2, and not all messages are lost then

P_2 eventually delivers m.

How can you implement reliable broadcast?

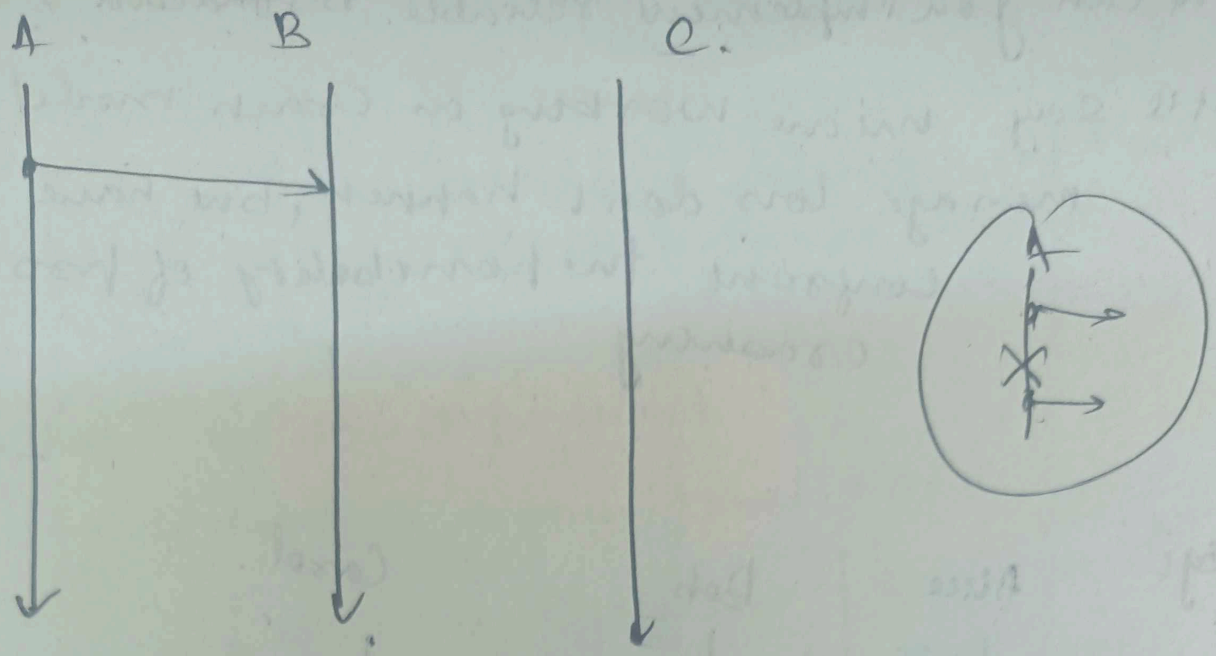
Let's say we are working on crash model and message loss don't happen, but have to confront the possibility of process crashing.

-31:00.



Does this violate the Reliable broadcast

→No, as Carol deliver karne se pehle crash ho gayi to no ab correct process ko nahi to no defn se hi bahar hi



A ko B aur C dono ko bhejna tha

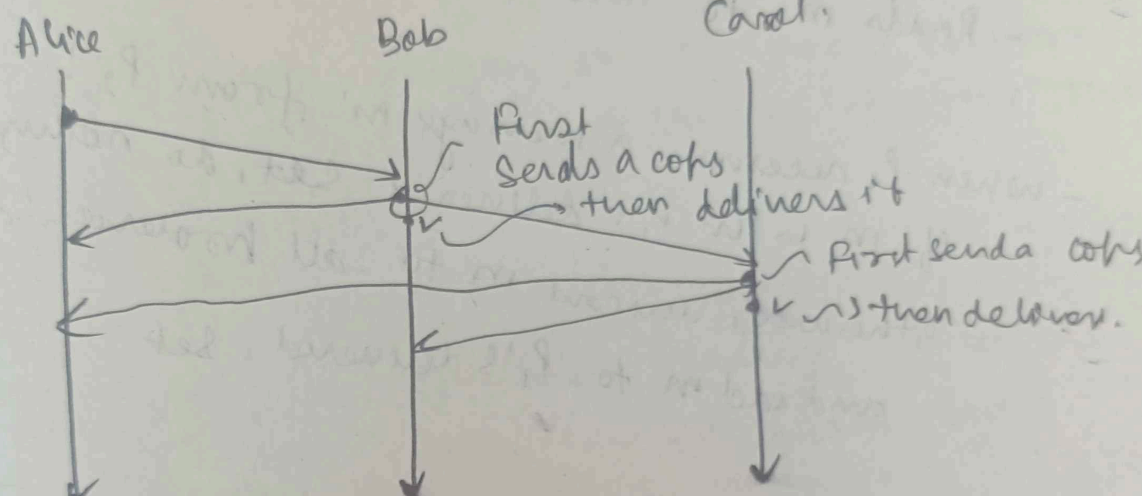
Lekin C ko bhejne se pehle A crash
ho gaya

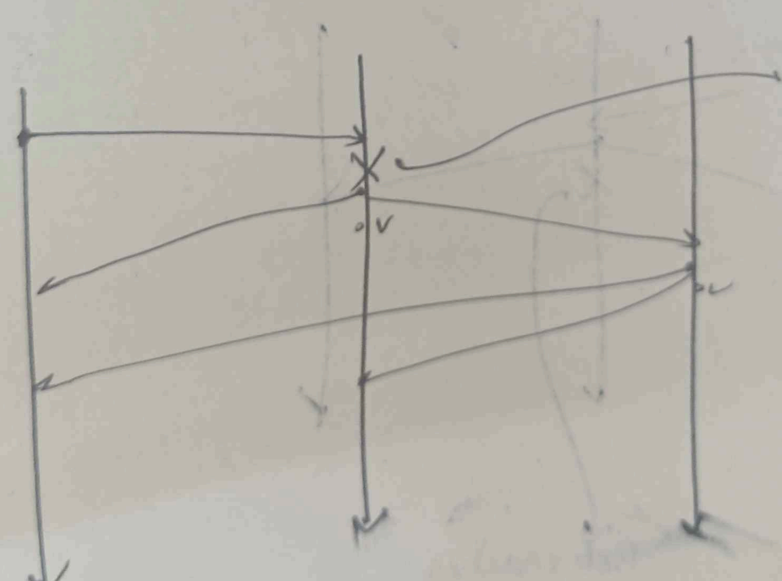
to y.e violates reliable broadcast.

→ knowing that a process can crash in the middle of trying to broadcast a messa
can you think of away we could
still have reliable broadcast

we introduce the following protocol.

→ when you receive a message, you don't deliver it yet, you pass it on to everyone else, and once you have done that you deliver it to yourself.





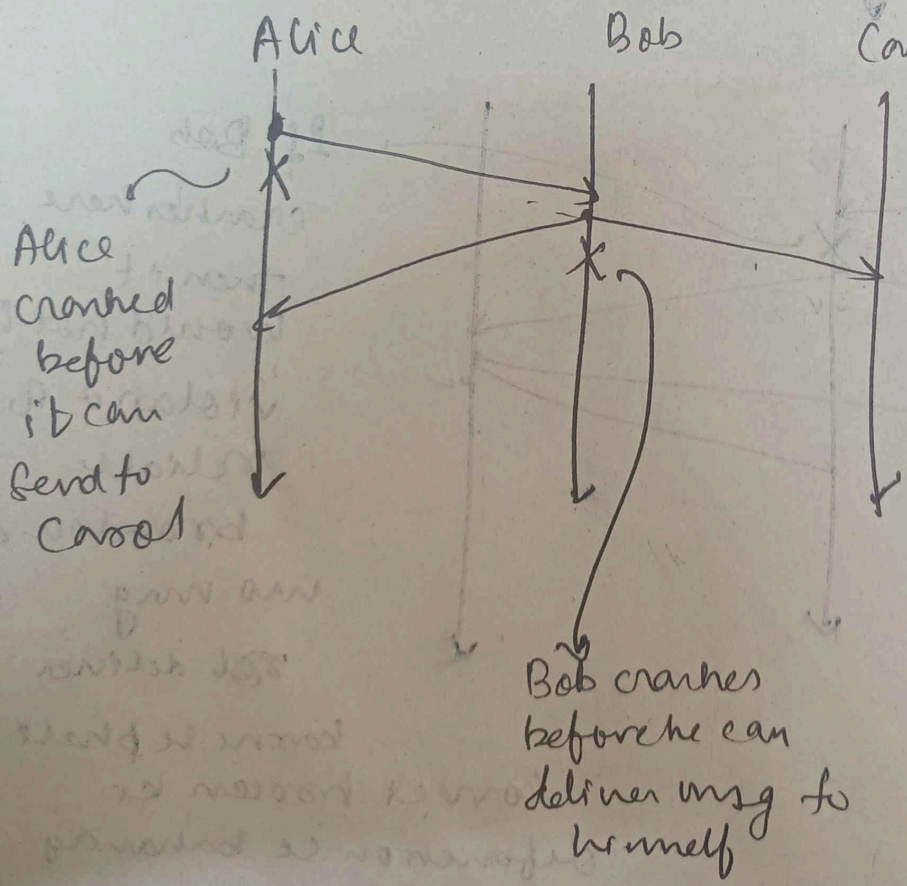
msg ko deliver karne se phale correct process ke definition se bahar aa gaya h

Reliable broadcast algorithm

11:00

- All processes keep a set of delivered messages in their local state.
- when P wants to broadcast a message m:-
  - P unicasts m to all processes. (also to itself)
  - P adds m to its delivered set.

- when P_i receives a message m' from P_j:
  - if m is in P_i's delivered set, do nothing.
  - otherwise, unicast m to all processes. (also to send)
    and add m to P_i's delivered set



So, now Carol is the only correct process.

and going back to the def"

"If a correct process delivers a broadcast msg m, then all correct processes delivers m"

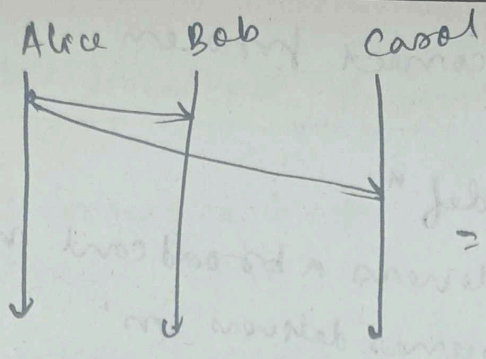
But now Carol is the only one correct process, So now whether or not Carol delivers the msg, it will still be satisfying this property.

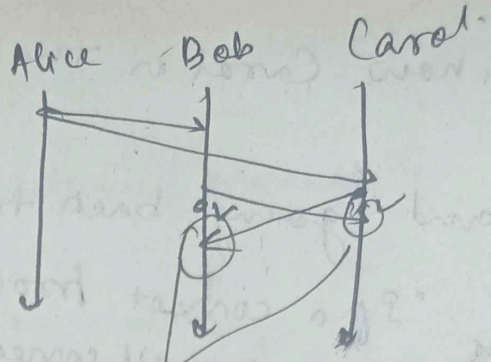
UT-10

→ You can have processes sending back to whatever they received from, but that is unnecessary.

So as an optimization we can leave out that message.

PTO





If meth the don't send back to sender optimization, we still have duplicate messages

So, Bob and Carol received the msg twice

So, Something that they can do is keep track of what messages they have delivered and then if (they receive one that they have already delivered then they don't have to deliver again.

"Fault tolerance" often involves making copies of things"

(take eg:- In case of reliable delivery we were able to tolerate message loss by just sending multiple copies of a msg until atleast one get through

and

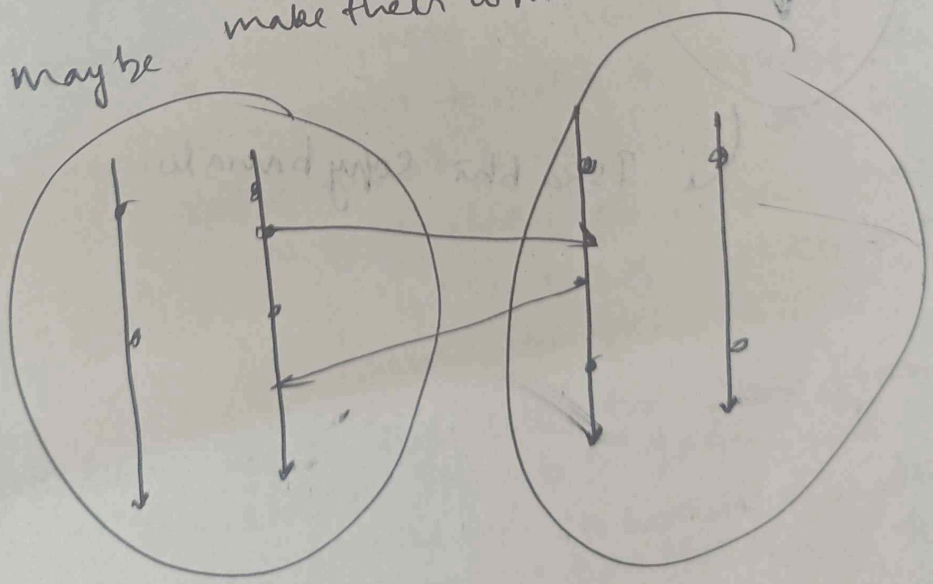
In case of reliable broadcast we were able to tolerate process crashing by making copies of received messages and forwarding them.

---

what other than messages do we have to be worried about maybe losing ?

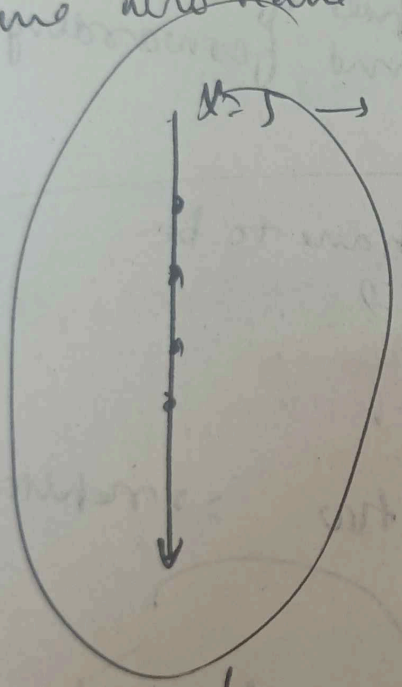
→ Internal events, keeping data,

(maybe make these copies too :) replication.



One thing that we talked about early in the course is that if we have a process and that process has some events on it, we can think of each of those events as being a state and whatever our state is at a certain point in time is determined by all the events that have happened until then

And we also have process local memory.

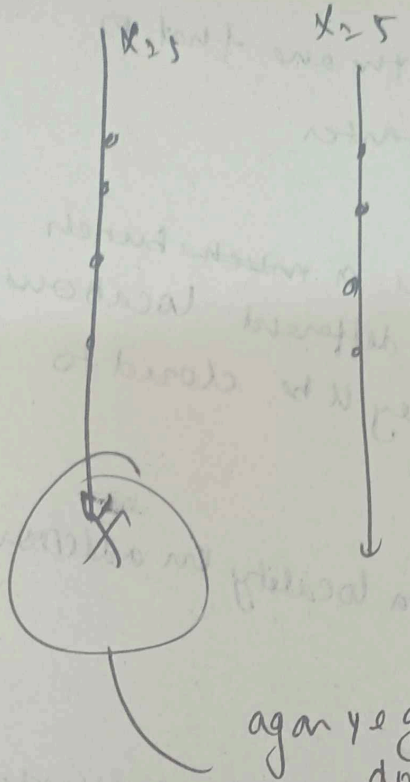


Ya to koi event ne x=5 kiya ho

ya

x=1 kiya ho for some events ne x++ kiya ho

Iska bhi copy banalo.



agar ye gaya bhi to kaise, hamare pas
dooura, ek a replica h.

Are there any other benefits of replication?

——→ Performance,

let is say I am here at home, (California)
and I want to make a request to this
Store which is running on a machine
in Australia, or I can make a
req to its replica which is just
down the road from me.

the data center just down the road from me
is probably going to be faster,
no promises as its internet everything is
asynchronous but it's likely that the

Physically closer machine is the one that is more likely to hear from faster.

So, one thing I can do is make a ~~mitch~~ bunch of replicas and put them in different locations around the world, so, they'll be closed to the clients that need them

⇓

So then you get good data locality in addition to fault tolerance.

→ Another reason to do replication is just dividing the work,