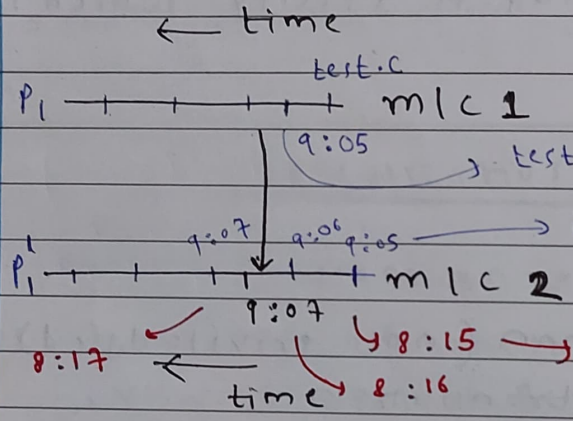
Lec 1 - AM (10th October, 2022)

Clock synchronization

* Solar clock used in ancient times

* Quartz clock = depend on manufacturer, drift may be there

* If we don't synchronize the clock
  ↳ synchronizing machines for meeting becomes difficult



let test.0 send to m/c 2, let test.0 sleep for 2 sec and after 2 sec, it will run
let m/c 2 start at 8:15, test.0 will reach m/c 2 at 8:17 (after 2 second sleep)

2 problems:

→ If clock of m/c 2 behind clock of m/c 1, then code has to wait for some time

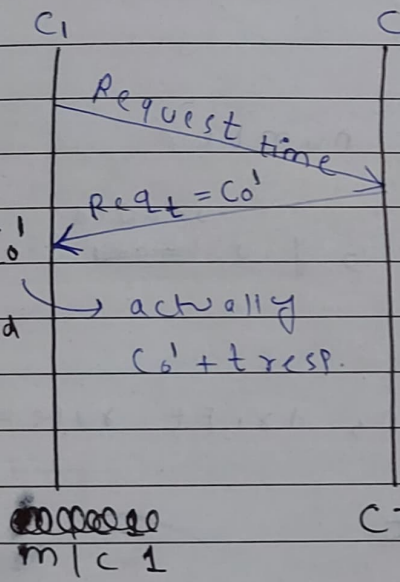
→ If clock of m/c 2 ahead clock of m/c 1, then error will be there and program not be started.

⇒ No problem occurs in case of centralized systems

time < C {slower second clock}
time > C {faster second clock}

(i) Centralized time server (CTS)
(ii) Berkely Master Clock
(iii) Distributed synchronization

} synchronization methods



* after receiving Co' from time server, it adjusts its own clock and blindly follow that

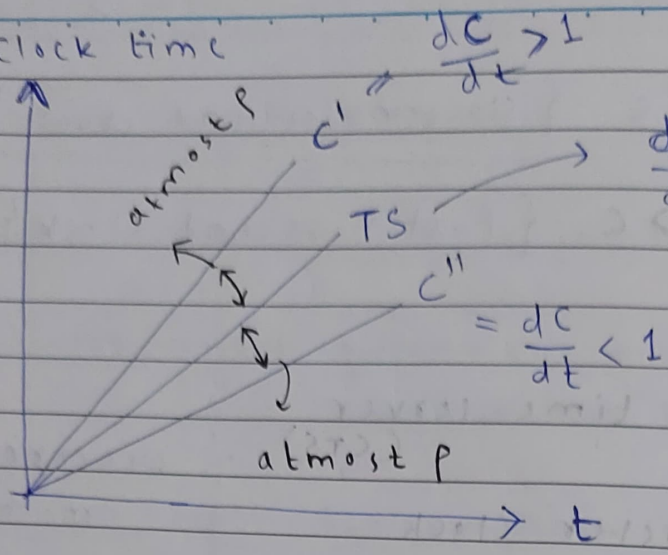
→ christian clock synchronization

Problem: over the N/w, it takes some time to reach m/c 1 from CTS, so in actual Co' will be equal to Co' + t response.

C1 = Co' + t resp.
= Co' + Δt { Δt time passes at CTS }

When to synchronize ⇒ ?

C'' = slower
C' = faster



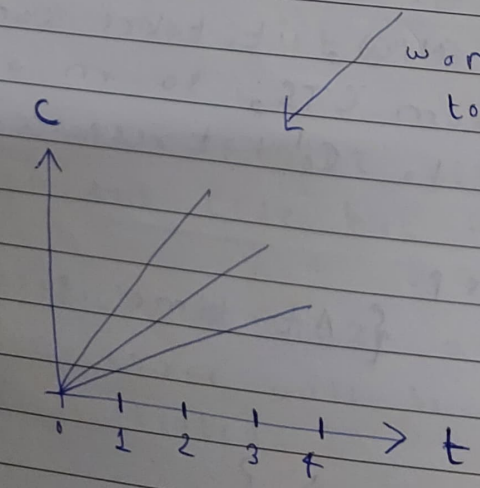
(since t elapsed 1 sec, clock also 1 sec)

For C', time elapsed 1 sec, clock elapsed more than 1 sec
For C'', vice versa

ρ = drift rate, very less number
↳ system dependent
↳ after how much time, you should synchronize

→ slowest to fastest m/c, drift rate = 2ρ

→ Drift in system : 2ρ.Δt



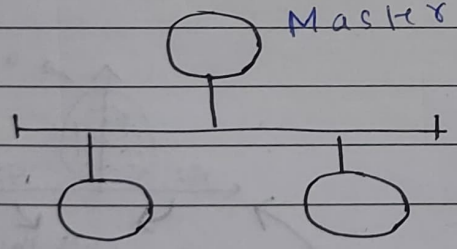
Page :
Date :

→ when to limit 2ρ to δ, after time unit $= \frac{\delta}{2\rho}$, we need to synchronize (send the request to time server and synchronize)

→ Another Problem: only 1 time server, many processes send their request

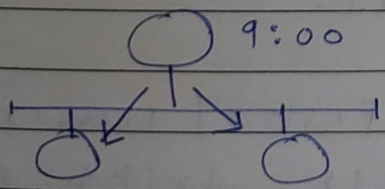
↙
① message loss
② link Failure since no replica
③ Reliability Problems.

↙
Therefore need to move from centralized to Decentralized system.

* 

Master (chosen/elected by system) (this will first broadcast its time), also known as daemon/master

→ Berkely clock Algo.

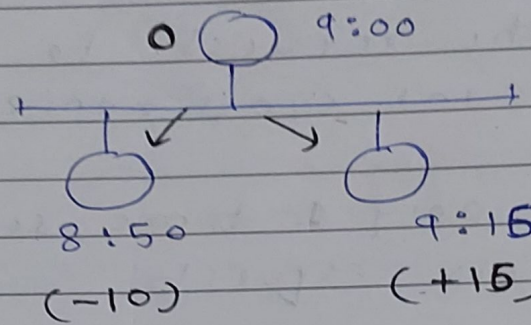


→ If master slowest/fastest, accordingly time will change

~~8:50~~ ~~9:15~~
9:00 9:00 (updated)

Camlin

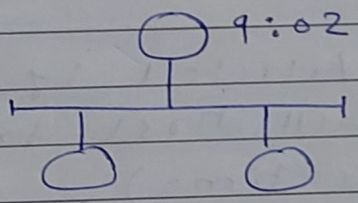
Another solution = Take the average of all drifts and update accordingly

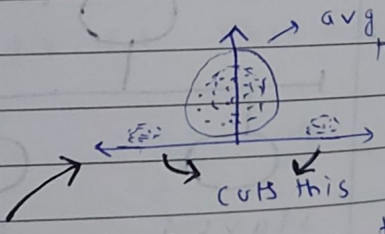


average diff = $0 - 10 + 16 = \frac{6}{3} = 2$

offset

All m/c updated with Master Clock + OFF





Why we avg.? ⇒ eliminating / cutting the highest, lowest drift.

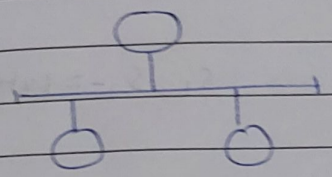
problem = * If master faster / slower and no. of machines less, drift shifts to higher / lower values

* master broadcasting msg. to slave through 1-way communication, some error may be there (due to single point of failure)

→ To cluster clock to Avg. clock time

↙ "Reason for Averaging"

Another Approach: no master/slave



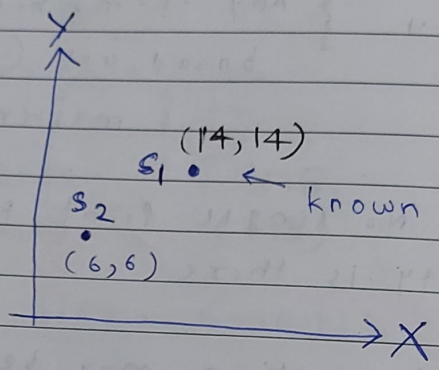
} Any of machine can broadcast (No concept of master/slave)

→ Benefit = No single point failure, hence reliability is there

→ Problem = Congestion may be there, Message passing overhead

lec 2 - AM (12th October 2022)

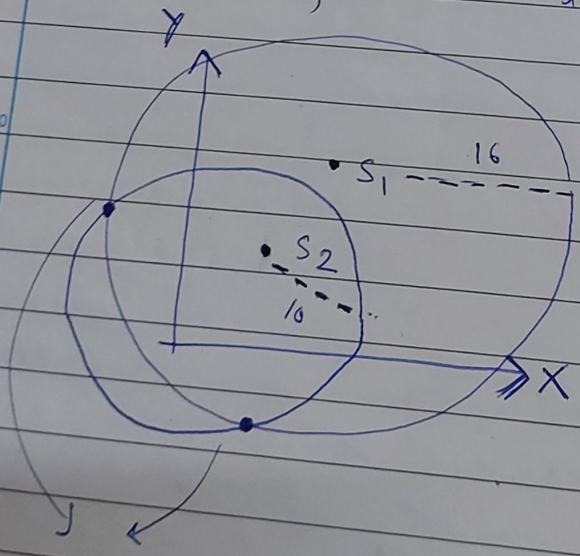
GPS
↳ geostationary satellites
↳ n no. of devices



S_1, S_2 = satellite

D_1 ↙ distance from any known location ↙ from S_1, S_2

From (14, 14), device is at radius 16



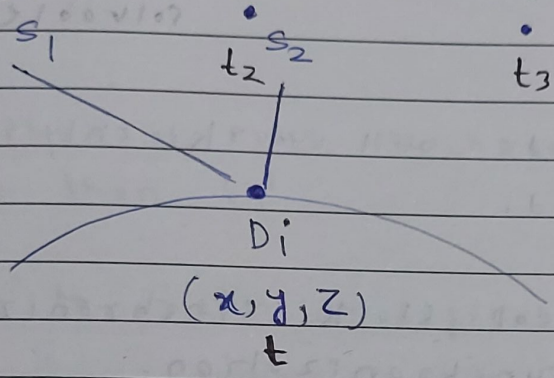
device may be either of these points at ^

* only 2 satellites are not enough for global location of any device

* use as many satellites to exactly know the position of device.

How to calc. distance?

(x_1, y_1, z_1) → known (x_2, y_2, z_2) → known. (x_3, y_3, z_3) → known



unknown = x, y, z

t_1, t_2, t_3 = local clocks

Beacon — its ~~known~~ position, local clock

↳ sent by satellite to device

$$d = \sqrt{(x-x_1)^2 + (y-y_1)^2 + (z-z_1)^2} = \text{Beacon from } S_1$$

$$\sqrt{(x-x_2)^2 + (y-y_2)^2 + (z-z_2)^2} = \text{Beacon from } S_2$$

$$\sqrt{(x-x_3)^2 + (y-y_3)^2 + (z-z_3)^2} = \text{Beacon from } S_3$$

↙

3 unknown, 3 equation but not solvable atleast 4 equations needed to solve non-linear equations.

synchronize after δ/2 time

$s = d/t$

Page :
Date :

For satellite 1:

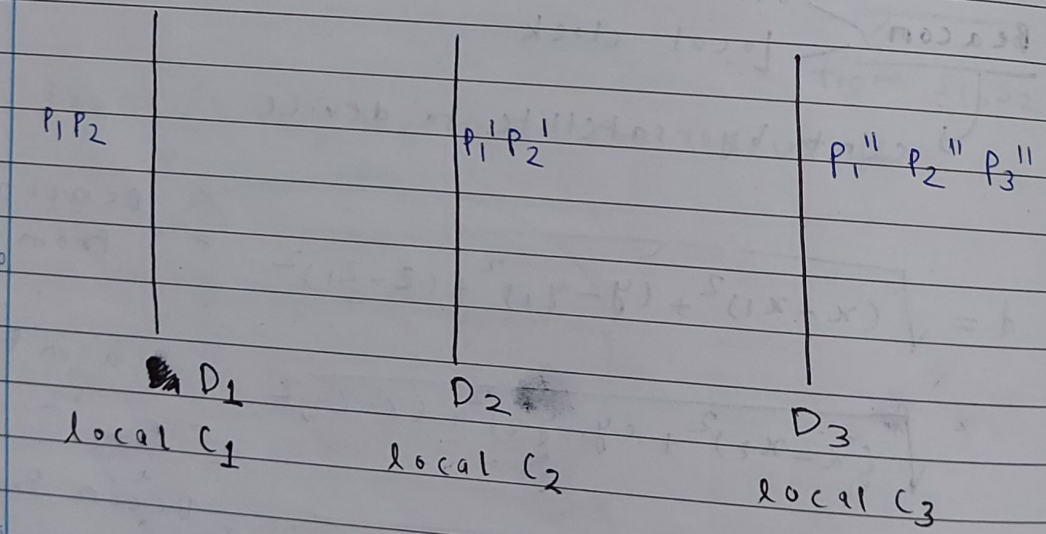
Time elapsed = $(t - t_1)$

$d = c * (t - t_1)$

↳ Now 4 equations are there, therefore solvable

But above methods will work only if clocks are synchronized.

Move from physical clock synchronization to logical clock synchronization.



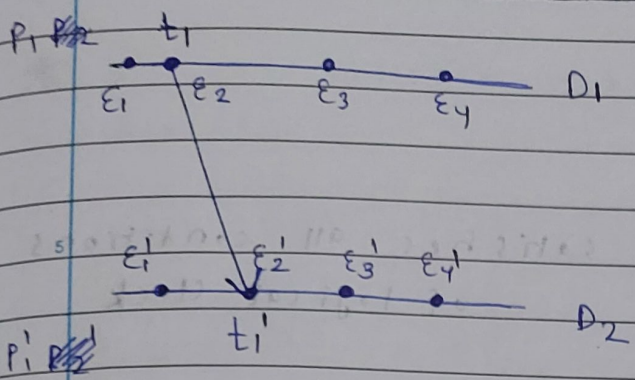
Among $D_1 \rightarrow D_n$, only subset $\{D_i, D_j\}$ are communicating at particular time

If D_3 not communicating with any other device, no need to synchronize

Camlin

Page : relationship
Date :

Lamport Logical Clock (Happend before ~~relationship~~)



At t_1, Let request sent from D_1 to D_2

$\varepsilon_1, \varepsilon_2 -- \varepsilon_4$ = events
$\varepsilon_1' \varepsilon_2' \varepsilon_4'$ ↑

In Physical clock synchronization, t_1 should be less than t_1'

How to order in logical clock synchronization?

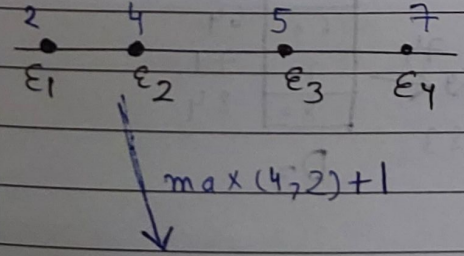
(i) if events belong to same device, ordered by Physical clock time of device ($A \rightarrow B$)

(ii) Send-Receive (Request-response)
↳ happen before than the timestamp of Received message

In above example: $t(\varepsilon_2) < t(\varepsilon_2')$

$A \rightarrow B$, $CT(A) < CT(B)$

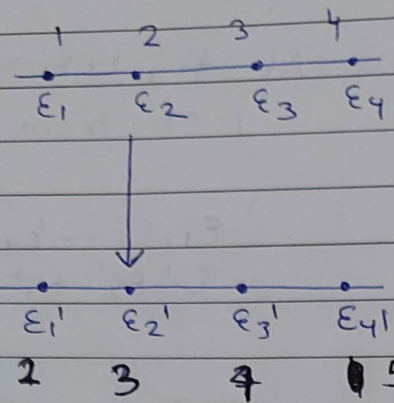
Ex1:



if $t(\varepsilon_2) > t(\varepsilon_2')$

$t(\varepsilon_2') = \max(t(\varepsilon_1), t(\varepsilon_2)) + 1$

Ex 2:



⇒ satisfies all conditions of logical clock

* if A → B, B → C, then A → C

A happened before B

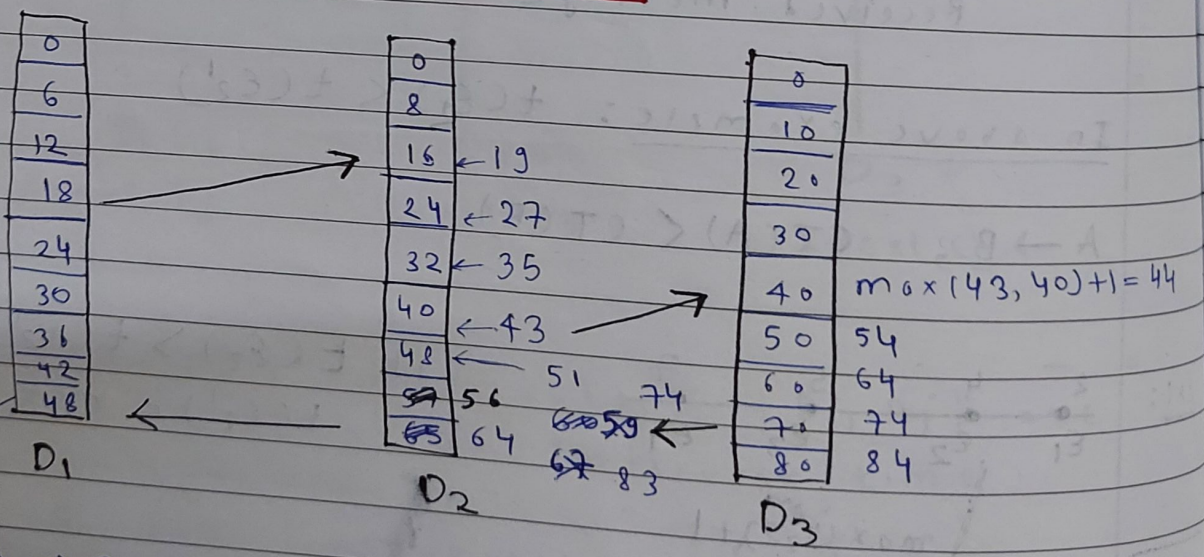
B happened before C

A happened before C

Logical clocks follow the transitivity rule.

Total ordering

Ex 3:



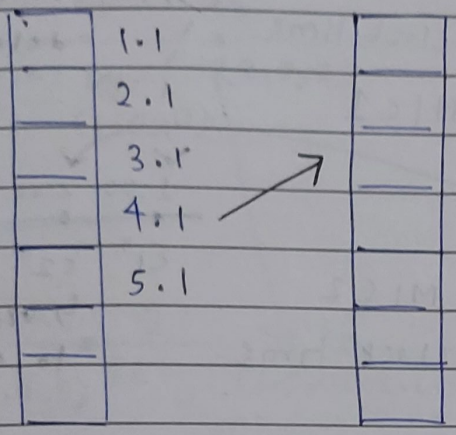
A → B follows for all devices

D_1 & D_3 by-default synchronized according to transitivity rule

event number ← x.y → device number

Page :
Date :

* We can also number the events of different devices.

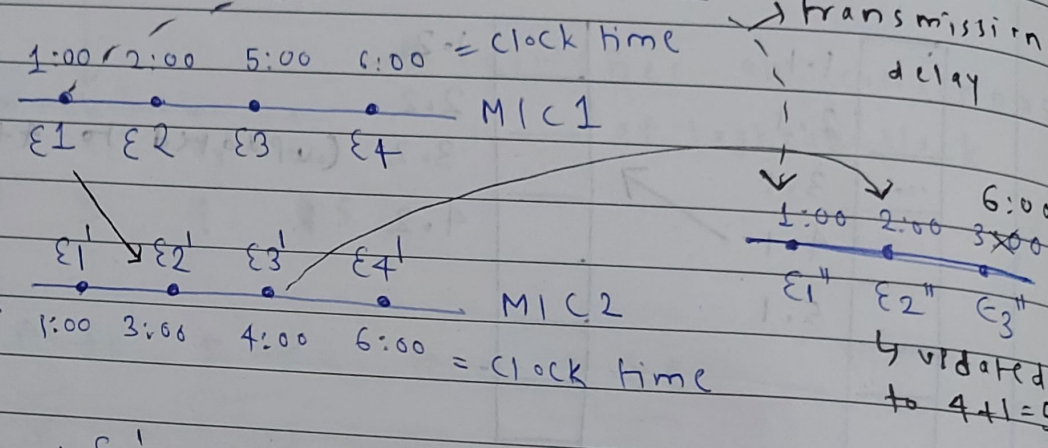


Modifications done similarly to previous one.

lec 3 - AM (14th October, 2022)

Page :
Date :

if $A \rightarrow B$, then $CT(A) < CT(B)$
true is also true



$\varepsilon_1 \rightarrow \varepsilon_2'$, ~~then~~ $CT(\varepsilon_1) < CT(\varepsilon_2')$ true

logically ε_1 occurs before ε_1'', due to transmission delay, it does not reach ε_1'' but msg. from ε_3' reach ε_2'' first.

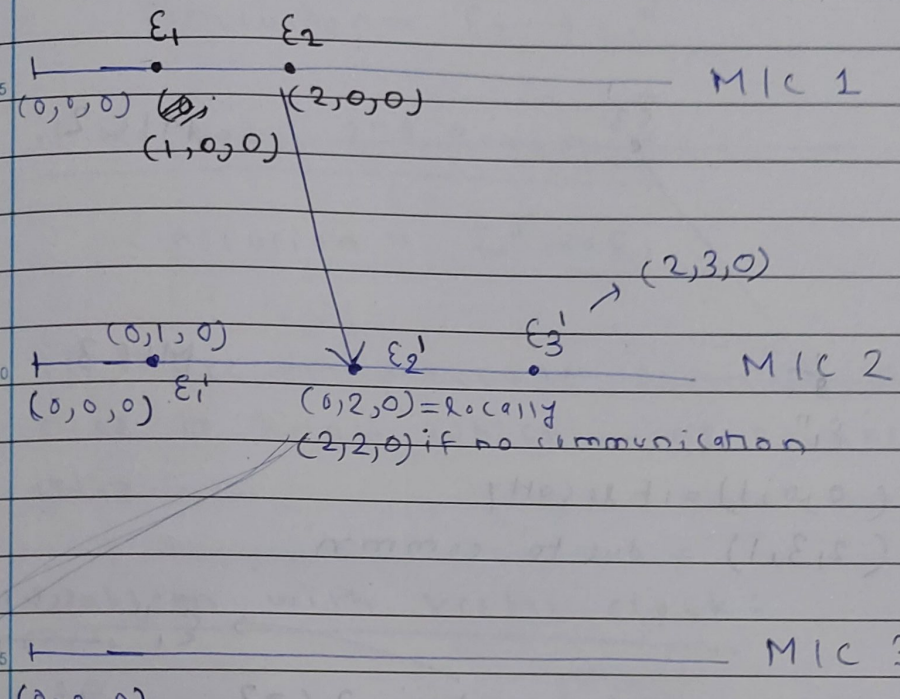
if $CT(\varepsilon_1'') < CT(\varepsilon_2)$, we cannot say that ε_1'' occur before ε_2

↳ limitation of Lamport logical clock (knowing clock time cannot conclude which event occur first)

⟹

send the clock parameters along with msg. in packet

Vector clock



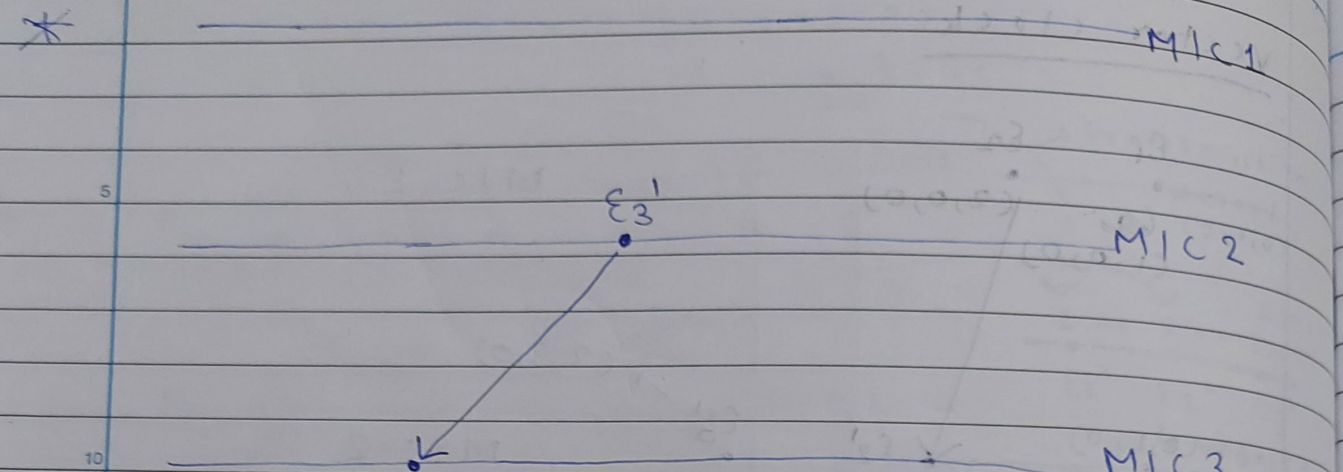
* **Vector Representation:**

$(n_1, n_2 -- n_m)$ = timestamp of m machines

Initially vector = (0, 0, 0)

(0, 2, 0)

↑ ↑ = update the field where comm^n takes place



(0,0,1) = if locally

(2,3,1) = due to comm

Given: $TS(\varepsilon_2) = (2,0,0)$ $\quad 2<=2,\ 0<=3,\ 0<=1$ $\quad \varepsilon_1''$ occurred before ε_2

$TS(\varepsilon_1'') = (2,3,1)$

if $CT_1 < CT_2$, compare all the values of vectors

NOTE: if $CT_i[1] <= CT_j[1]$, then event j occurred before event i

Page :
Date :

& 3rd

if we take 1st parameters: (only)

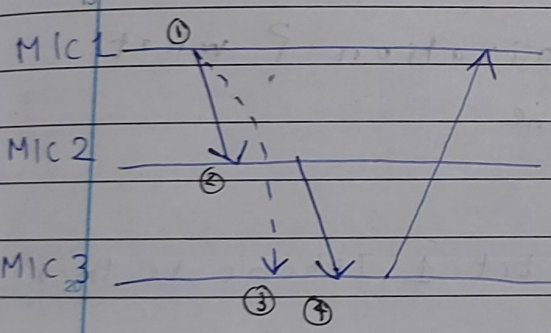
conclusion = $\xi_3 \rightarrow \xi_1$"

if we take 2nd parameter:

conclusion = ξ_1" $\rightarrow \xi_3$

if there is message passing, then only we need to know which event occured before/after.

Problem with vector clock:

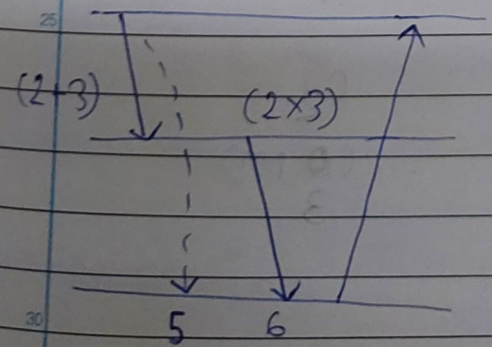


1st event
$P_1 \rightarrow P_2$

2nd: $P_1 \rightarrow P_3$ (same msg. to P_3 also)

3rd: $P_2 \rightarrow P_3$

↳ we expect in this order ① → ② → ③ → ④

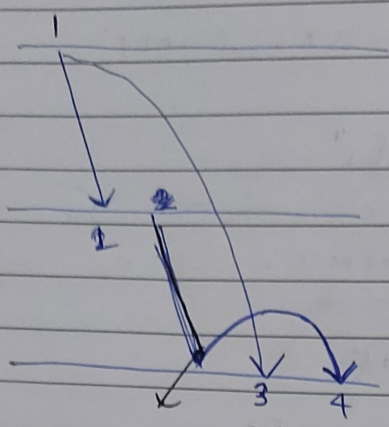


due to N/w delay, if msg 4 arrives before msg 3, there will be problems.

NOTE:
* Along with syncing clocks, the order of messages must also be ~~preserved~~ preserved.

Camlin

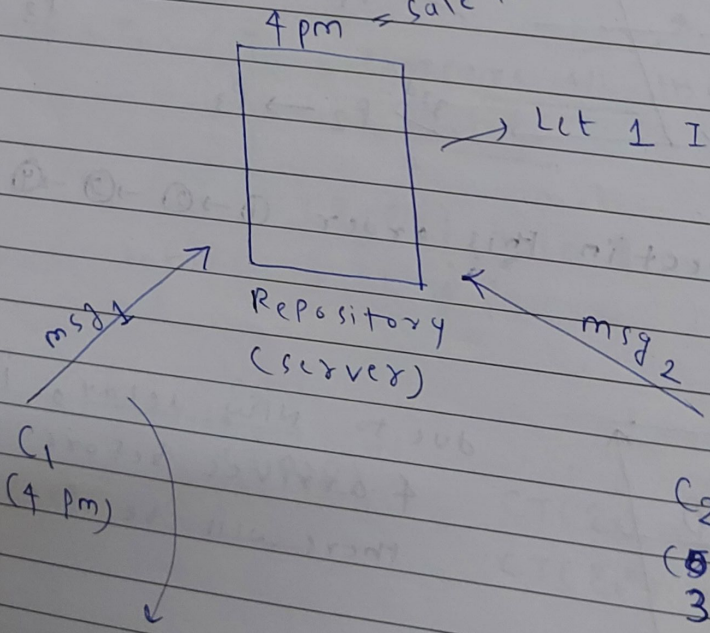
Vector clock = global clock



wait till msg 3 comes

Causal ordering = order of received msg. is same as order of sending msg.

If no Global synchronization ? what problems ?

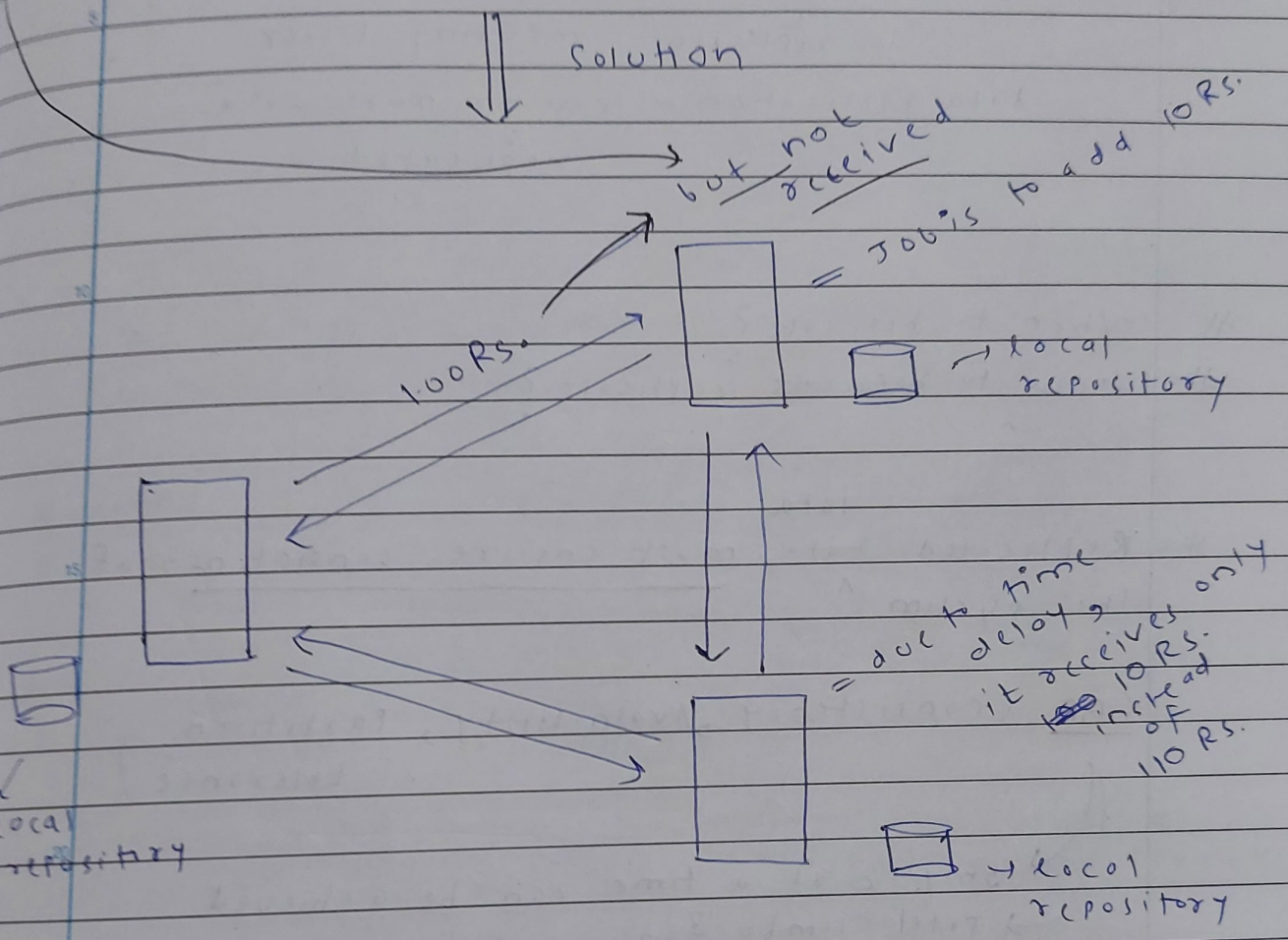


Let transmission time = 1.5 hr
(So C_2 get phone instead of C_1 due to delay)

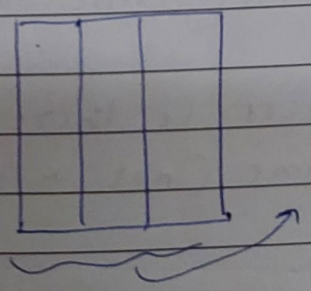
① data loss = If msg. lost
② data duplicacy = if Ack. lost, 3rd m/c receives 210 Rs.

Page :
Date :

Inconsistency (if first come first serve), if no proper ordering of messages (no Global Clock synchronization)



* Logical meaning of snapshot:



combine to form whole Image (no data loss, no redundant ~~picture~~ picture)

↙

if snapshot is consistent

* Take Global snapshot of m/c to avoid data loss & duplicate

Lec (31st October, 2022)

Replication & consistency

Replication — data replication
— computation replication
↓
replicate computing power

data replication = few or more data replicated

When to replicate?
How to ~~replicate~~ replicate?

* Replicated ~~data~~ data must ensure consistency of the system

CAP (consistency, Availability, Partition tolerance)

→ only 2 at a time can be achieved
→ Total comb = $^3C_2$

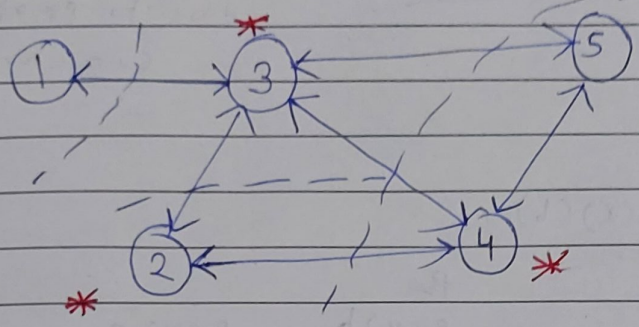
Consistency = machines must have the most updated data. everyone reading same data

Availability = update of data should be there at all machines at that same time (not wait for any other request)

Fault tolerance
scalability > replication

Page :
Date :

Partition tolerance ⇒



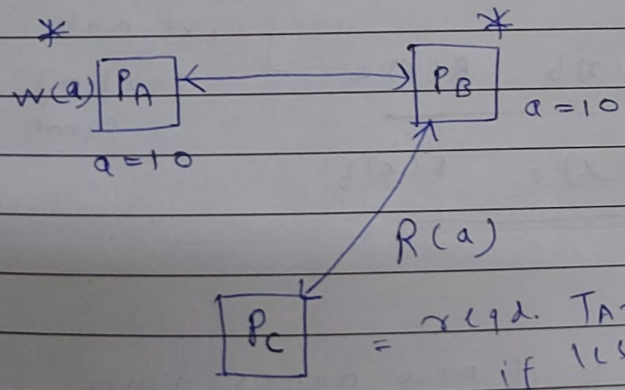
* = replicated data

if 3 logs off, graph divides into 2 parts whether the system can continue the work or not after 1 device logs off ?

* sync. with help of logical clocks
* computation stuck in 1-to-1 mapping

* response of the server ⇒ depends on frequency of server.

Strict consistency



= reqd. $T_A + t$ to reach P_B if less than $T_A + t$, P_C reads a old value of a

* let update at T_A & t time to reach replicated system

$T_A < T_B < T_A + t$ ⇒ this does not happen in strict consistency

* very hard to achieve in run time.

Camlin

~~easy to achieve~~

Sequential consistency → weaker than strict consistency due to protocol can change

P_1	W(x) a	
P_2	W(x) (b)	
P_3	~~R(x)~~ R(x) b	R(x) a
P_4	~~R(x) b~~ R(x) b	R(x) a

communicating with each other, and agreed upon a protocol in which order they should read ~~execute~~ (All of them would have same data)

P_1	W(x) a		not allowed since P_3 & P_4 are not reading in same order
P_2	W(x) b		
P_3	R(x) b	R(x) a	
P_4	R(x) a	R(x) b	

* Update the protocol if new nodes join

→ read in same order as ~~write~~

Causal Consistency

* maintain same "happen-before" relationship

Linearization

Linearizability → between strict & sequential

P_1	P_2	P_3
$x = 1$	$y = 1$	$z = 1$
print(y, z)	print(x, z)	print(x, y)

x, y, z = 0 (Initially)

→ if this sequence:

```
x = 1
print(y, z)        → o/p: 0 0 1 0 1 1
y = 1
print(x, z)
z = 1
print(x, y)
```

→ valid since we are reading either not updated value / immediate updated value

*

```
x = 1
print(y, z)
y = 1
z = 1
print(x, z)
print(x, y)
```

Consistent

TS(W) > TS(R)
↳ not allowed

Page :
Date :

* x = 1
print (x, 2)
print (y, 2)
y = 1
z = 1
print (x, y)

} not valid (not consistent)

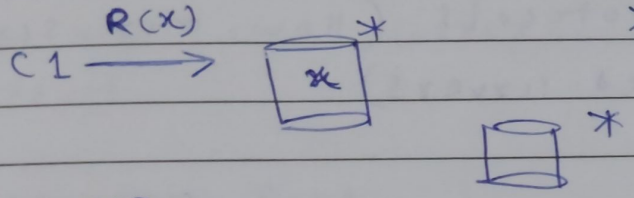
Page :
Date :

Lec (4th November, 2022)

Client-centric Consistency Models

↳ perspective from a single client

* Monotonic reads => returns same / more recent data



a) x = 10
b) x = 11
c) if x = 10 again, ~~older~~ & newer 10 cannot be distinguished

* Monotonic writes => replication of data moved in order to ~~servers~~ servers.

* Read your writes => whenever we write / update data, read the new data always.

* Writes Follows reads ⇒

abc@ - time 0
xyz# - time 1
↳ mno$

update done on most recent update (on time 1 not time 0)

→ eventual consistency → assumption that updates will be done to replicated servers ~~eventually~~ eventually

↙

CAP theorem { A and P maintained ~~omit~~
C compromised

Camlin

triggered updates = updates only when one of replicated server update

Periodic updates = updates after a certain period of time

Page: Date:

epidemic protocols

mostly all systems go for triggered based updates

How to send updation copy to other servers ?

↙

through epidemic protocols (How msg. spread out to replicated servers)

Spreading an elidemic
- Anti-entropy (since we are randomly picking any server)
- Rumor mongering

Updation
- Pull = P updates, other server pulling update
- Push = new update, sending update to other server
- Pull-Push
  - ↙
  - Pull the new update and also push the new update

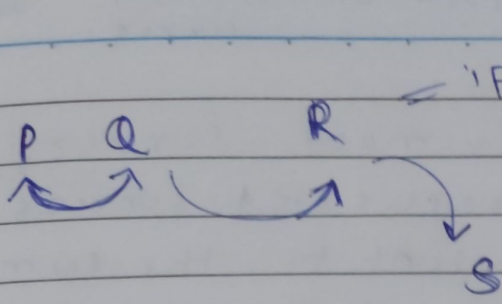
if only pull => one way comm., longer time to spread the epidemic

& push

Push-Pull (Pairwise update) = takes less time

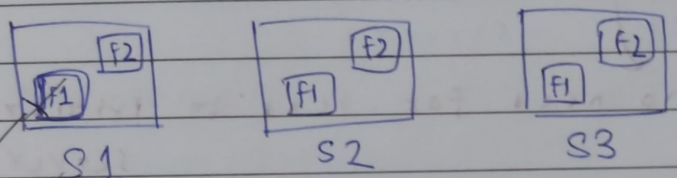
(Gossiping)

Rumour mongering => N replicated server, $^{N}C_2$ pairs, if any update among pairs, updates spread



(Anti-entropy)
Faster than previous, but no guarantee that all replicated servers have same/new update

→ Removing Data



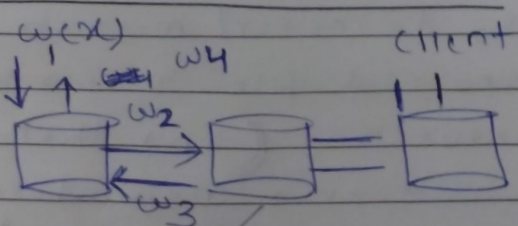
if deleted & time takes to update to other server

Solution = Death certificate issued.

each & every time, validate death certificate, therefore maintaining a log incurs overhead.

Replication
- Remote write
- Local write

Remote write protocols



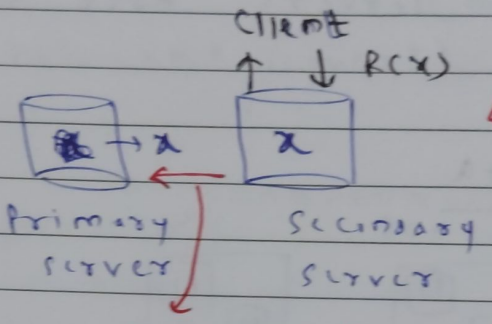
→ 1 server = primary server
other server = secondary server

Remote write = update to every other servers if update on primary server

→ 1st update on the primary server, then only updates to other replicated servers

→ Acknowledgement is sent to the primary server

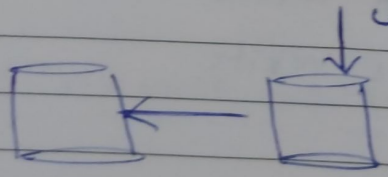
→ Forward the request to the particular server since ~~only the~~ all servers are replicated



Backup server = if primary server fails, it takes over charge.

Local Write Protocols — no concept of primary server
— current/main server is there

Migrating the file to other servers.



Migrating local data to local place where accessing rate is high.

rather than forwarding write requests, transfer/migrate/pull the required file to server requesting (and that server now becomes main server)

Page :
Date :

Problem = relocation transparency must be maintained, much more write frequency by different clients, local write protocol not feasible. (go for remote write in this case)

Synchronous replication = wait for ACK from replicated servers whether sync done or not., coordinator or leader will wait for ACK

Advantage = if ACK have come, full surety that all replicated servers have updated

Disadvantage = waiting time more for one single operation.

Async replication = like ~~eventual~~ eventual consistency, not wait for all ACK, operation successful if ~~no~~ ACK from majority of servers.

Advantage = must faster

Disadvantage = ① no guarantee all servers in consistent state.

② error when access to server from which no ACK

Lec (5th Nov, 2022)

→ Fault tolerance

Fault less with help of replication

If 100 m/c instead of 10, prob. of fault more

Probability of failure ∝ (depends on) no. of m/c attached to system

Single m/c → failures all or nothing

Distributed systems → Partial failures

→ Dependable systems : → Maintainability (can recover from failure without any loss)

availability (measured through 9's concept
(system should be live at time of access)
4 9's concept = 99.99% available, 5 9's = 99.999% system available)

High available system
- 5 9's = 99.999 %
- for critical request-response
- like rocket launcher

→ Reliability

→ Safety (partially affected client should not harm the whole system)

→ Types of faults:

1) Transient faults = failure occurred only once and it never and then never comes back (even if comes back, freq. is less)

2) Intermittent faults = getting error in time intervals.

3) Permanent failures = one major fault, entire system crashes (not able to recovery).
   ↳ much more easy to detect & maintain
   ↳ major fault (since 0/1 = either m/c working not working)

① & ② = hard to detect & maintain

→ Failure models:

→ Crash failure ⇒ A server halts without any notice & no chance of recovery, easy to detect and maintain.

→ omission failure ⇒ receive ommission
                     send ommission
   ↳ handles response request from client

send = not able to send msg.
receive = not able to receiver

→ Response failure → Value failure (expected result not recd, like garbage value)

→ State transition failure (order of responses not right)

→ Arbitrary failure ⇒ ~~not able to get response i.e. ~~ sometimes it timeout failure, response failure, system is confused which fault tolerance is required.

Byzantine fault

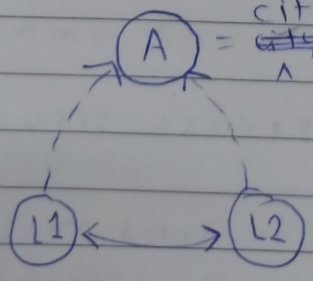
→ Agreement in faulty systems:

* Atleast k+1 replicated systems for fault tolerant system

* k-Fault ~~systems~~ tolerant

* How to detect erroneous machine?

→ How to detect Byzantine failure?



2 general Problem

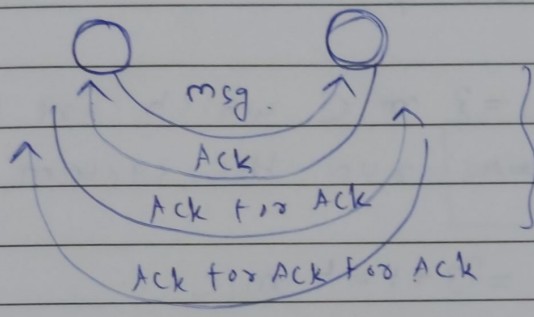
L1, L2 = generals. Let's attack at 9 or at 10

Ack for every request

Page :
Date :

→ communication channel = eavesdropping may be there, security issues, due to which expected results not there.

→ Infinite loop of final Acknowledgement of the acknowledgement



Byzantine general problem

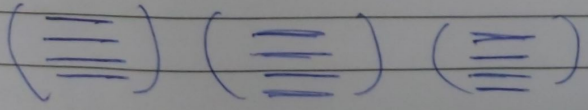
→ Iterative ~~algo~~ Algorithm

→ Faulty machine sends different msg. (garbage values)

→ may give correct result / wrong result

a, b, c, d = arbitrary msg from m/c 3

→ Comparison of results (rows) of all m/c, if they are different, that m/c is fault



→ n machines, k machines may be failure
k subset of n, k ∈ [0, n]

Page :
Date :

→ If m/c ~~comin~~ more, detection of fault is difficult due to large no. of comparisons.

→ 3m + 1 Process & 2m + 1 agreement

Ex: 4 m/c

let 1 m/c = faulty

m = 1

if I have 2m + 1 = 2 × 1 + 1 = 3 m/c which are perfect, we can run the system

If Total 3m + 1 = 3 + 1 = 4 m/c

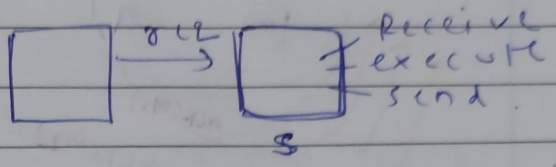
out of 4 m/c, 3 are working fine, then system works well

Page :
Date :

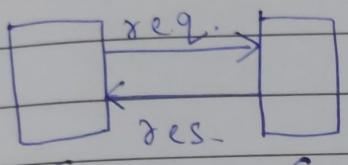
Lec (11th Nov, 2022)

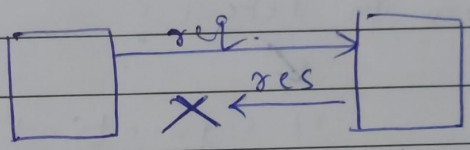
Agreement
- 2PC
- 3PC

Reliable Comm n.



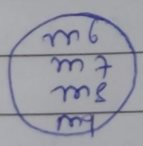
(if req lost, timeout can apply)



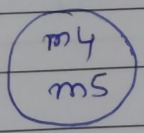


response not reach client

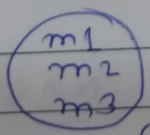
faulty unicast system (one-to-one)



C1



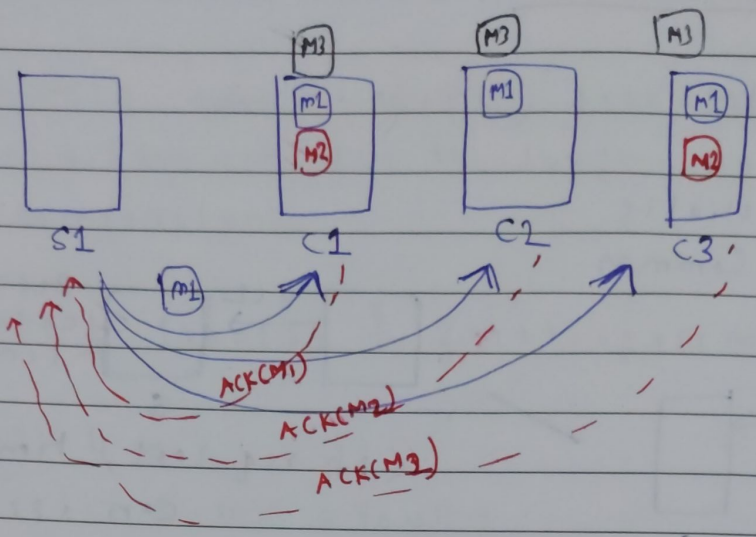
C2



C3

ACK

NACK



How to detect?

↙

if sequential ~~number~~
(every message has a unique sequence number, and if it does not find seq. no, then it sends NACK)

Acknowledgement explosion = Scalability Problem

↙

Problem of ACK based Protocol

No. of messages is less in NACK based Protocol

↙

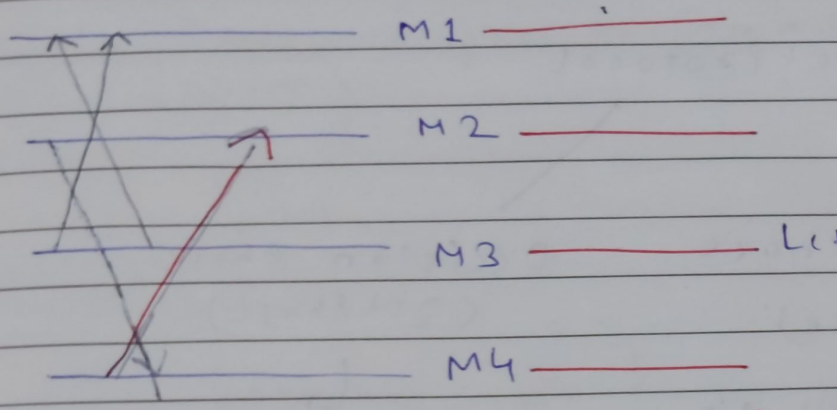
only m/c which is in failure, sends NACK

Atomicity property = either whole transaction occurs or nothing occurs

Page :
Date :

↗ analogously same as Global snapshot

Group view Semantics



Initially machines form a group, Let group = {1,2,3,4}

regrouping {let regroup = {1,2,4}}

In case of fault → abort

time
at ~~time~~ = t_2

M1

M2

M3

M4

(either all are aborted or again a regrouping (through agreement protocol with other live m/c))

Camlin

Distributed commit

2 Phase Protocol

- Voting Phase (1st phase)
  - System will go for voting whether all agree to commit, regroup or abort
- Decision Phase (2nd Phase)
  - Coordinator will send the global decision to subordinates whether to commit/ abort

even if 1 machine votes for abort, whole system aborts.

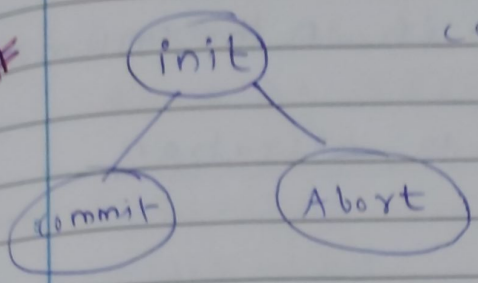
Coordinator = 1 m/c ← if n m/c

Sub-ordinates = (n-1) m/c

according

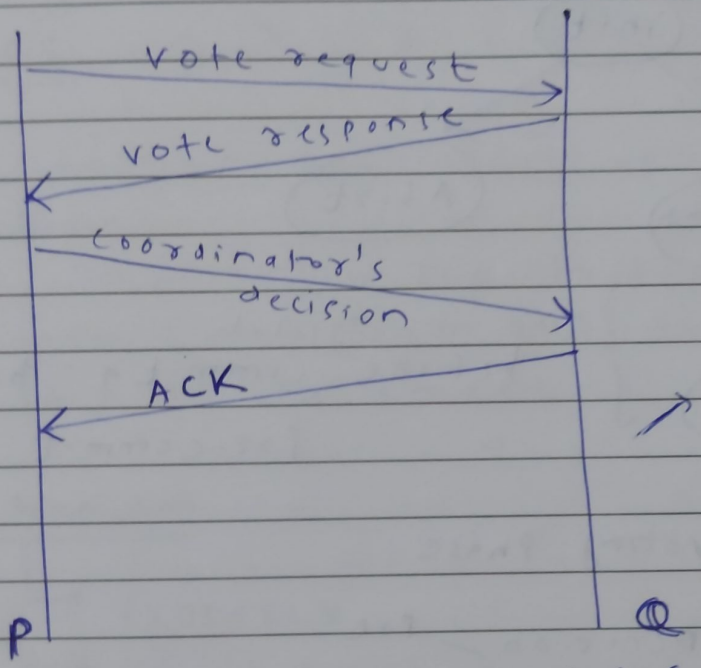
These selected to leader selection algorithm

All or none (Atomic)



coordinator asks for vote req. from subordinates, then it goes to commit OR Abort.

Problem with 2PC:



P = coordinator

* After ~~sending de~~ receiving vote response, coordinator fails (Single Point of Failure)

* if Q sends abort, then whole m/c abort
if Q sends commit, whole system deadlock

* if Q crashed

if after recovering, in abort phase, then abort

if in commit phase

received commit, then commit

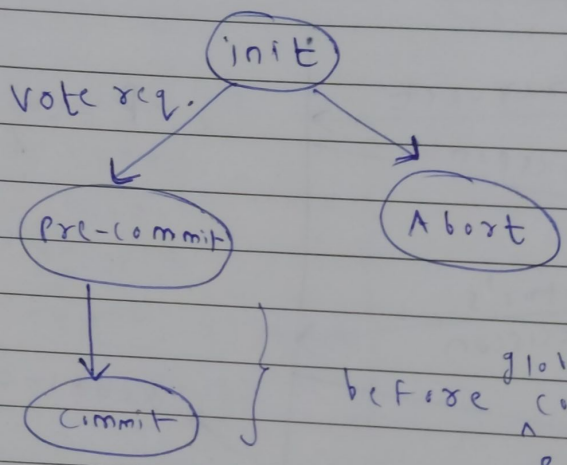
voted for commit, wait for coordinator's message

till Paxos & Raft (After midsem 1cc)
= 0 viz 2

Safety = ensures no harm is in the system

Liveness = no progress due to operation

3-Phase Commit (3PC)



before global commit, there is Pre-commit

3PC — Voting Phase
Decision — Pre, Global

→ if everyone sends ACK in Pre-commit, then system goes for global commit.

→ if 1 commit & N-1 precommit, then broadcast message of commit

LLC (14th Nov, 2022)

→ In Byzantine fault, agreement cannot be done by 2PC/3PC.

→ Byzantine fault handling harder to implement.

→ Crash failure easy to implement

→ **Paxos** → Iterative Algorithm

→ they do not take decisions from all the machines

→ Here, the leader try to take majority of response (not from all) to take decisions.

→ Not easy to implement

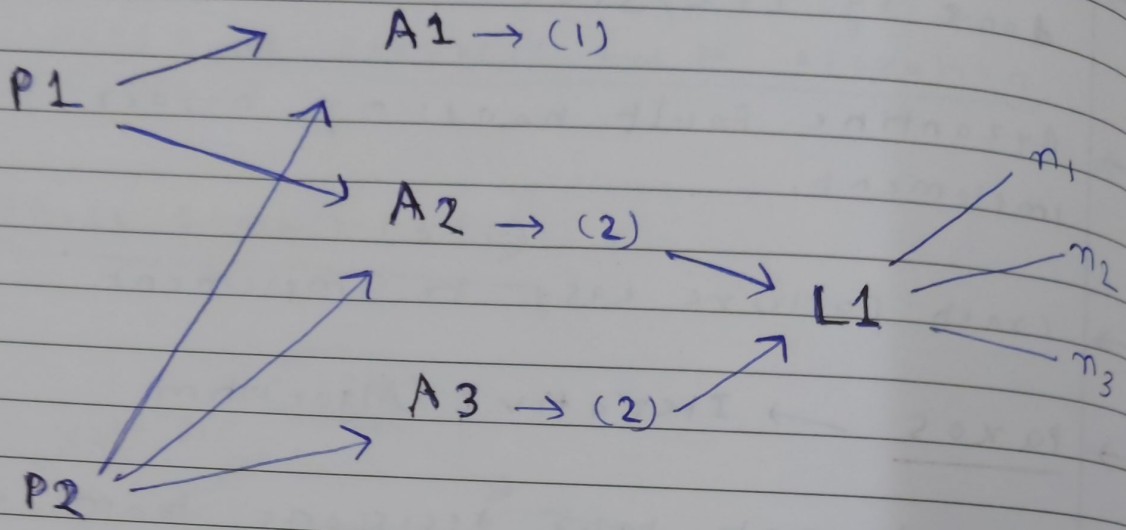
Mechanism
- Proposer
- Acceptor
- Learner

* Proposer ⇒ will propose it wants to update/write some value that I am proposing that these will be the new value. Can be single/multiple Proposer

* Acceptor ⇒ ~~After~~ ~~accepting~~ can accept/reject Proposal. Can send ACK/NACK. ~~After~~ send the response to learner node.

Not necessarily, proposals will be sent to all Acceptors.

Let N = nodes.



(1) = Accepted ^ from P1 (proposal)
(2) = Accepted ^ from P2

System look for majority of proposals. Hence (2) gets accepted and sent to learner and further sent to nodes.

Phase 1: proposal creation

Create/ Propose (Proposer)

* sends a delimiter kind of msg. (send number to other node that it has a proposal)
* sends 'n' from ith Proposer (n_i)

Promise (Acceptor)

* Accept / Reject the n_i

* Check if n_i is greater than ~~previo~~ number of previous Proposal

Ex: P1 ⟶ n_1

A1 ⟶ Check if $n_1 > n_2$

n_2 : previous Proposal

if true, then Proposal Accepted else rejected

* if Accepted, then it sends a ACKNOW-LEDGEMENT (Promise message). After receiving Promise message, it sends the original message.

↳ (n, v)

↳ value

↙ Proposal number

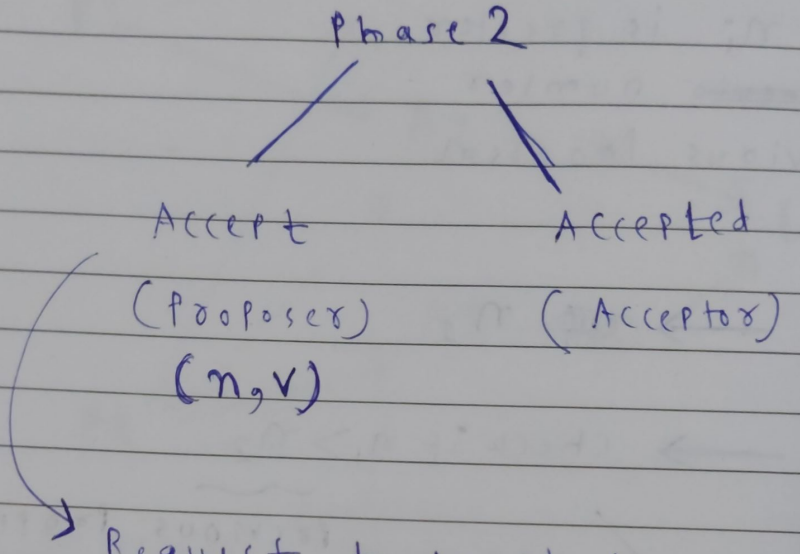
* if rejected, it sends NACK along with reason of not accepting.

* Validity
~~Validity~~ } By sending ACK/NACK
Safety }

Check Bounce (when not enough money in bank account) ⇒ Paxos

Page:
Date:

Phase 2 : After Phase 1, how database updated



Request to Accept the value by the Proposer (after receiving majority of Promises).

* **Accepted** = Final / Termination Phase, receives the value and updates the database with the value ~~accepted~~ accepted.

* if Proposer fails, Acceptor waits for some timeout period.

* if "no majority", all the Proposers restart with new sequence numbers. and according to ~~Proofs~~ Probability theory, since it is iterative algorithm 2 or 3 iterations the majority of ~~Acceptors are there~~ Proposals are accepted.

can also be used in "leader selection" problem not only as a "fault tolerant" mechanism.

we count the no. of Acceptors where most of promises are sent.

$t = T_1$

P_1 n_1

P_2 n_2

if P_1 has most no. of promises, then it is the leader, in case of Tie-breaker, start from the next iteration.

every replica has most recent update from the same Proposer (not necessarily, most recent update from system).

no conflict of instruction

learner can learn updations from Proposer

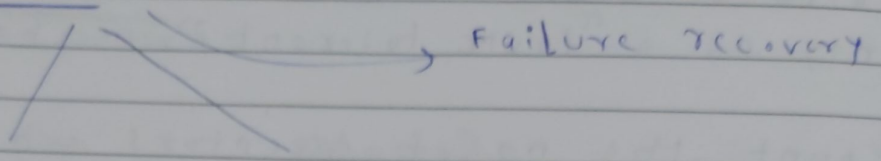
validity, liveness ~~is~~ is there

failure

Crash Failure / Fail stop ~~failure~~

lec (16th Nov, 2022)

RAFT = 2 Phases



Leader election — Log replication

Leader
Candidate
Follower

→ Initially all m/c are in follower state

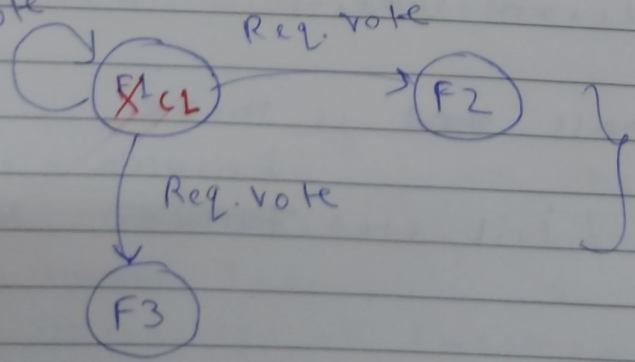
F1 — 150ms
~~150~~ 150ms

F2 — 200ms

F3 — 230ms

150–500ms.

→ After a timeout expires for 1st candidate, it becomes a candidate for leader election



→ if 2 m/c have same timeout, both will become candidate for leader selection.

Camlin

→ whichever m/c have high votes, that will be leader

→ if tie, start the iteration again as all as follower and timeout may be different.

→ split vote

→ Leader election gets activated when

- split vote
- Leader selected in 1st iteration gets crashed

→ (L) (F2) (F3)

* Followers neither execute any request
* not ask for any vote
* Just follow the leader as per the instruction sent by the leader.

→ RAFT use RPC
- to request vote
- to update log table

when there

→ Each server maintains a file, containing sequential requests that come up to that server. This is called log file.

Adv. of RAFT = decompose failure recovery into subproblem

Index	1	1	1	2	3	3	4	} Term No.
Term No.	x ← 4	y ← 1	z ← 3	x ← 0	y ← 2	z ← 0	x ← 6	⟹ Let this be leader
Value/ data								

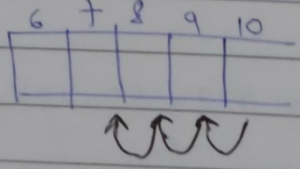
→ Use log Table < missing value / New updation need to be made

Case 1: F1 — 1 2 3 4 5 — y ← 2 ⇒ revert back

Case 2: F2 — 6 7 8 9 10 — z ← 4, x ← 1, x ← 2, y ← 3 ⇒ revert back till matches append rest

→ if term value greater than term no. follower has, then accept log file.

→ In case 2, revert back to 6, append the



rest of log file from leader

"Strict Write only" — If new leader, new updated values need to be made

Quiz 2 - 3.2, 3.3 till RAFT (6.1) (3.3 - 6.1)

endterm = whole syllabus

~~till RAFT = endterm~~

→ Log Replication uses 1 RPC call = "Append Entry"

→ when updation, each follower will send an ACK

→ Leader will send COMMIT msg. of the last (will wait for majority of ACK from followers, send the termination msg.)

→ Check the liveness of leader

→ Heartbeat msg. — used to check whether still any leader presents, or need to elect a leader, Heartbeat msg. sent periodically.
leader will select a time period after which it will send Heartbeat msg.

→ RAFT ensures — Safety (always progress even when leader crashes)
— Liveness

Safety = No other leader can change the index in log file, only 1 leader each time.

→ Recovery is more easier than Paxos
(crash) fault detection easier
Implementation also easy
faster Algorithm than Paxos.