# UNIT-I

## Distributed System

- A Distributed System is one in which components located at networked computers communicate & coordinate their actions only by passing messages.

- It is a no of autonomous processing element (not necessarily homogenous) that are interconnected by a computer Network & they cooperate in processing their assigned task transparently.

  processing element → Computer / process / processor

  Autonomous → Nodes/element must atleast be equipped with their private control.

  Interconnected → Node must be able to exchange information.

  Transparent → It implies existence of autonomous nodes is transparent to the other users.

- What actually is distributed in distributed system –

  ① Processing logic – processing element with different logic are distributed

  ② Function → No single computer have designed with considerably lot of functionality. ( Due to Cost factor, Maintainance factor).

  ③ Data → Data can be replicated & fragmented as it better to distribute Data for, application such that it is near by it is redundant.

  ④ Control → Control is not with the single site, but control is distributed so that 'autonomy of control' is not there.

Distributed System has the following significant consequences →

① Concurrency.

② No global clock.

③ Independent failure.

http://www.abc.com/distrib/index.html
↳ protocol (HTTP)
↳ Domain name of the web server Hosted at www.abc.com.

Example of Distributed System :-

(use of Network in selected domains)

① web search –
② online games –
③ Finance Trending (market)
④ Internet
⑤ Intranet
⑥ Mobile & ubiquitous computing.

- Finance & Commerce
- The Information Society
- Creative Industry & entertainment
- Healthcare
- Education
- Transport & Logistics
- Science
- Environment Mgt

Challenges →

① Heterogeneity. (DS must be constructed from the variety of different Net^ks, OS, computer H/w, & programming languages.
② Openness. (DS must be extensible. (Develop Interface for DS)
③ Security. (DOS attack is a big problem) (using encryption)
④ Scalability. (DS need to be scalable & cost of adding a user in a constr amount of in terms the resources must be added.
⑤ Failure Handling. (Any proces, computer or net^k may fail independe of the others.).
⑥ Concurrency.
⑦ Transparency. (_ Application programmer →).

Motivation →

Resources sharing is the main Motivation for construction DS

Resources as – Printers, files, web pages or database records are managed by servers

# System Models



- Systems that are intended for use in real world environments should be designed to function correctly in the widest possible range of circumstances and in the face of many possible difficulties & Threats.
- Each Model types is intended to provide an abstract, simplified but Consistent discription of a relevant aspect of distributed system design.

① Physical Model → The most explicit way in which to describe a system, they capture the H/w composition of a system in terms of the computers, Mobile Phones, & their interconnecting Networks.

② Architectural Model → Describes a system in terms of computational & communication tasks performed by its computational elements, the computational elements being individual computers or aggrigates of them supported by appropriate Network interconnections.

③ Fundamental Model → Takes an abstract perspective in order to examine individual aspects of a distributed system.

**Difficulties & Threats for distributed Models →**

- → widely varying modes of use.
- → wide range of system environment. (It accommodate Heterogenious H/w, OS & Net's)
- → Internal Problem. (No synchronized clocks, conflict data update & many mods of H/w & S/w failure involving the individual system components.)
- → External Problem. (Attacks on data integrity & Secrecy, Das attacks)

## ① Physical Model →

- It is a representation of the underlaying H/w elements of a distributed system that abstracts away from specific details of the computer & networking technologies employed.

ex Baseline Physical Model -

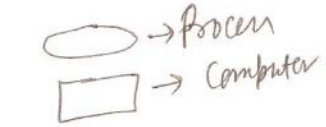
Where H/w or S/w Component located at Networked Computers Communicate & coordinate their actions only by Passing Messages.

# Communicating entities & communication Paradigms :-

| Communicating entities (what is communicating) | | Communication entities (How they communicate) | | |
|---|---|---|---|---|
| System-oriented entities | Problem-oriented entities | Interprocess Communication (API) with socket (Pg no.) | Remote invocation (Two way exchange) | Indirect communication |
| Nodes | Objects | Message Passing | Request-Reply | Group Communication (One to many) |
| Processes | Components | Socket | RPC | Publish-Subscribe |
| Threads | Web services | Multicast | RMI | message queues (Point to Point Service) |
| | | | | Tuple Spaces (where process can place arbitrary items of structured data) |
| | | | | DSM (Distributed Shared Memory) |

(2) Architectural Model :-

(a) client server Model (search engine) :-



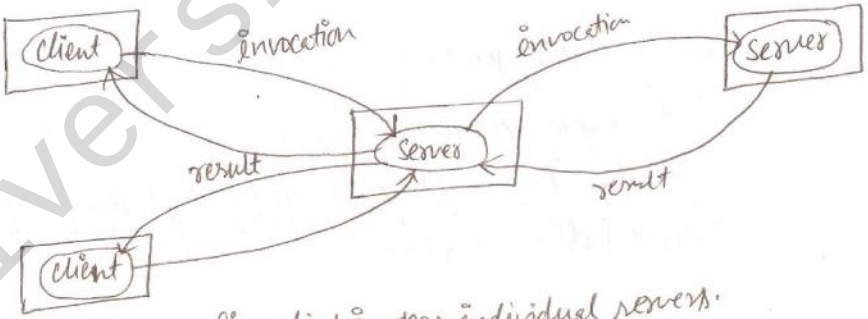




fig- client invokes individual servers.

(b) service Provided by Multiple servers :- Here data is partitioned & replicated to increase performance & availability & to improve fault tolerance.

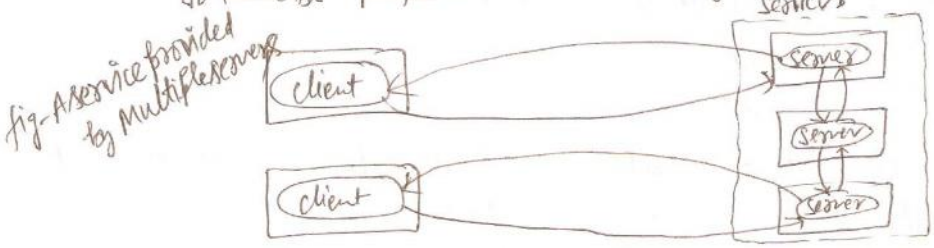



fig- A service provided by Multiple servers

© Proxy Server & Caches :-

- A proxy server provides a shared cache of web resources for client m/c at site or across several sites.
- A cache is a store of recently used data objects that is closer than the objects themselves.

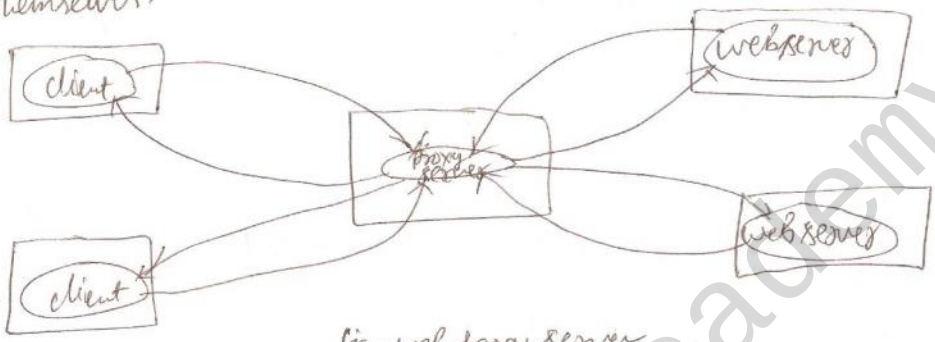



fig - web proxy server

(d) Peer Processes :-

- All process play similar role, interacting cooperatively as peers to performs a distributed activity or computation without any distinction bet$^{n}$ client & server.

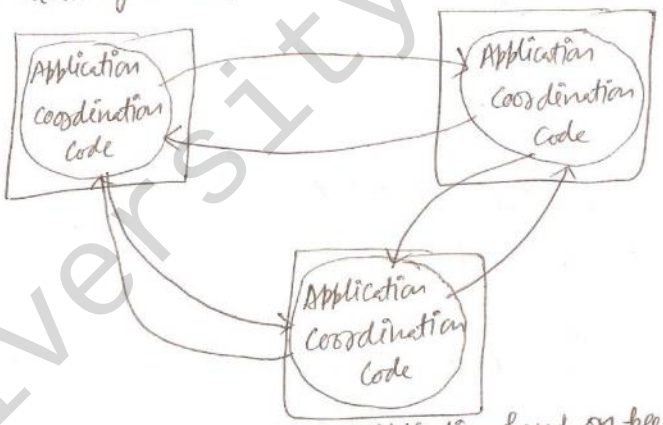
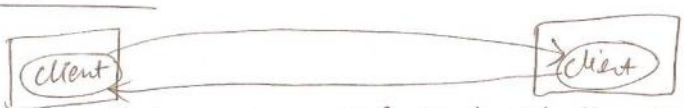
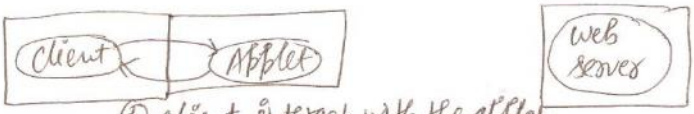



fig - A distributed application based on peer process.

(e) Variation of client-server Models :-




(a) client request results in the downloading of Applet code




(b) client interact with the applet

fig - web applets

① Mobile Code ( Applet)

② Mobile agents: A mobile agent is running program that travels from one computer to another on network carrying out a task on someone's behalf such as collecting information eventually returning with results.

③ Network Computer → It downloads the operating system & any application s/w needed by the user from remote file server. Application are run locally but the files are managed by remote file servers, thus user can migrate to another m/c easily.

④ Thin Clients → The term thin client refers to s/w layer that support a window based user interface on a computer that is local to the user while executing applications programs on a remote computer.

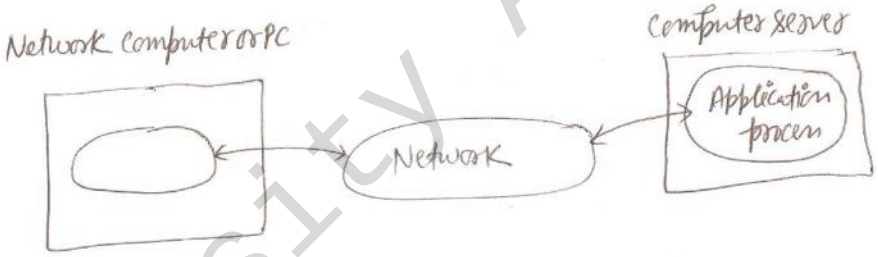



fig - Thin clients & Computer servers

⑤ Mobile devices & Spontaneous networking →

Mobile devices – GSM, Bluetooth, IR etc.

Spontaneous net^k → Mobile devices + other devices e.g PDA

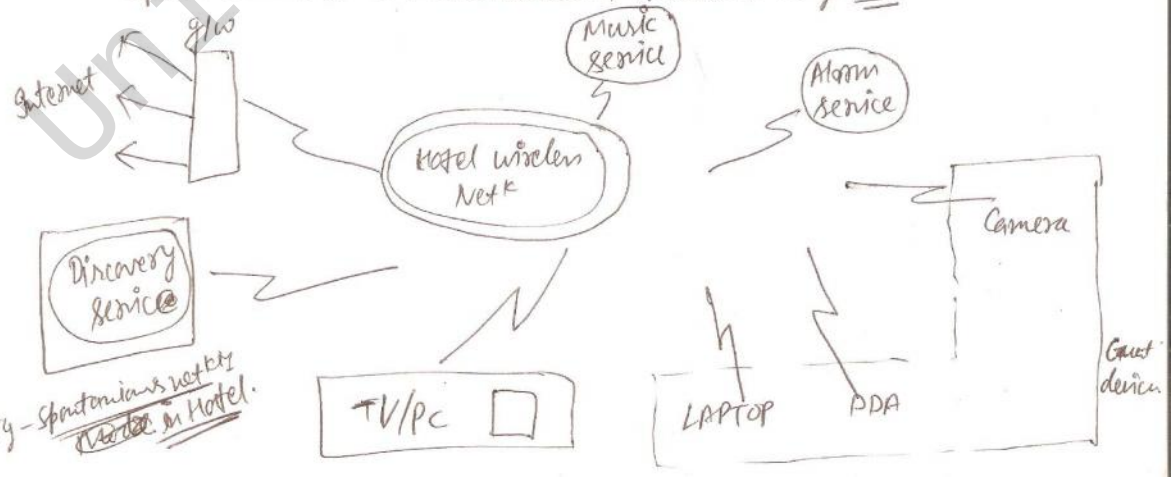



fig - Spontaneous net^k in Hotel.

③ Fundamental Model includes –

(a) Interaction Model → It is concerned with performance of process communication channels and absence of global clock, Interaction Model is further classified as Synchronous system.
(i.e bound on process execution time, message delivery time & clock drift)

(b) Asynchronous system → There is no-bound on execution time, message delivery time & clock drift as Internet.

(c) Failure Model → It classified failure of process & basic communication channels in a distributed system. It is based on masking, integrity & validity approach.

(d) Security Model → It identifies the possible threats to processes & communication channels in a open distributed system such as, integrity, auth, privacy, etc.

## Limitation of Distributed System →

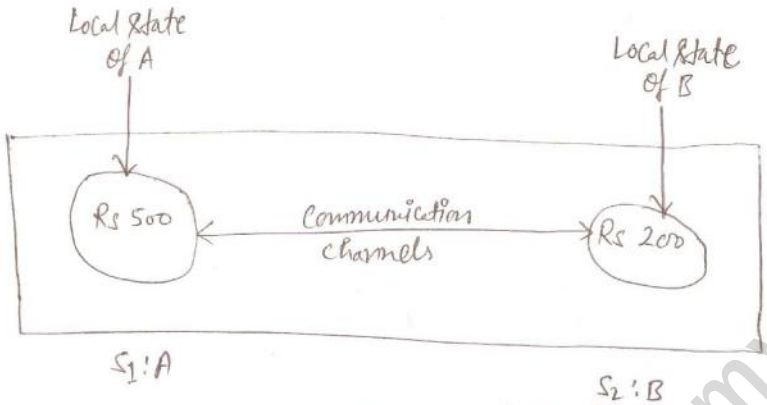
- Global Knowledge is not readly available, since it is impossible to collect uptodate information of global state of distributed system.

Because of (limitation) :–

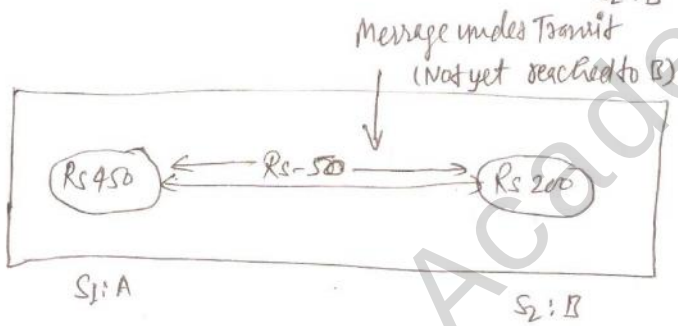
① Lack of Global clock (common clock)
② unprecedented Message delay
③ Non-existance of Physically shared Memory.

These limitation create difficulty in obtaining coherent global state.

Ex-




Case A-




Case B


Case 1- $S_1$ records its local state (Rs 450) just after debit (-50) & $S_2$ records its location (zero) before receiving.
If transmit message is not taken care off-

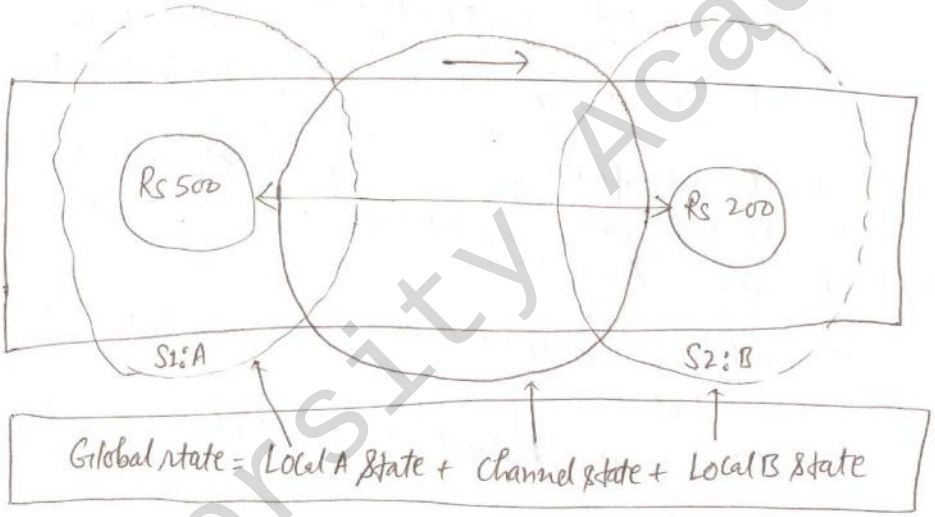
Global State = local state $S_1$ + local state $S_2$
= 450 + 200
= 650 ⇒ 50 Rs missing i.e. Incoherent system

Case 2:- $S_1$ records its state (Rs-500) before debit & $S_2$ records its state after credit hence there is surplus (Rs-50) which is also in-coherent.

From above two example, it is clear that communication channel cannot record its state by itself.

- Hence, sites must have to coordinate their state recording activities in order to record the channel state or coherent state of system.

Global state (GS) = Local state $S_1$ + Local state $S_2$ + channal state bet$^n$ $S_1$ & $S_2$



* Some types of H/w & S/w Resources that are distributed are

Given as →

① H/w Resources –

① CPU – Compute server, remote object server, worm programs, Other servers are file servers, Remote servers their cpu is shared resources.

② Memory – Cache server is shared (holds recently accessed web pages)

③ Disk – file server, virtual disk server, Vido on demand servers.

④ Screen – Net X windows system, (X-11) allow process in remote computers to update the content of windows.

⑤ printer – Networked printers accept print jobs from many computers, Managing them with a Queing system.

⑥ Network Capacity – packet transmission enables many simultaneous comm channels (stream of Data) to be transmitted in the same circuit.

② Data/S/w Resources

① web pages – web servers enable multiple clients to shared read-only page content.

② file – file server enables Multiple client to share read–write files.

③ object – Possibility for S/w objects are limitless.

④ database – Databases are intended to record the definitive state of related sets of data.

⑤ Newsgroup Content – (News on Internet)

⑥ Video/Audio Streams – server can store entire video on disk & deliver them at playback speed to Multiple client.

⑦ exclusive lock – a system level object provided by lock server, enabling several clients to coordinate their use of resources. (Such as printer that does not include a queing scheme)

## The impact of absence of global time or global clock :→

- It will be difficult to order event. that is temporal ordering of events. This will result in difficulty in design & development of distributed system.

ex→ An OS is responsible for scheduling process. A basic criterion used in scheduling is the temporal order in which requests to execute process arrives. Due to the absence of global clock it is difficult to reason about the temporal order of events in a distributed system.

- Hence algorithms for DS is more difficult to design & debug as compared to algorithms for centralized system. In the absence of a Global Clock makes it harder to collect up-to-date information on state of entire system.

## Lamports Logical Clock→

- An event changes the system state, which in turn influences the occurrence & outcome of future events. That is past events influence future events & this influences among casually related events referred as "casual affects"

So logic we define→

① Happened before relation (→) (casually related events of a→b)

a→b, means event "a happened before b".

- a & b are events in the same process & a occured before b".
- a is sending message & b is receipt of same message.
- If a→b & b→c ⇒ a→c (transitive)

⑪ Concurrent event→ (a||b) Two distinct event a & b are said to be Concurrent if a ↛ b & b ↛ a.

Logical clocks:

- Process $P_i$ has clock $C_i$.
- For an event a, $C_i(a)$ returns timestamp of event a.
- Timestamp monotonically increase.

Clock works with following conditions:

($C_1$) For any two event a & b in a Process $P_i$, if a occurs before b, then

$$C_i(a) < C_j(b)$$

($C_2$) If a is the event of sending a message m in process $P_i$ & b is the event of receiving the same message m at process $P_j$, then

$$C_i(a) < C_j(b)$$

The following implementation rules (IR) for the ~~clock~~ clock guarantee that the clocks satisfy the correctness condition $C_1$ & $C_2$.

[$IR_1$] clock $C_i$ is incremented bet$^n$ any two successive events in process $P_i$;

$$C_i = C_i + d \quad (d > 0)$$

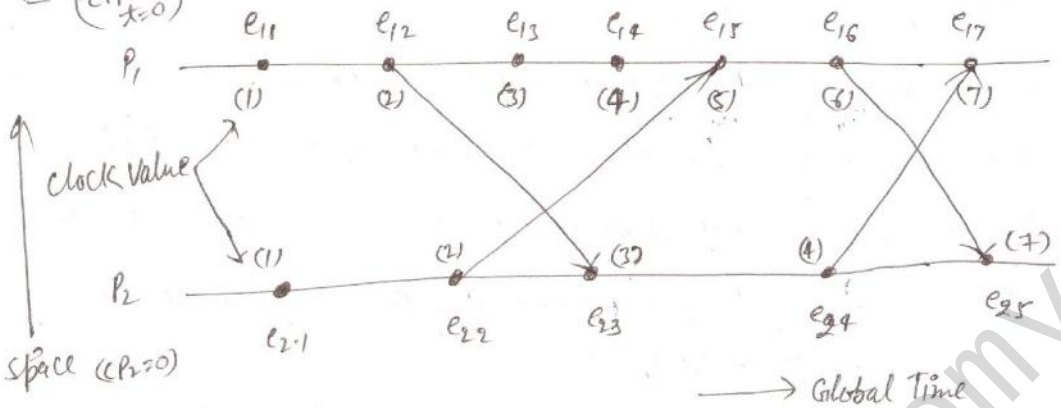
i.e. If a & b are two successive events in $P_i$ & $a \to b$ then

$$C_i(b) = C_i(a) + d$$

[$IR_2$] If event a is the sending of message m by Process $P_i$, then message m is assigned timestamp $t_m = C_i(a)$ (by $IR_1$). On Receiving the same message m by process $P_j$, $C_j$ is set to a value greater than equal to its present value & greater than $t_m$.

$$C_j = \max(C_j, t_m + d) \quad (d > 0)$$

Ex-1 ($CP_1=0$, $t=0$)




Let

$C_{P_1} \Rightarrow 0$ & $d=1$

$P_2 \Rightarrow 0$

$e_{1.4} \Rightarrow \max[C_{1.3}, [e_{2.1}]d] \Rightarrow \max[1, [2+3]]$

④ $e_{1.4} \rightarrow$ is the internal event in Process $P_1$ which causes $C_{P_1}$ to be increment to 1 due to $IR_1$.

Similarly, $e_{2.1}$ & $e_{2.2}$ are two events in $P_2$ resulting in

$C_{P_2}=2$ due to $(IR_1) \rightarrow C_i = C_i + d \ (d>0)$

$tm=0$, $d=1$

$e_{1.2} \rightarrow \max(1, 1+1) \Rightarrow 2$

$e_{1.3} \Rightarrow \max(2, 2+1) \Rightarrow 3$

$e_{2.2} \Rightarrow \max(2, 2+3) \Rightarrow 5$

$e_{1.6} \rightarrow P_1 \rightarrow$ increment $C_{P_1}$ to 6 due to $IR_1$.

$tm \Rightarrow 6$

$\max((4+1, 6+1))$

$C_i = \max[C_i, tm+d] \ (d>0)$ ($C_i+d$)

$e_{2.3} \Rightarrow \max(2+1, 2+1) \Rightarrow 3$

$e_{2.2} \Rightarrow (2+1, 4+1) \Rightarrow 5$

$e_{2.5} \Rightarrow \max(4+1, 6+1) \Rightarrow 7$ (tm)

$e_{1.7} \Rightarrow \max(4+1, 6+1) \Rightarrow 7$

due to ($IR_1$ / $IR_2$)

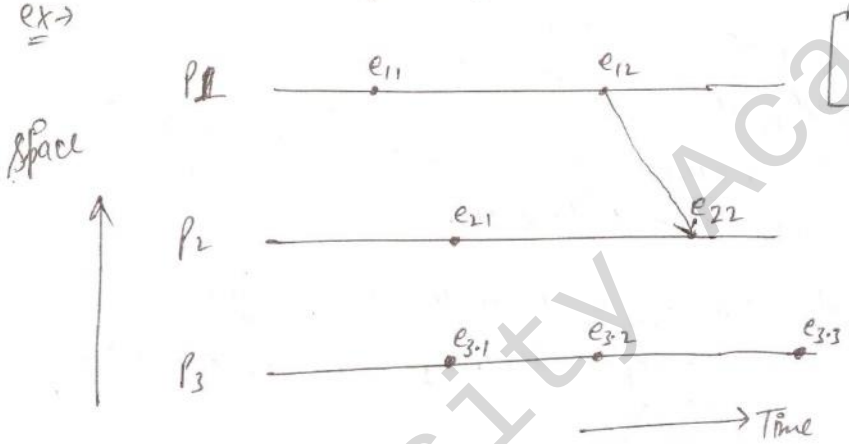
## Limitation of Lamport's Clock →

→ Lamport system of logical clocks, if $a \to b$ the $C(a) < C(b)$. However the reverse is not necessarily true if the events have occured in different process.

→ Thus if a & b are event at different processes & $C(a) < C(b)$, the $a \to b$ is no necessarily true; event a & b may be casually related or may not be casually related.

→ Thus Lamport's system of clocks is not powerful to capture such situations.

ex →



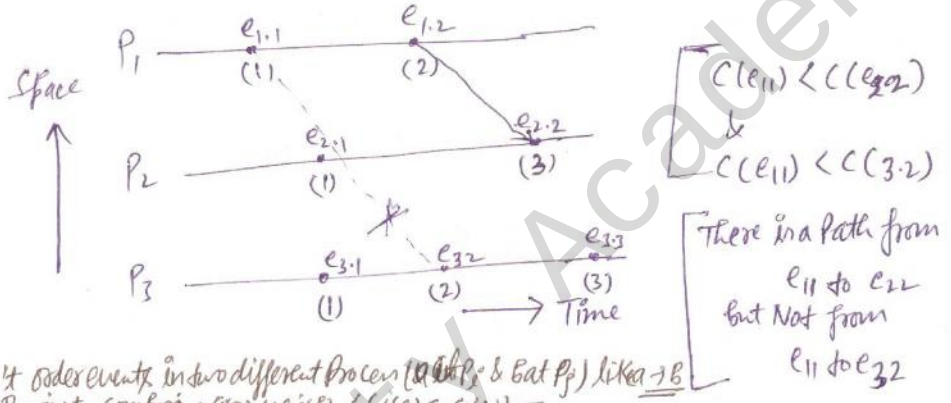

$$\begin{bmatrix} C(e_{1,1}) < C(e_{2,2}) \\ C(e_{1,1}) < C(e_{3,2}) \end{bmatrix}$$

We can see that event $e_{1,1}$ is casually related to event $e_{1,2}$ but Not $e_{3,2}$ Not even $e_{3,2}$, since a path exist from $e_{1,1}$ to $e_{2,2}$ but Not $e_{1,1}$ to $e_{3,2}$.

## Limitation of Lamport's Logical Clock :→

- Lamport logical clocks, if $a \to b$ then $C(a) < C(b)$. However the reverse of this is not necessarily true if the events have occurred in different Processes.
- That means, if a & b are events in different Processes & $C(a) < C(b)$, then $a \to b$ is note necessarily true; events a & b may be casually related or may not be casually related.

Ex →




We can't order events in two different Process (like $P_i$ & but $P_j$) like $a \to b$
By just comparing $C_i(a)$ & $C_j(b)$ ($C_i(a) < C_j(b)$)

- Because each clock does can independently advance due to the occurrence of local events in a process & lamport's logical clock system cannot distinguish bet$^n$ the advancement of clock due to local events, from those, due to exchange of message bet$^n$ process.
- using the timestamps assigned by lamport's clocks, we cannot reason about the casual relationships of the events occuring in different processes by just looking the timestamps of the events.

Vector clocks →

- Let n be the no of processes in a distributed system. each process $P_i$ is equipped with a clock $C_i$, which is an integer vector of length.
- the clock $C_i$ can be thought of a function that assigned a vector $C_i(a)$ to any event a. $C_i(a)$ is referred to as the timestamp of event a at $P_i$, $C_i[i]$, is the ith entry of $C_i$, corrospondence to $P_i$'s own logical time. $C_i(j)$, $j \neq i$ is $P_i$'s best guess of the logical time at $P_j$.

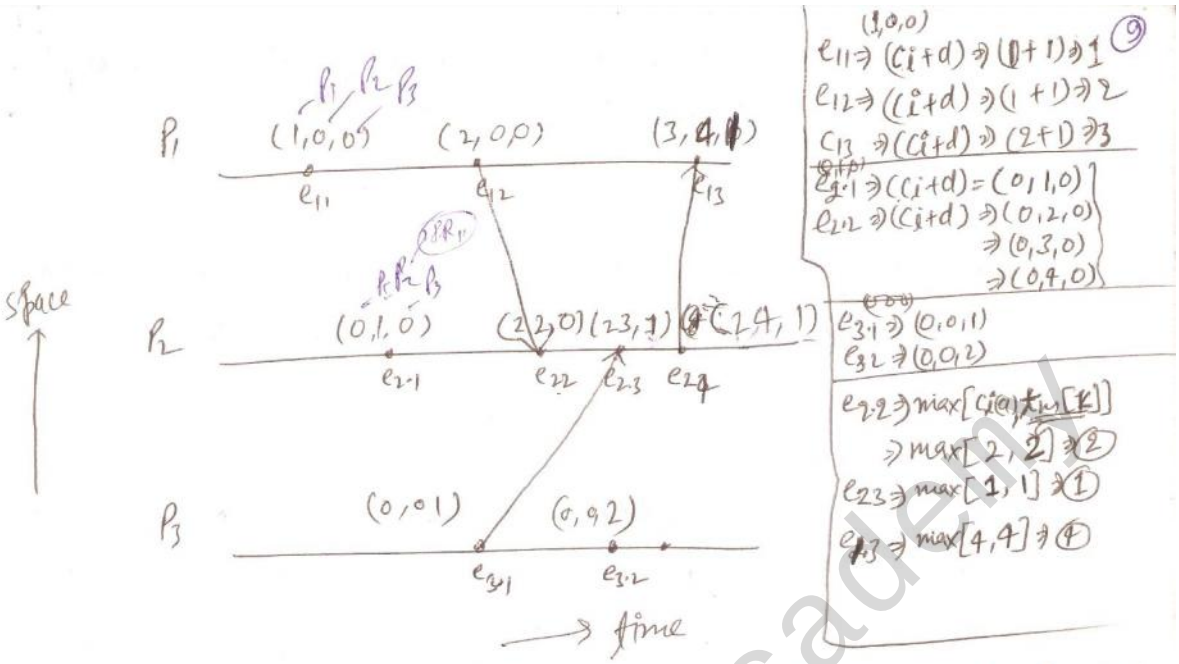
[$IR_1$] clock $C_i$ is incremented bet$^n$ any two successive events in process $P_i$-

$$C_i[i] = C_i[i] + d \qquad (d > 0) \qquad -(i)$$

[$IR_2$] if event a is sending of the message m by process $P_i$, then message m is assigned a vector timestamp $t_m = C_i(a)$; on receiving the same message m by process $P_j$, $C_j$ is updated as follows-

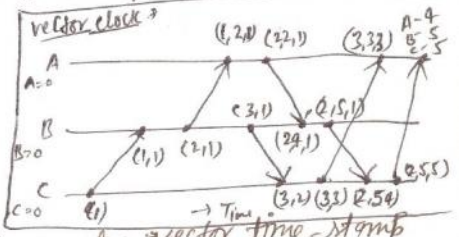
$$\forall k, \quad C_j[k] = \max(C_j[k], t_m[k]) \qquad -(ii)$$

Note- on the receipt of messages, a process learns about the more recent clock values of the rest of the processes in the system.




- In Rule IR1, we treat message send & receive by a process as events. In IR2 a message is assigned a timestamp after the sender process has incremented its clock due to IR1.

$$\boxed{\forall i, \forall j : C_i[i] \geq C_j[i]}$$




- Vector time stamps can be compared as follows. For any two vector time-stamp $t^a$ & $t^b$ of events a & b, respectively.

($\leq$ → Partial order
$\|$ → Not " " because of it is not transitive.)

- Equal — $t^a = t^b$ iff $\forall i$; $t^a[i] = t^b[i]$;
- Not Equal — $t^a \neq t^b$ iff $\exists i$; $t^a[i] \neq t^b[i]$;
- Less Than or Equal — $t^a \leq t^b$ iff $\forall i$; $t^a[i] \leq t^b[i]$;
- Not Less Than or Equal — $t^a \not\leq t^b$ iff $\exists i$; $t^a[i] \not\leq t^b[i]$;
- Less Than — $t^a < t^b$ iff $(t^a \leq t^b \wedge t^a \neq t^b)$;
- Not less Than — $t^a \not< t^b$ iff $\neg(t^a \leq t^b \wedge t^a \neq t^b)$;
- Concurrent — $t^a \| t^b$ iff $(t^a \not< t^b \wedge t^b \not< t^a)$;

- a & b is casually related if $t^a < t^b$, even if a & b occurred at different process.

- Thus vector clocks allows to order events & decides whether two events are casually related or not by simply looking at the timestamp of the events.

## Casual ordering of messages →

- The casual ordering of messages deals with the notation of maintaining the same casual relationship that holds among "message send" events with the corresponding "message receive" events ~~with the~~

- if Send ($M_1$) → Send ($M_2$) (where Send ($M_1$) is the event sending message $M_1$) then every recipient of both message $M_1$ & $M_2$ must receive $M_1$ before $M_2$.

- The casual ordering of message should not be confused with the casual ordering of events, which deals with the Notation of casual relationship among the events.

- Casual ordering of message is not automatically guaranteed.

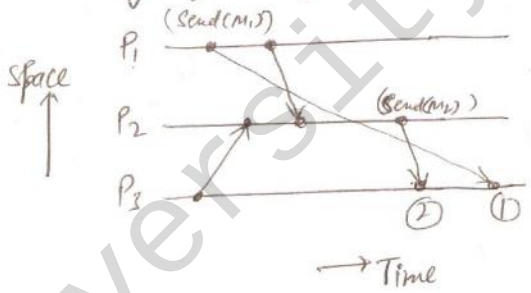



fig- example of voilation of casual ordering of message.

**Importance of Algorithm** → Algorithm for casual ordering of message is important in developing (1) distributed algorithm (2) Implementation of above distributed algorithm.

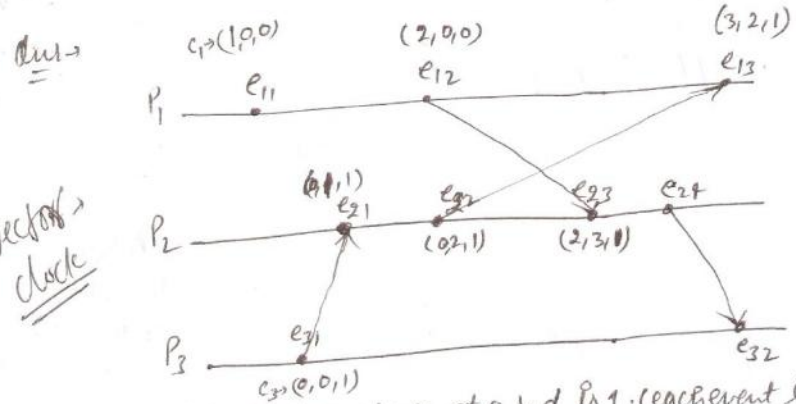
ex: For application such as (replicated database), it is important that every process incharge of updating a replica ~~not~~ receives the updates in the same order to maintain the consistency of database.

**Basic of algo**

- Algo is design to deliver a message to a process only if the message immediately ~~but~~ preceding it has been delivered to the process. otherwise the message is not delivered immediatly but is buffered until the message immediatly preceding it is delivered.

Aim →

Vector clock




Limitation (10)

If $(a \to b)$ then $C(a) < C(b)$

but $C(a) < C(b)$ $(a \to b)$ ?

- Assume that all clock starts at 0 & d is 1. (each event is incremented by clock 1)

At event $e_{12}$, $C_1(e_{12}) = 2$, event $e_{12}$ is sending of message to $P_2$.

- when $P_2$ receives the message $(e_{23})$, if clock $C_2 = 2$, the clock is reset to 3.

- Event $e_{24}$ is $P_2$'s sending a message to $P_3$, that message is received at $P_3$ at $e_{32}$. $C_3$ is 1 (as one event has passed). By rule $IR_2$, $C_3$ is reset to the max$^m$ of $C_2(e_{24}+1)$ & the current value of $C_3$, so $C_3$ becomes 5.

$$C_3(e_{31}) = 1 < 2 = C_1(e_{12})$$

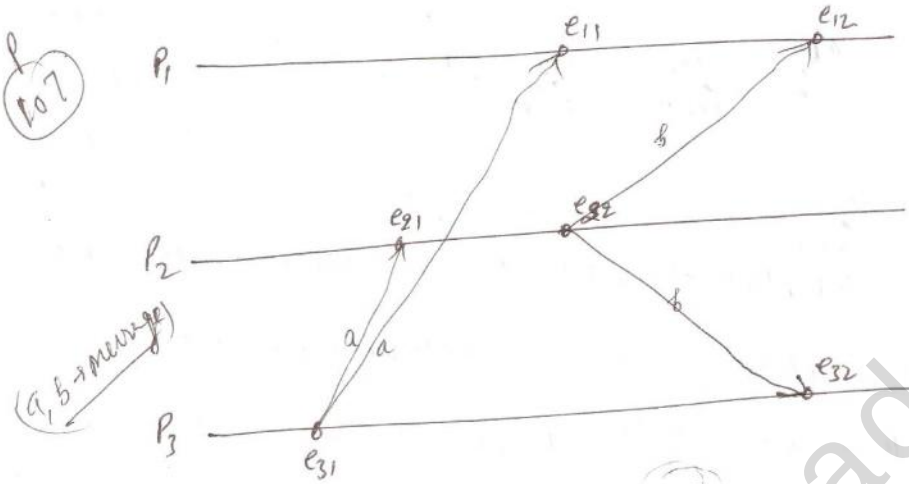
$e_{31}$ & $e_{21}$ is causally unrelated, $C_1(e_{11}) < C_3(e_{32})$

$e_{11} \to e_{32}$ ∴ Hence can not say one way to other.

So, The vector clock, each process keeps track of what is believes the other processes' internal clocks are. The goal is to provide the ordering upon events within the system.

BSS → Birman Schiper-Stephenson Protocol

Ref: [107]




- n → no. of procesn
- $P_i$ → i^th procesn
- $C_i$ → vector clock associated with $P_i$. $j^{th}$ element is $C_i[j]$ & contains $P_i$'s latest value for the current time in procesn $P_j$
- $t^m$ → vector time stamp for message m.

$e_{3,1}$ ⇒ $P_3$ send message a, $C_3 = \{0,0,1\}$; $t^a = \{0,0,1\}$.

$e_{2,1}$ ⇒ $P_2$ receive message a, As, $C_2 = \{0,0,0\}$, $C_2[3] = t^a[3] - 1 = 1-1 = 0$ &

$C_3(3) = 1$, $C_3(2) = 0$, $C_3(1) = 0$ ⟩ $C_2(0,0,0) = \begin{bmatrix} C_2(3) \Rightarrow 0 \\ C_2(1) \Rightarrow 0 \\ C_2(1) \Rightarrow 0 \end{bmatrix}$

$C_2[1] = t^a[1]$
$C_2[2] = t^a[2]$ ⟩ 0 So the message is accepted & $C_2$ is set to (0,0,1)

$e_{11}$ → $P_1$ receives message a, As $C_1 = \{0,0,0\}$, $C_1[3] = t^a[3] - 1 \Rightarrow 1-1 = 0$ &
$t^a = \{0,0,1\}$

$C_1[1] \Rightarrow t^a[1]$
$C_2[2] \Rightarrow t^a[2]$ ⟩ 0 So, the message is accepted & $C_1$ is set to (0,0,1)

$e_{22}$ → $P_2$ sends message b, $C_2 = \{0,1,1\}$, $t^b = \{0,1,1\}$

$e_{12}$ → $P_1$ receives message b, As $C_1 = \{0,0,1\}$, $C_1[2] = t^b[2] - 1 \Rightarrow 1-1 \Rightarrow 0$

$C_1[1] = t^b[1]$
$C_1[3] = t^b[2]$ ⟩ 0, So the message is accepted & $C_1$ is set to (0,1,1)

$e_{32}$ → $P_3$ receives message b, As $C_3 = (0,0,1]$, $C_3[2] = t^b[2] - 1 = 1-1 \Rightarrow 0$

$C_1[1] = t^b[1]$
$C_3[2] = t^b[2]$ ⟩ 0, So the message is accepted & $C_3$ is set to (0,1,1)

## Chandy-Lamport's Global State Recording Algorithm →

- It is a proposed model to capture a consistent global state. the algo use a Marker (a special message) to initiate the algorithm & the marker has no effect on the underlaying computation. The communication channels are assumed to be FIFO.

### Marker sending Rules for a Process P →

- P records its state
- For each outgoing channel C from P on which a marker has not been already sent, P sends a marker along C before P sendes further message along C.

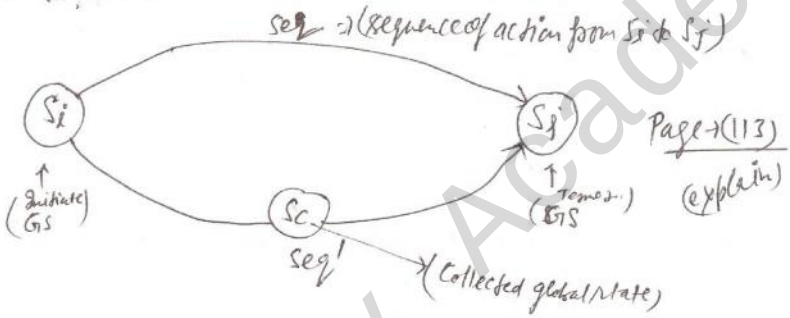
### Marker Receiving Rule for a process Q →

- On receipt of a marker along a channel C -

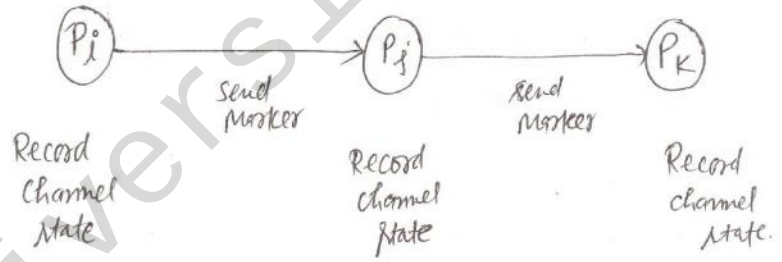
If Q has not recorded its state
then
begin
Record the state of C as an empty sequence.
Follow the "Marker sending Rule".
end
else
Record the state of C as the sequence of message received along C after Q's state was recorded & before Q received the Marker along C.

- FIFO channel condition + marker help in satisfying Consistency Condition.

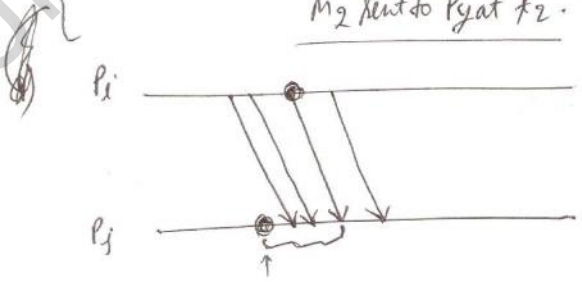
- Initiation of marker can be done by any process, with its own unique marker :→ <process id, sequence no>.
- Several processes can initiate state recording by sending markers. Concurrent sending of markers allowed.
- One possible way to collect global state - ~~all initiator of marker~~
  - all process send the recorded state information to the initiator of marker. Initiator process can sum up the global state.




ex




ex→ {Channel state $M_1$ sent to $P_x$ at $t_1$ ... / $M_2$ sent to $P_y$ at $t_2$ ...}



## Termination Detection :->

- A Distributed Computation generally consists of a set of cooperating process which communicate with each other by exchanging messages. In this case of Distribd Computation, it is important to know when the computation has terminated.
- Termination detection, infact, is an example of usage of the Coherent View (Consistent global state) of distributed system.

System Model -> A process may either be in an active state or idle state. Only active processes can send messages. An Active process may become idle at any time.

- An idle process can be come active on receiving a computation message.
- Computation message are those that are related to the underlaying computation being performed by the cooperating process.
- A computation is said to have terminated if and only if all the process are idle & there are no messages in transit. the message sent by the termination detection algorithm are known as the Control messages.

Basic Idea ->

- One of the cooperating process monitors the computation is called the Controlling agent.
- Initially all process are idle, the controlling agent's weight equals 1, & the weight of the rest of the process is zero. the computation starts when the controlling agent sends a computation message to one of the process.
- Any time a process sends a message, the process's weight is split bet" itself & the process receiving the message (the message carries the weight of the receiving process).
- The weight received along with a message is added to the weight of the process.
- Thus algo assigns a weight $W (0 < W \leq 1)$ to each active process (including agents) & to each message in transit.

- The weight assigned are such that at any time, they satisfy an invariant

$$\boxed{\Sigma w = 1}$$

- On receiving the computation, a process sends its weight to the controlling agent, which adds the received weight to its own weight.

- When the weight of the controlling agent is once again equal to 1, it concludes that the computation has terminated.

Notation →

✓ B (Dw) → computation message sent as a part of the computation & Dw is the weight assigned to it.

✓ C (Dw) → control message sent from the process to the controlling agent & Dw is the weight assigned to it.

Huang's Termination Detection algo.—

Rule (1) → The controlling agent or an active process having weight w may send a computation message to a process P by doing -

Derive $w_1$ & $w_2$ such that

$w_1 + w_2 = w$ $\quad w_1 > 0, \; w_2 > 0;$

$w = w_1$

Send B($w_2$) to P;

Rule (2) → On receipt of B(Dw), a process P having weight w does;

$w = w + Dw$;

If P is idle, P becomes active.

Rule (3) → An Active process having weight w may become idle at any time by doing -

Send C(w) to the controlling agent;

$\boxed{w = 0}$

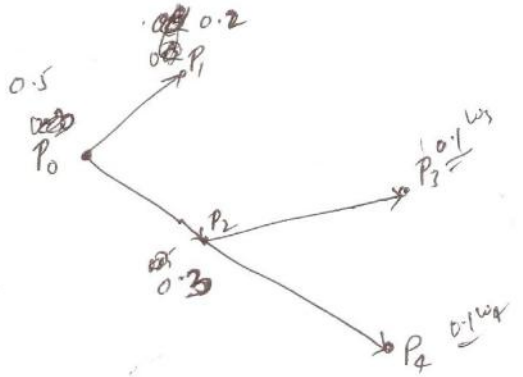
(The process becomes idle)

Rule (4) → on receiving C(Dw), the controlling agent having weight w takes the following actions -

$\boxed{w = w + Dw;}$

If $w = 1$, conclude that the termination has terminated.

Ex →




$(w_1 + w_2 + w_3 \Rightarrow 1)$

→ Process $P_0$, designated the controlling agent, with $w_0 \Rightarrow 1$. It asks $P_1$ & $P_2$ to do some computation. It sets $w_1$ to 0.2 & $w_2$ to 0.3 & $w_3 \Rightarrow 0.5$.

→ $P_2$ in turn asks $P_3$ & $P_4$ to do some computations. It sets $w_3$ to 0.1 & $w_4 \rightarrow 0.1$

→ When $P_3$ terminates, it sends $C(w_3) = C(0.1)$ to $P_2$, which changes $w_2$ to $(0.1 + 0.1) = 0.2$.

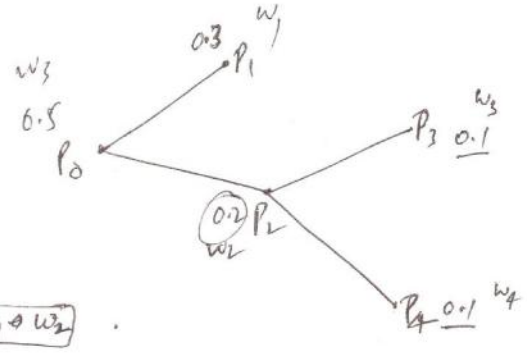
→ When $P_2$ terminates, it sends $C(w_2) = C(0.2)$ to $P_0$, which changes $w_0$ to $0.5 + 0.2 = 0.7$

→ When $P_4$ terminates, it sends $C(w_4) = (0.1)$ to $P_0$, which changes $w_0$ to $0.7 + 0.1 = 0.8$

→ When $P_1$ terminates, it sends $C(w_1) = C(0.2)$ to $P_0$, which changes $w_0$ to $0.8 + 0.2 = 1$

$P_0$ there upon conclude the computation is finished.

Total No of Messages Passed → 8 (One to start each Computation & one to Return the weight)




$w_1$ $w_2$ = $w_3$

$0.3 + 0.2 \Rightarrow w_3 \Rightarrow w$

$w \Rightarrow w_1$

Send $\Rightarrow$ $w_1 \Rightarrow w_2$

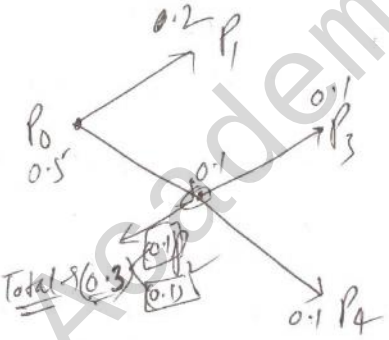
①

$P_3 \rightarrow (0.1 + 0.1) \Rightarrow (0.2)$

$P_4 \rightarrow (0.2 + 0.1) \Rightarrow (0.3)$

$P_2 \rightarrow (0.3 + 0.5) \Rightarrow 0.8$

$P_1 \rightarrow 0.8 + 0.3 \Rightarrow (1.1)$ ×

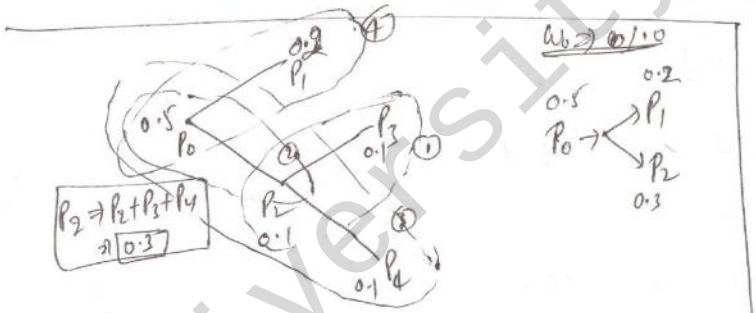



① $w_1, w_2$

$(w_1 + w_2 = w)$

Send $B(w_2)$ to $P_1$

$w \Rightarrow w_1$




$w_0 \Rightarrow 0.1 + 0$




$P_1 \rightarrow w_0 \Rightarrow 1$

$w_1 \rightarrow 0.2$

$w_2 \rightarrow 0.3$

$w_3 \rightarrow 0.5$

$w_1 + w_2 + w_3 = 1.0 = w_0$

0.2 0.3 0.5

→ When $P_3$ terminates it sends —

$c(w) \Rightarrow c(0.1)$ to controlling agent $P_2$ which changes $w_2$ to → $(0.1) + (0.1) = (0.2)$

→ When $P_2$ terminates it sends —

$c(w) = c(0.2)$ to " " $P_0$ "

" $w_0$ to → $(0.5 + 0.2) \Rightarrow (0.7)$

→ When $P_4$ " " " " $P_0$ which

$c(w) = c(0.1)$ to " "

" $w_0$ to → $(0.7) + (0.1) \Rightarrow 0.8$

→ When $P_1$ " $c(w) = c(0.2)$ to " " $P_0$ "

" $w_0$ to → $(0.8) + (0.2) \Rightarrow (1.0)$ Ans

Global state →

$GS = \{LS_1, LS_2, \ldots LS_n\}$

$rec(m_{ij}) \Rightarrow LS_i$ if $time(rec(m_{ij})) < time(LS_j)$

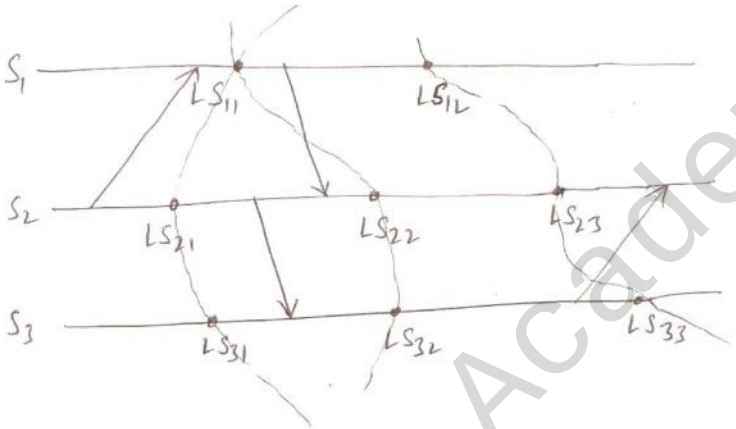
$send(m_{ij}) \Rightarrow LS_i$ if $time(send(m_{ij})) < time(LS_i)$

$transit(LS_i, LS_j) = \{m_{i,j} \mid send(m_{ij}) \in LS_i \;\&\; rec(m_{ij}) \notin LS_j\}$

$inconsistent(LS_i, LS_j) = \{m_{i,j} \mid send(m_{ij}) \notin LS_i \;\&\; rec(m_{ij}) \in LS_j\}$

$consistent(LS_i, LS_j) = \forall i, \forall j : 1 \le i, j \le n :: inconsistent(LS_i, LS_j) = \phi$

Ex →




① $\{LS_{12}, LS_{23}, LS_{33}\}$ is consistent. ✓

② $\{LS_{11}, LS_{22}, LS_{32}\}$ is inconsistent. ✓

③ $\{LS_{11}, LS_{21}, LS_{31}\}$ is strongly consistent. ✓

→ In A consistent GS, for every received message a corresponding send event is recorded in the Global state.

→ In an inconsistent global state, there is at least one message whose receive event is recorded but its send event is not recorded in the Global state.

→ In strongly consistent if it is consistent & transitless.
(Not only the send events of all the recorded received events are recorded, but the received events of all the recorded send events are also recorded.

SEA Protocol →

Protocol →



$P_i$ sends a message to $P_j$ →

① $P_i$ sends message m, timestamped $t^m$, & $V_i$, to process $P_j$.

② $P_i$ sets $V_i[i] = t^m$.

$P_j$ receives a message from $P_i$ →

① when $P_j$, $j \neq i$, receives m, it delays the message's delivery if both,

(a) $V^m[j]$ is set; &

(b) $V^m[j] < t^j$

② when the message is delivered to $P_j$, update all set elements of $V_j$ with the corresponding elements of $V^m$, except for $V_j[j]$, as follows.

(a) If $V_j[k]$ & $V^m[k]$ are uninitialized, do nothing.

(b) If $V_j[k]$ is uninitialized & $V^m[k]$ is initialized, set

$$V_j[k] = V^m[k]$$

(c) If both $V_j[k]$ & $V^m[k]$ are initialize, set

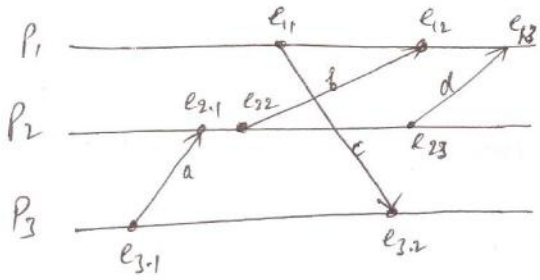
$$V_j[k][k'] = \max[V_j[k][k'], V^m[k][k']]$$

for all $k' = 1 \cdots n$.

③ update $P_j$'s vector clock.

④ Check Buffered messages to see if any one can be delivered.

Ex:-




$e_{3,1}$ → $P_3$ sends message a to $P_2$.

$C_3 = \{0, 0, 1\}$
$t^a = \{0, 0, 1\}$
$V^a = \{?, ?, ?\}$
$V_3 = \{?, (0, 0, 1), ?\}$

$e_{2,1}$ → $P_2$ Receive message a from $P_3$. As $V^a[2]$ is uninitialized, the message is accepted
$V_2$ is set to $\{?, ?, ?\}$ & $C_2$ is set to $\{0, 0, 1\}$.

$e_{2,2}$ → $P_2$ sends message b to $P_1$ $C_2 = \{0, 1, 1\}$
$t^b = \{0, 1, 1\}$
$V^b = \{?, ?, ?\}$
$V_2 = \{(0, 1, 1), ?, ?\}$

$e_{1,1}$ → $P_1$ sends message c to $P_3$.

$C_1 = \{1, 0, 0\}$
$t^c = \{1, 0, 0\}$
$V^c = \{?, ?, ?\}$
$V_1 = \{?, ?, \{1, 0, 0\}\}$

$e_{1,2}$ → $P_1$ receives message b from $P_2$ –

As, $V^b[1]$ is uninitialized the message is accepted.
$V_1$ is set to $\{?, ?, ?\}$
$C_1$ is set to $\{1, 1, 1\}$

$e_{3,2}$ → $P_3$ receives message c from $P_1$.

As, $V^c[3]$ is uninitialized the message is accepted.

$V_3$ is set to $\{?,?,?\}$

$C_{\cdot\cdot}$ " " $\{1,0,1\}$.

$e_{23}$ → $P_2$ sends message d to $P_1$

$C_2 = \{0,2,1\}$

$t^d = \{0,2,1\}$

$V^d = \{\{0,1,1\}, ?, ?\}$

$V_2 = \{(0,2,1), ?, (0,0,1)\}$

$e_{13}$ → $P_1$ receives message d from $P_2$.

As, $V^d[1] < C_1[1]$ so the message is accepted,

$V_1$ is set to $\{(0,1,1), ?, ?\}$ & $C_1$ is set to $\{1,2,1\}$

## Schiper – Eggli – Sandoz Protocol →

Data structure & Notation :→

- Each process P maintains a vector denoted by V_P of size (N-1), where N is the No of processes in the system.

- An element of V_P is an ordered pair (P', t) where P' is the ID of the Destination process of a message & t is a vector timestamp.

$t_M$ = logical time at the sending of message M.

$t_{P_i}$ = Present/current logical time at process $P_i$.

Protocol →

Sending of message M from Process $P_1$ to Process $P_2$ →

- Send message M (timestamp $t_M$) along with $V\_P_1$ to process $P_2$.
- Insert Pair $(P_2, t_M)$ into $V\_P_1$. If $V\_P_1$ contains a pair $(P_2, t)$, it simply gets over written by new Pair $(P_2, t_M)$. Note that the Pair $(P_2, t_M)$ was not sent to $P_2$. Any future message carrying $(P_2, t_M)$ pair, can't be delivered to $P_2$ until $t_M < t_{P_2}$.

Arrival of Message M at Process $P_2$ →

- If V_M (the vector accompanying message M) does not contain any pair $(P_2, t)$ then the Message can be delivered.

else (* A pair $(P_2, t)$ exists in V_M *)

If ( $t \not< t_{P_2}$ then )

the Message can't be delivered (* it is buffered for later delivery)

else

the message can be delivered.

If Message M can be delivered at Process $P_2$, then the following three actions are taken-

① Merge V_M accompanying M with $V_{-P_2}$ in the following Manner.

* If $(\exists (P,t) \in V\_M$, such that $P \neq P_2) \& (\forall (P',t) \in V_{-P_2}, P' \neq P)$. then Insert $(P,t)$ in $V_{-P_2}$.

UNIT-2 Mutual Exclusion ①

- In the problem of mutual exclusion, prevents concurrent access to a shared resources by several uncoordinated user-requests is serialized to secure the integrity of the shared resources.

- It requires that the actions performed by a user on a shared memory must be atomic. that is if several user concurrently access a shared resources then action performed by a user, as far as the other users are concerned, must be instantaneous & ~~indivisible~~ indivisible such that the net effect on the shared resources is the same as if user actions were executed serially, as opposed to in an interleaved manner.

- The Problem of ME frequently arises in distributed systems whenever concurrent access to shared resources by several sites is involved. For correctness ~~access to shared resources by server~~ it is necessary that the shared resources be accessed by a single site (or process) at a time.

ex→ Directory Mgt, where an update to a directory must be done automatically because if update & reads to a directory proceed ~~to a directory must be done automatically because~~ concurrently, reads may obtain inconsistent information. If an entry contains several fields, a read oper$^{n}$ may read some fields before the update & some after the update.

Mutual exclusion in Single Computer System Vs Distributed Systems

- The Problem of ~~or~~ ME in single-computer system, where shared memory exists.
- In single-computer systems, the status of a shared resources & the status of users is readily available in the shared memory, & sol$^{n}$ to the mutual exclusion problem can be easily implemented using shared variable (e.g. Semaphores).

- However in DS, Both the shared resources & the users may be distributed & Shared Memory does not exist. (Approach based on shared variable are no applicable to DSs & approach based on Message Passing Must be used).

- The Problem of ME becomes much more complex in DS (as Compared to Single-computer systems) because of the lack of Both shared memory & a common physical clock & because of unpredictable Message delay

The classification of ME Algorithms:

- The tend to differ in the communication topology (e.g tree, ring, etc) & in the amount of info maintained by each site about other sites.

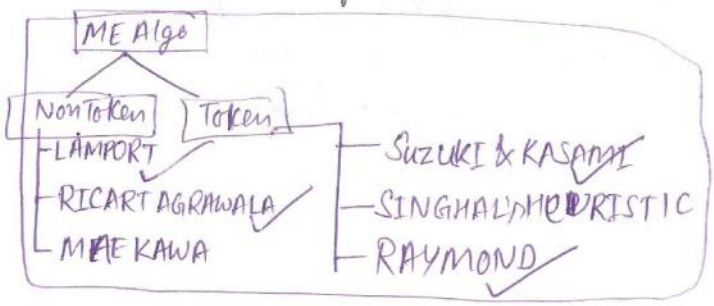
- Theses algorithms can be grouped into 2 classes.

(uses message to arrive at ME)

① Non Token based - these algo require two or more successive rounds of message exchanges among the sites. these algorithms are assertion based because a site can enter its critical section (CS) when an assertion defined on its local variables becomes true. ME enforced because the assertion becomes true only at one site at any given time.

(uses Token Privileged message to arrive at ME)

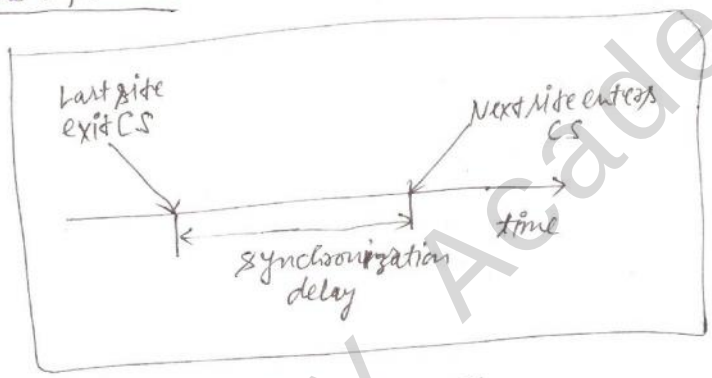
② Token based - In these algorithms, a unique token (also known as the Privilege message) is shared among sites. A site is allow to enter its CS if it possesses the token & it continues to hold to token until the execution of the CS is over.



## Requirements of ME Algo:

- At most one process in critical section (Safety) & (fault tolerance)
- if more than one requesting process, someone enters (liveness) & (freedom from deadlock)
- a requesting process enters within a finite time (no starvation)
- requests are granted in order (fairness)

## How to measure the Performance:

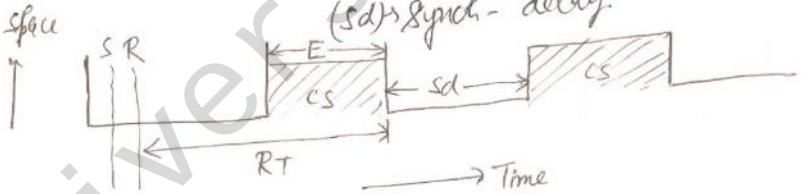



System throughput = 1/(sd + E)

(E) → avg. CS execution time

(sd) → Synch. delay

#




S → CS request arrives.
R → CS request message is sent out.
sd → Synchronous delay (time bet^n two CS)

Performance matrix for ME algo can be easily depicted by time diagram as —

① Response time (RT) → It is the time bet^n R & end CS. (time bet^n req^t message is sent out & completion of critical section. For small RT the performance of ME algo will be high.

② Synchronous delay (sd) → It is time bet^n two ~~consecutive~~ Consecutive CS. That is time bet^n END of CS & Begin of CS. In this period message are exchanged to arrive at ME decision.

Note - for small 'sd' Performance of ME algo will be high.

③ No of Message Per CS →

- Both two CS there are message exchange as discussed above, As no of message exchange reduces, the performance will improve.

(Note No of message per CS must be less to improve the performance)

④ System throughput → It is derived from 'Sd' & 'E' as

$$\text{System throughput} = \frac{1}{Sd+E}$$

- (Sd+E) time unit only one CS request is executing, so throughput is $[1/(Sd+E)]$. this can also be called as average time for each CS invocation or rate at which CS request Complete.

# SIMPLE Sol^n To Distributed ME →

- A site, called the control site, is assigned the task of granting permission for the CS execution. To request the CS, a site sends a Request message to the control site.
- ~~To request~~ The control site queues up the requests for the CS & grants them permission, one by one.
- This method to achieves mutual exclusion in distributed systems requires only three message per CS execution.

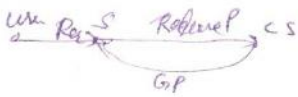

Central site has several drawback—

① Single Point of failure, the control site.

② The control site is likely to swamped (flood) with extra work.

③ Synchronous delay of algorithm is 2T, because a site should first release permission to the control site & then control site should grant permission to the next site to execute CS.

Throughput → $\frac{1}{(2T+E)}$ (If we reduce the Synchronous delay the throughput will be Max^m.)



## NON TOKEN BASED ALGORITHMS →

- In non-token based ME algo, a site communicates with a set of other sites to arbitrate who should execute the CS next.
- For a site $S_i$, request set $R_i$ contains ids of all those sites from which the site $S_i$, must acquire permission before entering the CS.

### 1) LAMPORT'S ALGORITHM →

- LAMPORT was first to give a Distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme.
- In LAMPORT's algorithm, $\forall i : 1 \le i \le N :: R_i = \{S_1, S_2, S_3, \cdots S_N\}$.
- Every Site $S_i$ keeps a queue, request_queue$_i$, which contains mutual exclusion requests ordered by their timestamps.
- This Algorithm requires messages to be delivered in the FIFO order bet$^n$ every pair of sites.

**Algo -**

**Requesting the CS →**

① When a Site $S_i$ wants to enter the CS, it sends a Request$(ts_i, i)$ message to all the sites in its request set $R_i$ & Places the request on request_queue$_i$. $(ts_i, i)$ → timestamp of the request.

② When a Site $S_j$ ~~request~~ receives the Request$(ts_i, i)$ message from site $S_i$, it returns a timestamped REPLY message to $S_i$, & Places site $S_i$'s request on request_queue$_j$.

## Executing the CS →

- Site $S_i$ enters the CS when the two following conditions hold →

[$L_1$] - $S_i$ has received a message with timestamp larger than $(ts_i, i)$ from all other sites.

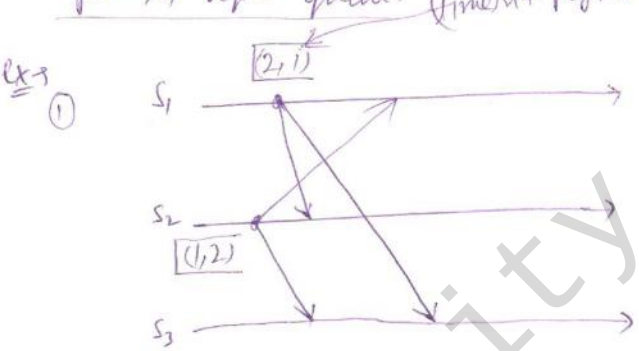
[$L_2$] - $S_i$'s request is at the top of request_queue.

## Releasing the CS →

③ Site $S_i$, upon exiting the CS, removes its request from the top of its request queue & sends a timestamped RELEASE message to all the sites in its request set.

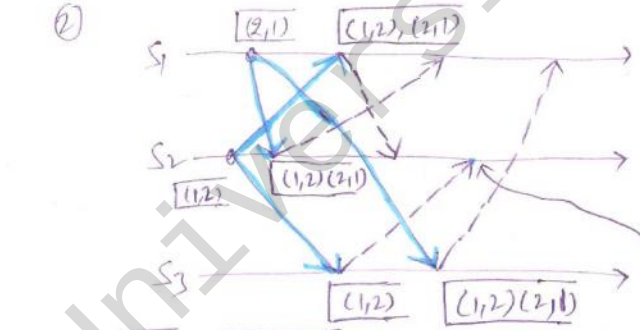
④ When a site $S_j$ receives a Release message from site $S_i$, it removes $S_i$'s request from its request queue. (timestamp of Rest)

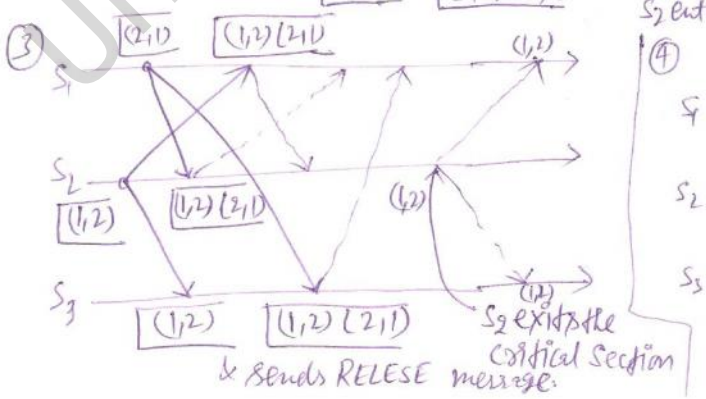
Ex →




| $(ts_i, i)$ | |
|---|---|
| 2, 1 | → ($S_1$) |
| 1, 2 | → ($S_2$) |

fig- Site $S_1$ & $S_2$ are making requests for the CS.




$S_2$ exits the Critical Section & Sends RELESE message.

$S_2$ enters the CS

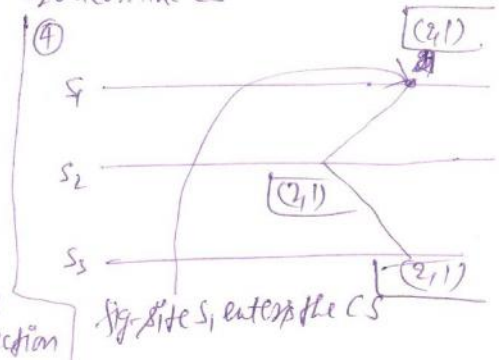



fig- Site $S_1$ enters the CS

## The Ricart-Agrawal Algorithm :→

- Ricart Agrawal is an optimization of Lamport's algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages.
- Here, $\forall i : 1 \le i \le N :: R_i = \{S_1, S_i \cdots S_N\}$.

### Algo→

Requesting the CS→

① When a site $S_i$ wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.

② When site $S_j$ receives a REQUEST message from site $S_i$, it sends a REPLY message to site $S_i$, if site $S_j$ is neither requesting nor executing the CS or if site $S_j$ is requesting & $S_i$'s request's timestamp is smaller than site $S_j$'s own request's timestamp. The request is deferred otherwise→
(Put of the later time)

Executing the critical Section→

③ Site $S_i$ enters the CS after it has received REPLY message from all the sites in its request set.

Releasing the critical section→

④ When site $S_i$, exit the CS, it sends REPLY message to all the deferred requests.

Ex→

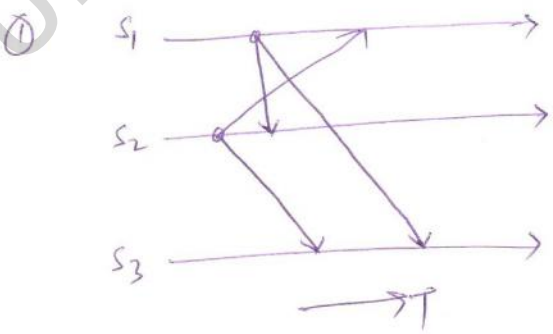
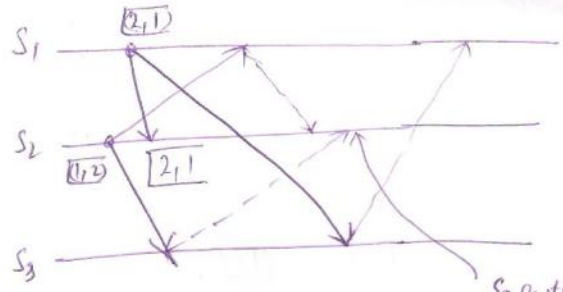


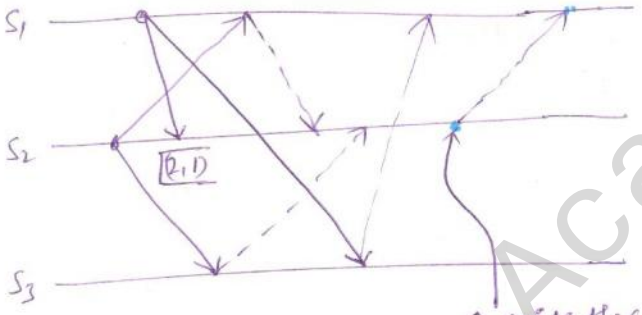

Fig- $S_1$ & $S_2$ are making req. for the CS.

(2)




$S_2$ enters the CS.

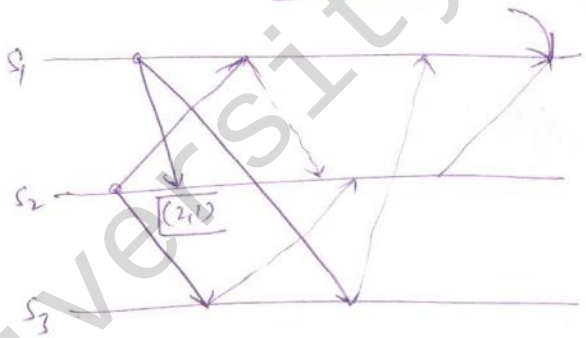
(3)




$S_2$ exits the CS & sends a REPLY message to $S_1$.

(4)

$S_1$ enters the CS



## Token Based Algo →

- A unique token is shared among all the sites. A site is allowed to enter its CS if it possesses the token. Depending upon the way a site carries out its search for the token, there are numerous token based algo.

① Token-Based algo use sequence no instead of timestamps.
(a sites increments its seq no everytime it makes a req^n for the token)

② (A Primary func^n of the sequence no is to distinguish betⁿ old & current requests.)

② → (137) Correctness Proff of token-based algo.

## SUZUKI-KASAMI'S BROADCAST ALGORITHM :→

Req^ing CS

① If the requesting site $S_i$ does not have the token, then it increments its sequence no, $RN_i[i]$, & sends a REQUEST($i$, sn) message to all other sites.
(sn is the updated value of $RN_i[i]$)

② When a site $S_j$ receives this message, it sets $RN_j[i]$ to max($RN_j[i]$, sn). If $S_j$ has the idle token, then it sends the token to $S_i$, if

$$RN_j[i] = LN[i] + 1$$

$LN[j]$ → the sequence No of the request that sites $S_j$ executed most recently.

Executing the CS :→

③ Site $S_i$ executes the CS when it received the token.

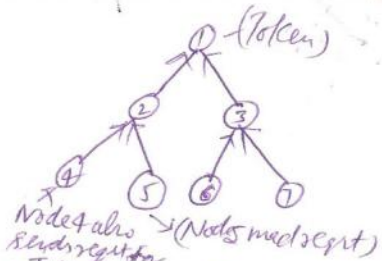
Releasing the CS → After Finishing the CS, site $S_i$ makes the following actions–

④ It sets $LN[i]$ element of the token array equal to $RN_i[i]$.

⑤ For every site $S_j$ whose ID is not in the token queue it appends its ID to the token Queue if $RN_i[j] = LN[j] + 1$

⑥ If token queue is nonempty after the above update, then it deletes the top site ID from the queue & sends the token to the site indicated by the ID.

Suzuki - Kasami ex>




Step 1 : $S_1$ has token, $S_3$ is in Queue>

| Site | Seq Vector RN | Token Vector LN | Token Queue |
|---|---|---|---|
| $S_1$ | 10, 15, 9 | 10, 15, 8 | 3 |
| $S_2$ | 10, 16, 9 | 0 | |
| $S_3$ | 10, 15, 9 | | |

Step 2 : $S_3$ gets token, $S_2$ in Queue →

| | | | |
|---|---|---|---|
| $S_1$ | 10, 16, 9 | | |
| $S_2$ | 10, 16, 9 | | |
| $S_3$ | 10 16 9 | 10 15 9 | 2 |

Step 3 → $S_3$ gets token Queue empty>

| | | | |
|---|---|---|---|
| $S_1$ | 10, 16, 9 | | |
| $S_2$ | 10, 16, 9 | 10, 16, 9 | <empty> |
| $S_3$ | 10, 16, 9 | | |

## Suzuki & Kasami's →

Token →

- Queue (FIFO) Q of requesting Process.
- LN[1...n] :→ Sequence No of request that j executed most recently

The requested message →

- REQUEST (i, K) - request message from node i for its K^th critical section.

Other Data structures →

- RN_i[1..n] for each node i, where RN_i[j] is the largest sequence No received so far by i in a Req^st message from j.

To Request CS →

→ If i does not have token, increment RN_i[i] & send ~~REQUEST~~ REQUEST (i, RN_i[i]) to all nodes.

→ If i has token i already, enter CS if the token is idle (No pending req^st), else follow rule to release CS.

On Receiving REQUEST (i, sn) at j :→

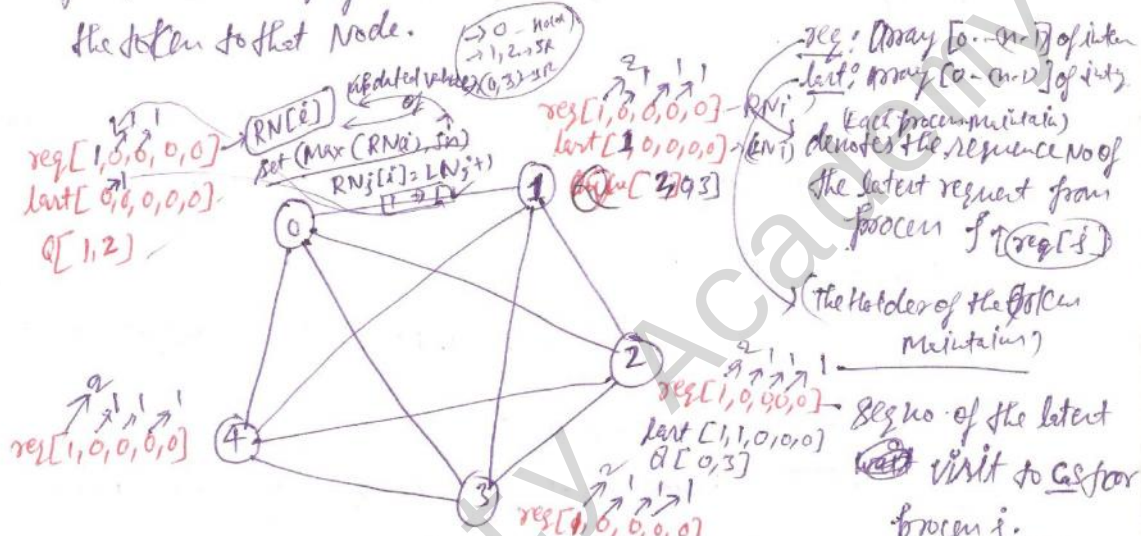
- Set RN_j[i] = max[RN_j[i], sn]
- If j has the token & the token is idle, then send it to i if RN_j[i] = LN[i] + 1. If token is not idle, follow rule to Release CS.

To enter the CS →

- Enter the CS if token is present.

To Relase the CS →

- Set LN[i] = RNi[i]
- For every node j which is not in Q (in token), add node j to Q if RNi[j] = LN[j]+1
- If Q is non empty after the above, delete first node from Q & send the token to that Node.




- When a process i receives a request (K, num) from process K, it sets req[K] to max (req[K], num).

The holder of the Token →

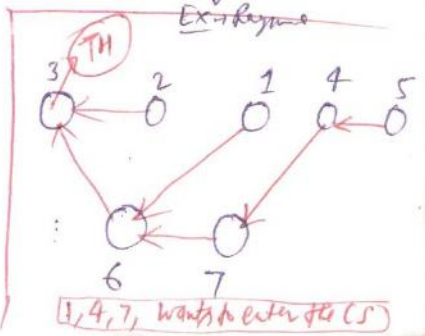
- Complete CS
- sets Last[i] := its own num
- Update Q by retaining each process k only if :-

$$1 + last[K] = req[K]$$ → (guarantee the fairness of req's)

- Sends the token to the head of Q, along with the array last & & the tail Q.

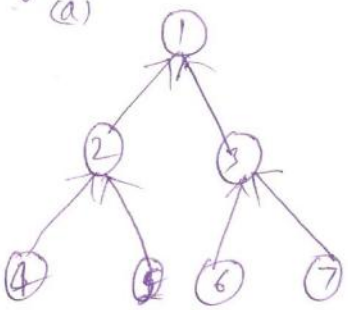
$$token = (Q, last)$$

Initially - req[i] = last[j] = 0



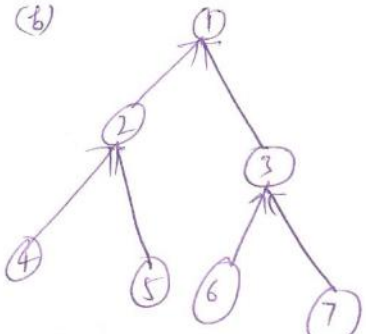
## Raymond's Tree Based Algo →

(a)



① Token is at Node 1
node 5 made a request

(b)

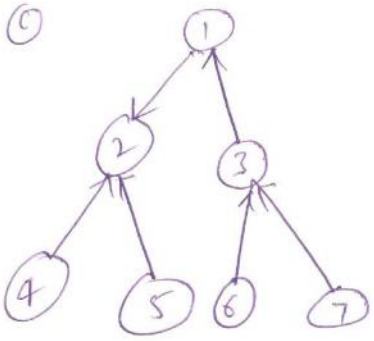


② Node 2 receives the Reqst
it sends the request to Node 1

(c)

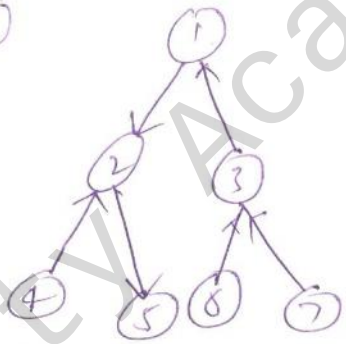
③ Node 4 also sends a requests, node 2 receives it.

(c)



④ Token is at node 2 now
node 2 becomes the root

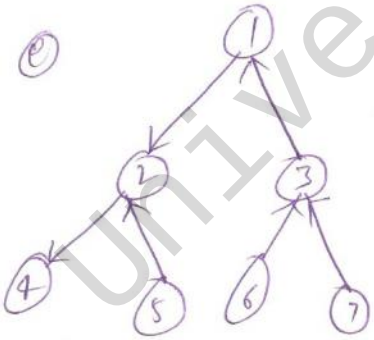
(d)



⑤ node 5 gets the token, it enters CS

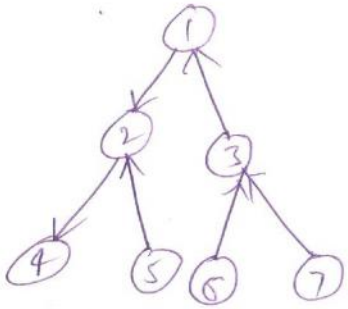
⑥ node 2 sends a request to node 5

(e)



⑦ Node 5 sends the token
to node 2

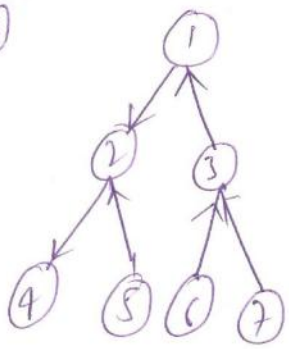
(f)



⑧ Node 4 gets the token, it enters CS

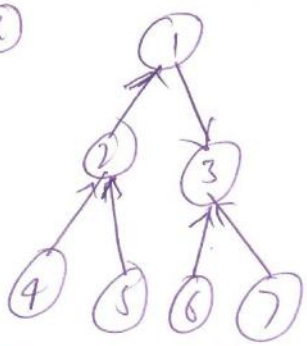
⑨ Node 3 sends a request

(9)




(10) the regst from node 2 comes to node 4

(8)



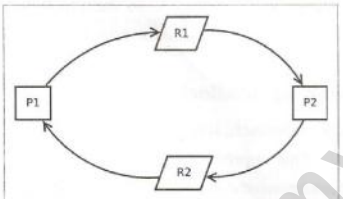

(11) Node 3 gets the token, & becomes the root.

UNIT-2



# Deadlock

A **deadlock** is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.



Both processes need both resources. P1 requires additional resource R1, P2 requires additional resource R2; neither process can continue.

In an operating system, a deadlock is a situation which occurs when a process enters a waiting state because a resource requested by it is being held by another waiting process, which in turn is waiting for another resource. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.[1]

Deadlock is a common problem in multiprocessing systems, parallel computing and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization.[2]

In telecommunication systems, deadlocks occur mainly due to lost or corrupt signals instead of resource contention.[3]

## Examples

Deadlock situation can be compared to the classic "chicken or egg" problem.[4] It can be also considered a paradoxical "Catch-22" situation.[5] A real world analogical example would be an illogical statute passed by the Kansas legislature in the early 20th century, which stated:[1][6]

> "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

A simple computer-based example is as follows. Suppose a computer has three CD drives and three processes. Each of the three processes holds one of the drives. If each process now requests another drive, the three processes will be in a deadlock. Each process will be waiting for the "CD drive released" event, which can be only caused by one of the other waiting processes. Thus, it results in a circular chain.

## Necessary conditions

A deadlock situation can arise if and only if all of the following conditions hold simultaneously in a system:[1]

1. **Mutual Exclusion:** At least one resource must be non-shareable.[1] Only one process can use the resource at any given instant of time.
2. **Hold and Wait** or **Resource Holding:** A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. **No Preemption:** The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
4. **Circular Wait:** A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, $P = \{P_1, P_2, ..., P_N\}$, such that $P_1$ is waiting for a resource held by $P_2$, $P_2$ is waiting for a resource held by $P_3$ and so on till $P_N$ is waiting for a resource held by $P_1$.[1][7]

These four conditions are known as the **Coffman conditions** from their first description in a 1971 article by Edward G. Coffman, Jr.[7] Unfulfillment of any of these conditions is enough to preclude a deadlock from occurring.

## Deadlock handling

Most current operating systems cannot prevent a deadlock from occurring.[1] When a deadlock occurs, different operating systems respond to them in different non-standard manners. Most approaches work by preventing one of the four Coffman conditions from occurring, especially the fourth one.[8] Major approaches are as follows.

### Ignoring deadlock

In this approach, it is assumed that a deadlock will never occur. This is also an application of the Ostrich algorithm.[8][9] This approach was initially used by MINIX and UNIX.[7] This is used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable.

### Detection

Under deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected. An algorithm is employed that tracks resource allocation and process states, it rolls back and restarts one or more of the processes in order to remove the detected deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler of the operating system.[9]

Deadlock detection techniques include, but are not limited to, *model checking*. This approach constructs a finite state-model on which it performs a progress analysis and finds all possible terminal sets in the model. These then each represent a deadlock.

After a deadlock is detected, it can be corrected by using one of the following methods:

1. **Process Termination:** One or more process involved in the deadlock may be aborted. We can choose to abort all processes involved in the deadlock. This ensures that deadlock is resolved with certainty and speed. But the expense is high as partial computations will be lost. Or, we can choose to abort one process at a time until the deadlock is resolved. This approach has high overheads because after each abortion an algorithm must determine whether the system is still in deadlock. Several factors must be considered while choosing a candidate for termination, such as priority and age of the process.
2. **Resource Preemption:** Resources allocated to various processes may be successively preempted and allocated to other processes until the deadlock is broken.

### Prevention

Deadlock prevention works by preventing one of the four Coffman conditions from occurring.

- Removing the **mutual exclusion** condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
- The **hold and wait** or **resource holding** conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none. Thus, first they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in resource starvation.[1] (These algorithms, such as serializing tokens, are known as the *all-or-none algorithms*.)
- The **no preemption** condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a *priority* algorithm. Preemption of a "locked out"

resource generally implies a rollback, and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

- The final condition is the **circular wait** condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration.[1] Dijkstra's solution can also be used.

## Avoidance

Deadlock can be avoided if certain information about processes are available to the operating system before allocation of resources, such as which resources a process will consume in its lifetime. For every resource request, the system sees whether granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants requests that will lead to *safe* states.[1] In order for the system to be able to determine whether the next state will be safe or unsafe, it must know in advance at any time:

- resources currently available
- resources currently allocated to each process
- resources that will be required and released by these processes in the *future*

It is possible for a process to be in an unsafe state but for this not to result in a deadlock. The notion of safe/unsafe states only refers to the *ability* of the system to enter a deadlock state or not. For example, if a process requests A which would result in an unsafe state, but releases B which would prevent circular wait, then the state is unsafe but the system is not in deadlock.

One known algorithm that is used for deadlock avoidance is the Banker's algorithm, which requires resource usage limit to be known in advance.[1] However, for many systems it is impossible to know in advance what every process will request. This means that deadlock avoidance is often impossible.

Two other algorithms are Wait/Die and Wound/Wait, each of which uses a symmetry-breaking technique. In both these algorithms there exists an older process (O) and a younger process (Y). Process age can be determined by a timestamp at process creation time. Smaller timestamps are older processes, while larger timestamps represent younger processes.

| | Wait/Die | Wound/Wait |
|---|---|---|
| O needs a resource held by Y | O waits | Y dies |
| Y needs a resource held by O | Y dies | Y waits |

## Livelock

A **livelock** is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.[10] Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.[11]

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.[12]

## Distributed deadlock

Distributed deadlocks can occur in distributed systems when distributed transactions or concurrency control is being used. Distributed deadlocks can be detected either by constructing a global wait-for graph, from local wait-for graphs at a deadlock detector or by a distributed algorithm like edge chasing.

In a commitment ordering-based distributed environment (including the strong strict two-phase locking (SS2PL, or rigorous) special case) distributed deadlocks are resolved automatically by the atomic commitment protocol (like a two-phase commit (2PC)), and no global wait-for graph or other resolution mechanism is needed. Similar automatic global deadlock resolution occurs also in environments that employ 2PL that is not SS2PL (and typically not CO; see *Deadlocks in 2PL*). However, 2PL that is not SS2PL is rarely utilized in practice.

**Phantom deadlocks** are deadlocks that are detected in a distributed system due to system internal delays but no longer actually exist at the time of detection.

## References


[1] Silberschatz, Abraham (2006). *Operating System Principles* (http://books.google.co.in/books?id=WjvX0HmVT1MC&dq=deadlock+operating+systems&source=gbs_navlinks_s) (7 ed.). Wiley-India. p. 237. . Retrieved 29 January 2012.

[2] Padua, David (2011). *Encyclopedia of Parallel Computing* (http://books.google.co.in/books?id=Hm6LaufVKFEC&lpg=PA1749&dq=computer networks deadlock definition&pg=PA524#v=onepage&q=deadlock&f=false). Springer. p. 524. . Retrieved 28 January 2012.

[3] Schneider, G. Michael (2009). *Invitation to Computer Science* (http://books.google.co.in/books?id=gQK0pJONyhgC&lpg=PA271&dq=deadlock signal lost&pg=PA271#v=onepage&q=deadlock signal lost&f=false). Cengage Learning. p. 271. . Retrieved 28 January 2012.

[4] Rolling, Andrew (2009). *Andrew Rollings and Ernest Adams on game design* (http://books.google.co.in/books?id=5Sj1Isrn_PnUC&lpg=PA421&dq=deadlock chicken egg&pg=PA421#v=onepage&q=deadlock chicken egg&f=false). New Riders. p. 421. . Retrieved 28 January 2012.

[5] Oaks, Scott (2004). *Java Threads* (http://books.google.co.in/books?id=mB_92VqJbsMC&lpg=PT82&dq=deadlock catch 22&pg=PT82#v=onepage&q=deadlock catch 22&f=false). O'Reilly. p. 64. . Retrieved 28 January 2012.

[6] A Treasury of Railroad Folklore, B.A. Botkin & A.F. Harlow, p. 381

[7] Shibu (2009). *Intro To Embedded Systems* (http://books.google.co.in/books?id=8hfn4gwR90MC&lpg=PA446&dq=coffman deadlock&pg=PA446#v=onepage&q=coffman deadlock&f=false) (1st ed.). McGraw Hill Education. p. 446. . Retrieved 28 January 2012.

[8] Stuart, Brian L. (2008). *Principles of operating systems* (http://books.google.co.in/books?id=B5NC5-UfMMwC&lpg=PA112&dq=coffman conditions&pg=PA112#v=onepage&q=coffman conditions&f=false) (1st ed.). Cengage Learning. p. 446. . Retrieved 28 January 2012.

[9] *Distributed Operating Systems* (http://books.google.co.in/books?id=l6sDRvKvCQ0C&lpg=PA177&dq=Tanenbaum ostrich&pg=PA177#v=onepage&q&f=false) (1st ed.). Pearson Education. 1995. p. 117. . Retrieved 28 January 2012.

[10] Mogul, Jeffrey C.; K. K. Ramakrishnan (1996). "Eliminating receive livelock in an interrupt-driven kernel" (http://citeseer.ist.psu.edu/326777.html). .

[11] Anderson, James H.; Yong-jik Kim (2001). "Shared-memory mutual exclusion: Major research trends since 1986" (http://citeseer.ist.psu.edu/anderson01sharedmemory.html). .

[12] Zöbel, Dieter (October 1983). "The Deadlock problem: a classifying bibliography". *ACM SIGOPS Operating Systems Review* **17** (4): 6–15. doi:10.1145/850752.850753. ISSN 0163-5980.


## Further reading


- Kaveh, Nima; Emmerich, Wolfgang. *Deadlock Detection in Distributed Object Systems* (http://www.cs.ucl.ac.uk/staff/w.emmerich/publications/ESEC01/ModelChecking/esec.pdf). London: University College London.
- Bensalem, Saddek; Fernandez, Jean-Claude; Havelund, Klaus; Mounier, Laurent (2006). "Confirmation of deadlock potentials detected by runtime analysis". *Proceedings of the 2006 workshop on Parallel and distributed systems: Testing and debugging* (ACM): 41–50. doi:10.1145/1147403.1147412.
- Coffman, Edward G., Jr.; Elphick, Michael J.; Shoshani, Arie (1971). "System Deadlocks" (http://www.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf). *ACM Computing Surveys* **3** (2): 67–78. doi:10.1145/356586.356588.
- Mogul, Jeffrey C.; Ramakrishnan, K. K. (1997). "Eliminating receive livelock in an interrupt-driven kernel". *ACM Transactions on Computer Systems* **15** (3): 217–252. doi:10.1145/263326.263335. ISSN 07342071.

- Havender, James W. (1968). "Avoiding deadlock in multitasking systems" (http://domino.research.ibm.com/tchjr/journalindex.nsf/a3807c5b4823c53f85256561006324be/c014b699abf7b9ea85256bfa00685a38?OpenDocument). *IBM Systems Journal* **7** (2): 74.
- Holliday, JoAnne L.; El Abbadi, Amr. "Distributed Deadlock Detection" (http://www.cse.scu.edu/~jholliday/dd_9_16.htm). *Encyclopedia of Distributed Computing* (Kluwer Academic Publishers).
- Knapp, Edgar (1987). "Deadlock detection in distributed databases". *ACM Computing Surveys* **19** (4): 303–328. doi:10.1145/45075.46163. ISSN 03600300.
- Ling, Yibei; Chen, Shigang; Chiang, Jason (2006). "On Optimal Deadlock Detection Scheduling". *IEEE Transactions on Computers* **55** (9): 1178–1187.


## External links

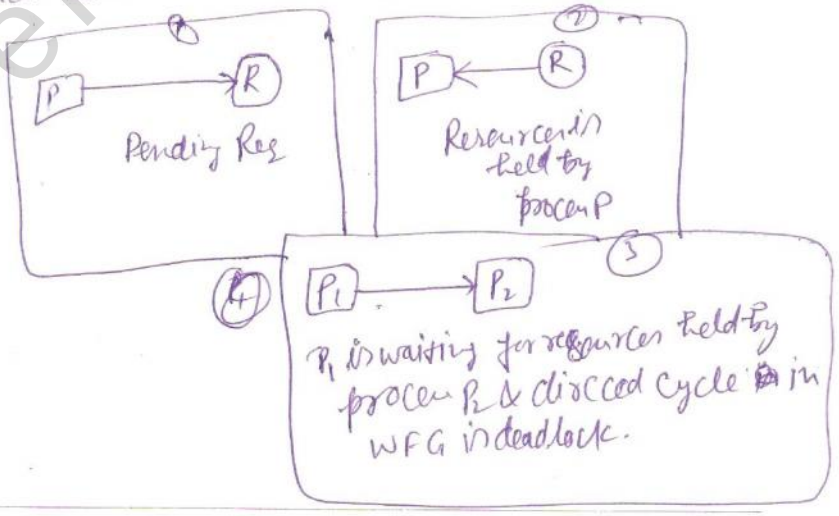
- " Advanced Synchronization in Java Threads (http://www.onjava.com/pub/a/onjava/2004/10/20/threads2.html)" by Scott Oaks and Henry Wong
- Deadlock Detection Agents (http://www-db.in.tum.de/research/projects/dda.html)
- DeadLock at the Portland Pattern Repository
- Etymology of "Deadlock" (http://www.etymonline.com/index.php?term=deadlock)
- ARCS - A Web Service approach to alleviating deadlock (http://www.arcs.us)

Deadlock Detection →

(1) Centralized deadlock detection- (one cordinator) → (E) → (2)
- WFG

(2) Distributed deadlock detection- (2)

(3) Hierarchical Deadlock detection- (2)



# DISTRIBUTED DEADLOCK DETECTION

## Introduction →

- A distributed system is a network of sites that exchange information with each other by message passing.
- A site consists of computing & storage facilities & interface to local users & a communication network.
- In DS, a process can request & release resources (local or remote) in any order, which may not be known a priori & a process can request some resources while holding others.
- If the sequence of the allocation of resources to process is not controlled in such environments, deadlock can occur.

## System Model →

- The Problem of deadlocks has been generally studied in DS under the following model -

  ① The systems have only reusable resources.
  ② Processes are allowed only exclusive access to resources.
  ③ There is only one copy of each resource.

- A process can be in two states - running or blocked.

  Running state (active state) - A process has all the resources & is either executing or ready to for execution.

  Blocked state - Process is waiting to acquire some resources.

## Resource Vs Communication Deadlock

Two types of deadlock has been discussed –

① resource deadlock – Process can simultaneously wait for several resources ~~deadlock~~ & can't process until they have acquired all those resources.

- A set of process is resource-deadlock if each process in the set requests resources held by another process in the set & it must receive all of the requested resources before it can become unblocked.

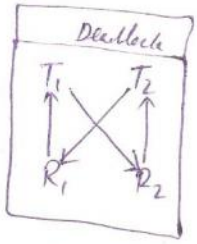
② Communication Deadlock → Processes wait to communicate with other process among a set of processes.

- A waiting process can unblocked on receiving a communication from any one of these processes.
- A set of process is communication-deadlock if each Process in the is waiting to communicate with another process in the set & no process in the set ever initiates any further communication until it receives the communication for which it is waiting.

"wait to communicate" → "wait to acquire a resource"

## A Graph-Theoretic Model →

- The state of Process-resource interaction in DS can be modeled by a bi-partite directed graph called a resource allocation graph.
- Nodes of this graph are processes & resources of a system, & the edges of the graph depict assignments or pending requests.
- A pending request is represented by a request edge directed from the node of a requesting process to the node of the requested resource.
- A resource assignment is represented by an assignment edge directed from the node of an assigned resource to the node of the assigned process.
- A system is deadlocked if its resource allocation graph contains a directed cycle or a knot.




## Wait-for-Graph → (WFG)

(See Page No-(152) of Singhal & Shivratri)

## Deadlock handling Strategies in Distributed System →

(1) Prevention. (It works by Preventing one of the 4 conditions of from occurring)
(ME, H&W, NP → CW)

(2) Avoidance. (Safe/unsafe state)

(3) Detection. (Here Deadlock are allowed to occur)
If deadlock is detected, it can be corrected by using one of the following methods —

① Process Termination.

② Resource Preemption.

## Issues in Deadlock Detection & Resolution →

- Deadlock detection & resolution entails addressing 2 basic issues –
  - (1) Detection of existing Deadlock.
  - (2) Resolution of Distributed deadlock.

### (1) Detection →

- The detection of Deadlock involved two issues –
  - * Maintenance of the WFG & Search of the WFG for the presence of Cycles (or Knots).
- In Distributed System, a cycle may involve several sites, so the search for cycles greatly depends upon how the WFG of the system is represented across the system.
- A correct deadlock detection algorithm must satisfy the following 2 conditions –
  - (1) Progress – No undetected deadlocks.
    - The algorithms must detect all existing deadlocks in finite time.
    - once a deadlock has occurred the deadlock detection activity should continuously progress until the deadlock is detected.
  - (2) Safety – No false Deadlock →
    - The algorithm should not report deadlocks which are non-existent (called Phantom deadlock).

Resolution →

→ Deadlock Resolution involves breaking existing wait for dependencies in the System WFG to resolve the Deadlock.

- It involves rolling back one or more processes that are deadlocked & assigning their resources to blocked processes in the deadlock so that they can resume execution.
- When WFG dependency broken, the corresponding info should be immediately cleaned from the system. If the info is not cleaned appropriately in a timely manner, it may result in detection of Phantom Deadlock.

Control organization for Distributed Deadlock Detection → (for detail Read Page No 155 - 5ks)

(1) Centralized Control - (Control site has the responsibility of constructing the Global WFG & searching it for cycles. (It has Single Point of failure)

(2) Distributed Control - Responsibility is shared equally. WFG is spread over many site, several sites participate in Deadlock detection algo.

Advantage - RAG & WFG is easy to by Constructed.

Disadvantage - congestion (Bottleneck Problem) at single Point failure is there.

(3) Hierarchical Control -

- This approach combines benifits of centralized & Distributed approach in one
- Sites detect deadlock in only decendent sites (child site) & resultant WFG or RAG is forworded to site to Calculate Result of WFG or RAG.

Advantage → Not susceptible to Single Point failure.

Disadvantage → Difficult to design algorithm sort of NP-Complete Problem, & also proof correctness of algo is difficult.

① Centralized Deadlock Detection Algorithms →

(a) Ho-Ramamoorthy's one & two phase algorithms.

② Distributed Deadlock Detection Algorithms →

(a) Obermarck's Path Pushing Algorithm.

(b) Chandy - Misra - Haas Edge chasing algorithm

(c) Diffusion computation (d) global state detection.

③ Hierarchical Deadlock detection -

(a) - Menasce - Muntz Algorithm.

(b) - Ho - Ramamoorthy's Algorithm.

---

①/(a) Ho-Ramamoorthy Algo →

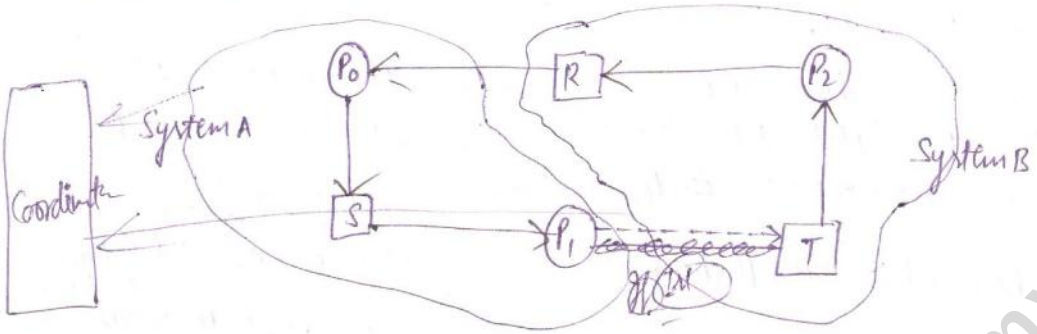
(a)(1) One-Phase algo →

- Each site maintains 2 status table -
  - Process table
  - Resource table
- One of the site Becomes the central Control site.
- The Central Control site periodically asks for the status tables.
- Control site builds WFG using the status tables.
- Control site analyzes WFG & resolves any present cycles.

Shortcomings →

- Phantom Deadlocks.
- High Storage & Communication Costs.

## Phantom Deadlocks




- System A has Process $P_1$ → holding resource S & waiting for T which is in System B. Resource T is presently held by Process $P_2$.

- Here Process $P_1$ sends 2 message to the Control Site stating the ① release of resources S & ② need of resources T.

- So in case the Message 2, that is waiting for T reaches the Control Site first then a cycle would be detected & it would be indicates that the System has a deadlock. (The false deadlock is called a Phantom deadlock).

### 7(a) 2-Phase Algorithms

- Each site maintains a status table for Processes that initiated at that site.
- Resources Locked & Resources waited upon.

① **Phase-1** - Control Site periodically asks for there locked & waited tables.
- It then searches for presence of cycle in there tables.

② **Phase-2** - If cycle are found in Phase-1 Search, Control site (WFG) makes 2nd request for the tables.
- The details found common in both table requests will be analyzed for cycle confirmation.

Short Coming - (Fantom deadlock)

- The one Phase algo is faster & requires fewer messages as compared to the two-Phase algorithm.
- However, it requires (1-Phase) more storage because every sites maintains two status tables & exchanges bigger messages because message contains 2 tables instead of one.

DDA →

(2)(a) Obermark's Path-Pushing algo → (Designed for Distributed database system) so Process are defined as (T) (TWF)

- In Path Pushing deadlock detection algo, information about the wait for dependencies is propagated in the form of Paths.

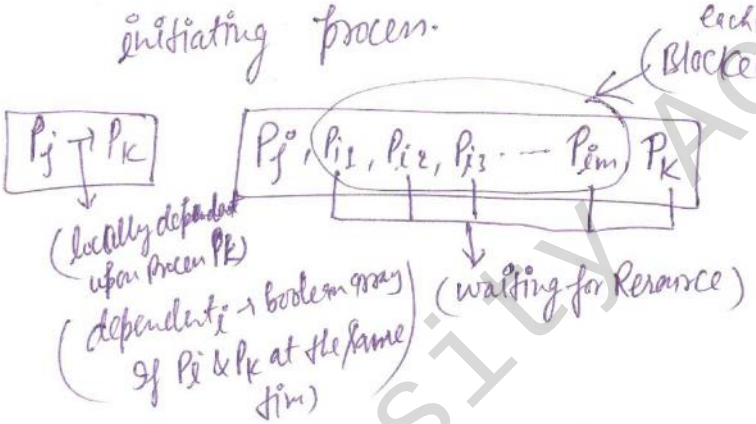
Algo →

- The site waits for deadlock-related information from other sites.
- The site combines the received information with its local TWF (Transaction wait for Graph) graph to build an updated TWF graph.
- For all cycles 'Ex → $T_1$ → $T_2$ → Ex' which contain the Node 'Ex', the site transmits them in string form 'Ex, $T_1$, $T_2$, Ex' to all other site where a sub-transaction of $T_2$ is waiting to receive a message from the Sub Transaction of $T_2$ at that site.

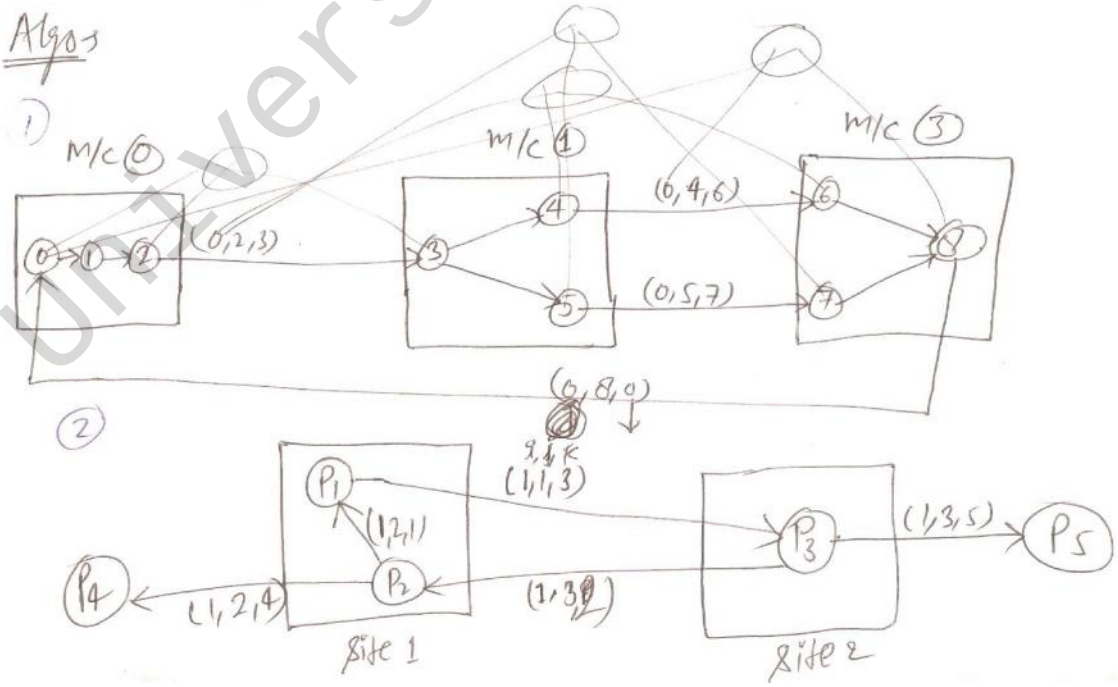
2(B) Chandy-Misra Haas's DDA ⑧

An Edge-Chasing Algorithm:

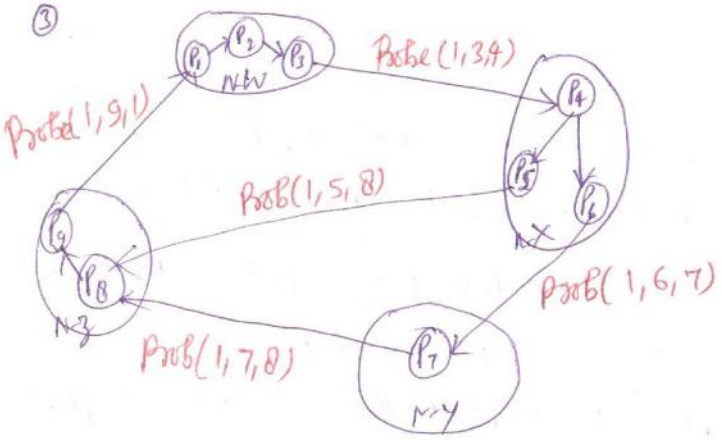
- Chandy algo uses a special message called a probe. A probe is a triplet $(i, j, k)$ denoting that it belongs to a deadlock detection initiated for process $P_i$ & it is being sent by the home site of Process $P_j$ to the home site of process $P_k$.
- A Probe message travels along the edges of the global TWF graph, & a deadlock is detected when a probe message return to its initiating process.




Algo:



③



## DISTRIBUTED DEADLOCK DETECTION →

- In DDD algorithm, all sites collectively cooperate to detect a cycle in the state graph that is likely to be distributed over several sites of the system.
- A DDDAlg. can be initiated whenever a process is forced to wait, & it can be initiated either by the local site of the process or by the site where the process wait.
- It is devided into 4 class -

① Path Pushing → The wait for dependencies information of the Global WFG is disseminated in the form of path (i.e a sequence of wait-for dependency edges).

② Edge-chasing → Special Message called probes are circulated along the edges of the WFG to detect a cycle. When a blocked process receives a probes, it propogates the probe along its outgoing edges in the WFG.

- A process declares a deadlock when it receives a probe initiated by it. (probes are of a fixed size Normally very short)

③ Diffusion Computation type →

- Make use of echo algorithm to detect deadlock. To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG.
- Thes queries are successively propogated through the edges of the WFG
- queries are discarded by a running process are echoed back by block processes as following way — when a blocked process receives first query message for a Particular deadlock detection initiation, It does not send a reply message until it has received a reply

Message for every query it sent (to its successor in WFG), for all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message. The initiator of a deadlock detection detects a deadlock when it receives a reply for every query it sent.

④ Global detection based deadlocks:

- A consistent snapshot of a DS can be obtain without freezing the underlying computation.

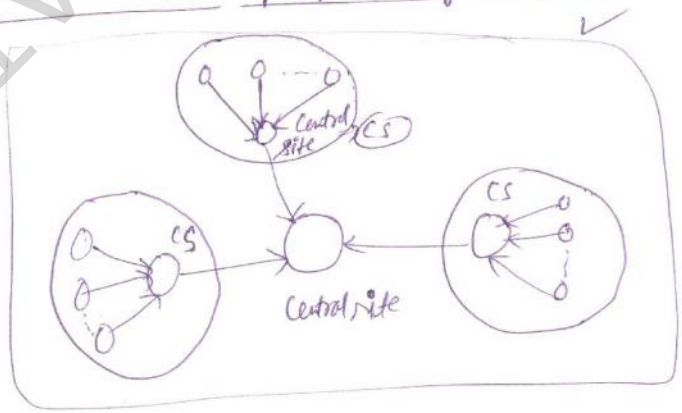
## ③ Hierarchical Deadlock Detection :-

### ① Menasce-Muntz Algo :-

- Sites (controllers) organized in a tree structure.
- leaf controllers manage local WFG.
- Upper controllers handles Deadlock Detection.
- each parent node maintains a Global WFG, union of WFG's of its children. Deadlock detected for its children.
- Changes propogated Upwards in the tree.

### ② Ho-Ramamoorthy's Algo :-

- Site are Grouped into clusters.
- Priodically 1 site choosen as central control site -
  - Central control site chooses control site for other clusters.
- Control site for each cluster collects the status graph these -
  - Ho-Ramamoorthy 1-Phase algo centralized DD used.
- All Control sites forward status report to central control site which Combines the WFG & performs cycle search.



# UNIT-III



## UNIT-3

## AGREEMENT PROTOCOL

**Introduction :->**

* In Distributed Systems, where sites (or processors) often compete as well as Cooperate to achieve a common goal, it is often required that sites reach Mutual agreement.
* For ex, In Distributed Data managers at sites must agree on whether to Commit or to abort a transaction.
* Reaching an agreement typically requires that sites have Knowledge about the values of other sites.
  ex- In DC commit, a site should Know the out come of local Commit at each site.
* When the System is free from failures, an agreement can eaisly be reached among the processors (or sites).
  ex- processors can reach an agreement by communicating their values to each other and then by taking a majority vote or a minimum, Max$^{m}$, mean etc, of those values.
* When a System Prone (Tendency) to failure this method does not work. this is because faulty processors can send conflicting values to other processors preventing them from reaching an agreement.
* In the Presence of faults, processors must exchange their values with other processors & relay the values received from other processors several times to isolate the effects of faulty processors.
* A processors refines its value as it learns of the values of other processors. this entire process of reaching an agreement is called an agreement Protocol.

- A very general model of faults is assumed. ex- a faulty processor may send spurious (ok but false) messages to other processors, may lie, may respond to received message correctly etc.
- Also non faulty processors do not know which processors are faulty.

- In Agreement problems, nonfaulty processors in Distributed system should be able to reach a common agreement, even if certain components in the system are faulty.
- The agreement is acheived through an agreement protocol that involves several rounds of message exchange among the processors.

## The System Model :-

Agreement problems have been studied under the following system model.

1. There are n processors in the system & at most m of the processors can be faulty.
2. The processors can directly communicate with other processors by message passing. thus, the system is logically fully connected.
3. A receiver processor always knows the identity of the sender processor of the message.
4. The communication medium is reliable (i.e. it delivers all messages without introducing any errors) & only processors are prone to failure.

- For simplicity, we assume that agreement is to be reached bet^n only two values, 0 & 1. Results can easily be extended to multivalue agreement.

## Synchronous Vs Asynchronous Computations—

- In a Synchronous Computation, processes in the system run in lock step manner, where in each step, a process receives messages (sent to it in the previous step), performs a computation, & sends messages to other processes (received in the next step). A steps of Synchronous Computation is also referred as a round.
- In Synchronous computation, a process knows all the messages it expects to receive in a round. A message delay or a slow process can slow down the entire system or computation.
- In an asynchronous computation, on the other hand, the computation at processes does not proceed in lock steps. A process can send & receive messages & perform computation at any time.

" Here in this chapter, synchronous models of computation are assumed."
the assumption of synchronous computation is critical to agreement protocols.

## Model of Processor failures :→

- In Agreement Problem we consider a very general model of processor failures
  * A processor can fail in 3 modes: crash fault, omission fault, Malicious fault.
  * In Crash fault → A processor stop functioning & never resume oper^n.
  * In omission fault → A processor "omits" to sends messages to some processes. (there are the messages that the processor should have send according to protocol or Algo it is executing.)(fail to perform some steps)
  * In Malicious fault - a processor behaves randomly & arbitrarily. (Malicious fault are also known as Byzantine faults)

Classification of Faults-

Based on Components that faults

- program/process
- Processor/Machine
- Link
- Storage
- Clock

Based on behaviour of faulty Component

- Crash - just crash
- Failstop - Crash with additional conditions
- omission - fails to perform some steps
- Byzantine → behaves arbitrarily
- Timing → violates timing constraints

---

Authentication Vs Non-Authentication Messages →

- To reach an agreement, processors have to exchange their values & relay the received values to other processors several times. the capability of faulty processors to distort (make false by addition) what they receives from other processors greatly depends upon the type of underlaying messages.
- there are two types of message - authenticated (signed messg) & non-authenticated (oral messg).
  * In an authenticated message system, a (faulty) processor cannot forge a message or change the contents of a received message.
  * In non-authenticated message system, a (faulty) processor can forge a message & claim to have received it from another processor or change the content of a received message before it relays the message to the other processors. A processor has no way to verifying the authenticity of a received message.

Performance aspect → The Performance or computational complexity of agreement protocol is generally determined by the following three metrics -

① time - (time taken to reach an agreement under a protocol)
② message traffic → (measured by the no of messages exchanged to reach an agreement.)
③ storage overhead - measures the amount of information that needs to be stored at processors during the execution of a protocol.

## Classification of Agreement Problems:-

- There are three well known Agreement Problem in distributed System-

Imp

1. The Byzantine Agreement Problem.
2. The Consensus Problem.
3. the interactive consistency Problem.

- In Byzantine agreement problem, a single value, which is to be agreed on, is initialized by an arbitrary processor & all nonfaulty processors have to agree on that value.
- In Consensus Problem, every processor has its own initial value & all nonfaulty processors must agree on a single common value.
- In the interactive consistency problem, every processors has its own initial value & all nonfaulty processors must agree on a set of common values.

* In all three problems, all nonfaulty processors must reach a common agreement.
- In Byzantine agreement & Consensus problems, the agreement is about a single value.
- Whereas in interactive consistency problem, the agreement is about a set of common values.

Table-1 Three Agreement Problem.

| Problem | Byzantine Agreement | Consensus | Interactive consistency |
|---|---|---|---|
| Who initiate the Value | One processor | All processors | All processors |
| Final agreement | Single Value | Single Value | A vector of values |

① The Byzantine Agreement Problem :→

- In the Byzantine agreement problem, an arbitrarily choosen processor. Called the source processor, broadcasts its initial value to all other processors.
- A Sol$^{n}$ to the Byzantine agreement problem should meet the following two objectives -

  Agreement → All nonfaulty processors agree on the same value.

  Validity → If the source processor is nonfaulty, then the common agreed upon value by all nonfaulty processors should be the initial value of the source.

- Two Point should be noted ~~if all nonfaulty~~
  ① if the source processor is faulty, then all nonfaulty processors can agree on any common value
  ② It is irrelevant what value faulty processors agree on or whether they agree on a value at all.
  ③ Termination → each Nonfaulty processor must eventually decide on a value.

② The Consensus Problem :→

- Every Processors Broadcast its initial value to all the other Processors. initial Value of the processors may be different. A protocol for reaching Consensus should meet the following conditions -

① Agreement → All nonfaulty Processors agree on the same single value.

② Validity → If the initial value of every nonfaulty processor is v, then the agreed upon common value by all nonfaulty processors must be v.

- Note that if the initial value of nonfaulty processors are different, then all non-faulty processors can agree on any common value. again we don't care what value faulty processors agree on.

③ Termination → Each Non-faulty procen must eventually decide on a value.

③ The Interactive Consistency Problem :→

- In the interactive consistency problem, every processor broadcasts its initial value to all other processors. the initial values of the processors may be different.
- A protocol for the interactive consistency problem should meet the following conditions-

Agreement → All non faulty processors agree on the same vector $A(V_1, V_2, \dots V_n)$.

Validity → If the ith processor is nonfaulty & its initial value is $V_i$, then the ith value to be agreed on by all nonfaulty processors must be $V_i$.

Termination → Each nonfaulty processor must eventually decide on the array A.

Relations Among the Agreement Problems →

- All three Agreement problems are closely related.
- The Byzantine agreement problem is a special case of the interactive consistency problem, in which the initial value of only one processor is of interest.
- Conversely, if each of the n processors runs a copy of the Byzantine agreement protocol, the interactive consistency problem is solved.
- The consensus problem can be solved using the sol$^n$ of the interactive consistency problem. this is because all nonfaulty processors can compute the value that is to be agreed upon by taking the majority value of the common vector that is computed by an interactive consistency protocol, or by choosing a default value if a majority does not exist.
- Thus sol$^n$ to the interactive consistency & consensus problems can be derived from the sol$^n$ to the Byzantine agreement problem.

Approach to the Byzantine Agreement Problem →

- The Byzantine agreement Problem was first defined & Solved.
- (ex - Byzantine Agreement Problem def$^n$.)

- It is obvious that all the processors must exchange the values through messages to reach a Consensus. Processors send their values to other process & relay received value to other processors. During the execution of the Protocol, faulty processors may confuse other processors by sending them Conflicting Values or by relaying to them fictitious values. → (false)

- Byzantine Agreement Problem is referred as the Byzantine generals problem, because the Problem resembles a situation where a team of generals in an army is trying to reach agreement on an attack Plan.

- The general are located a geographically distant & ~~communicate through messengers~~ Communicate through messengers. Some of the generals are traitors (equivalent to faulty Processors) & try to prevent loyal generals from reaching an agreement by deliberately transmitting erroneous information.

(1) The upper bound on the No of Faulty Processors →

- In order to reach an agreement on a common value, nonfaulty processors need to be free from the influence of faulty processors.
- If the faulty processors dominate in number, they can prevent non faulty processors from reaching a Consensus. Thus the no of faulty processors should not exceed a certain limit if a consensus is to be reached.
- It is impossible to reached a Consensus (Agreement) if the no of faulty processors, m, exceeds $\lceil (n-1)/3 \rceil$. (Lamport)
- $\lceil M+1 \rceil$ round of message exchanged (no faulty process)

$\lceil M+1 \rceil$ → is the lower bound on the No of message exchanged to reach a Byzantine agreement in fully connected

## (2)(2) An Impossibility Result →

- Byzantine Agreement is impossible if $n > \lfloor (n-1)/3 \rfloor$

  $m \rightarrow$ faulty Processors.

  $n \rightarrow$ 3 processors

  $\lfloor (3-1)/3 \rfloor = \lfloor 2/3 \rfloor = 0$

- Byzantine Agreement is impossible with $< (m-1)$ messages exchanged.

- Byzantine Agreement cannot be reached among three processors, where one processor is faulty.

- Consider a system with three processors, $P_0$, $P_1$ & $P_2$. we assumed that there are only 2 values 0, 1, on which processors agree & processors $P_0$ initiate the initial value. there are two possibility –

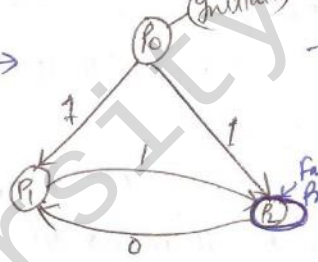
  (1) $P_0$ is faulty or (2) $P_0$ is not faulty

Faulty processor → oval (⬭)
Non faulty → ○

### (1) $P_0$ is not faulty →

Assume $P_2$ is faulty →

processor $P_1$ must accept 1 as the agreed upon value if condition 2 is be satisfied.



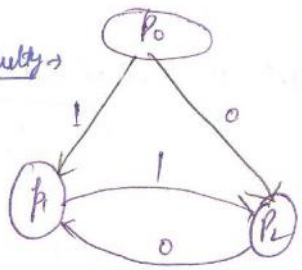

$P_0$ broadcast the initial value of → 1 to $P_1$ & $P_2$.

$P_2$ → Malicious processor & Communicate of value 0 to $P_1$.

$P_1$ → receive Conflicting value from $P_0$ & $P_2$

### (2) $P_0$ is faulty →

Assume Initiator $P_0$ is faulty →




$P_0$ send → 1 to $P_1$ & 0 to $P_2$

$P_2$ Communicate the value 0 to $P_1$

Recact

✓ $P_2$ → It must be 0 agreed upon value

✓ at $P_1$ → It must be 1 agreed upon value

→ Therfore No Sol$^{n}$ exists for Byzantine Agreement Problem for three processors, which can work under single process failure.

③ **Lamport-Shostak-Pease Algo →** (Oral message algo OM(m), m>0)

- Solves Byzantine agreement Problem for 3m+1 or more processors in the presence of at most m faulty processors.

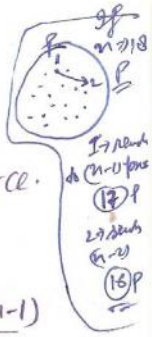
If m=1
n ≥ 3×1+1 ⇒ 4 total processors (3+1)

- If n be the total No of processors then $n \geq 3m+1$.
- The Algorithm is recursively defined as —

① Algorithm OM(0) :→ (OM(0) → If there is 0 faulty processor in the System)

1. The source processor sends its value to every processor.
2. each processor uses the value it receives from the source. (If receives no value then it uses a default value of 0).

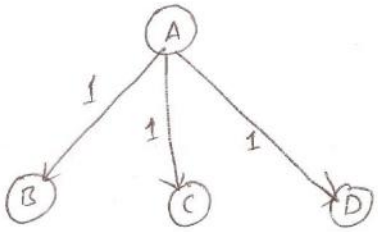
② Algorithm OM(m), m>0 :→



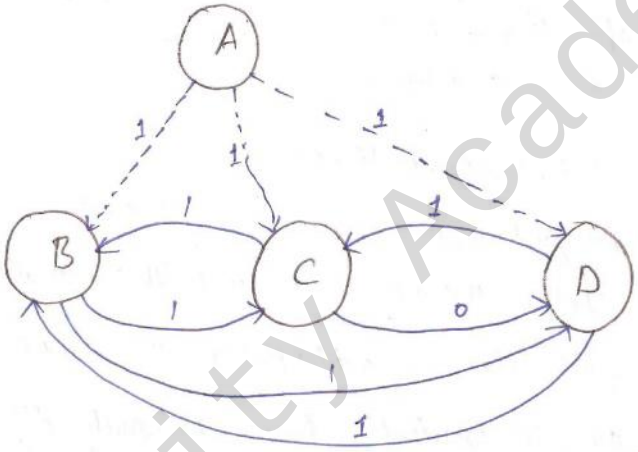

1. The source processor sends its initial value to every processor.
2. For each i, let $V_i$ be the value, processor i receives from the source. (If receives no value, then it uses a default value of 0).
   - processor i acts as the new source & initiates Algorithm OM(m-1) where it sends the value $V_i$ to each of the (n-2) other processors.
3. For each i and each $j (\neq i)$, let $V_j$ be the value, processor i received from processor j in step 2 using Algorithm OM(m-1). (If it receives no value, then it uses a default value of 0).
   - Processor i uses the majority $(V_1, V_2 \cdots V_{n-1})$.

* The execution of the Algorithm OM(m) invokes (n-1) separate executions of the Algo OM(m-1), each of which invokes (n-2) executions of the algorithm OM(m-2), & so on.
* Therefore, there are (n-1)(n-2)(n-3)···(n-K) separate executions of algo OM(m-K), K = 1, 2, 3, ··· m+1.
* The message complexity of the algo is $O(n^m)$.

Ex→ 4 Processor ex- Nonfaulty Commander.

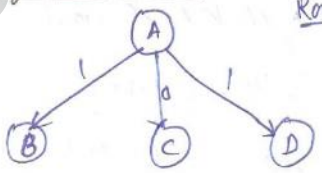



Round 1 - Processor A executes OM(1), where processor C is faulty

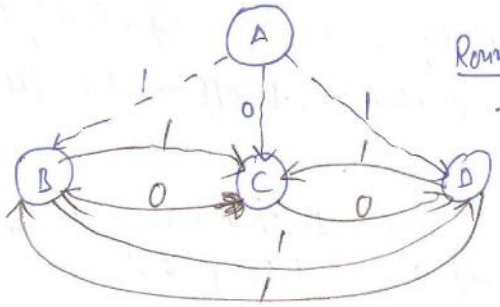



Round 2 : processors B, C, & D execute OM(0). Dashed line shows that Message sent during first Phase.

Ex→ 4 Processor - ex faulty commander




Round ① - Processor A executes OM(1), where processor A is faulty.




Round ② - Processor B, C, & D execute OM(0).

Message complexity → (LIEUTENANT COMMANDER] (Problem of Attack)

* $T(0,1) = (n-1)$
* $T(m,n) = (n-1)\,T(m-1,n-1)$, for $m > 0$.
* $T(m,n) = (n-1)(n-2)(n-3)\cdots\cdot(n-(m+1)) \in O(n^m)$

APPLICATIONS OF AGREEMENT ALGORITHMS :→

- Algorithms for agreement Problems find applications in Problems where processors should reach an agreement regarding their values in the presence of malicious failures.
- Two such applications - (A) clock synchronization in distributed systems, & (B) atomic Commit in distributed database.

(A) Fault Tolerance clock Synchronization :→

- In Distributed Systems, it is often necessary that sites (or processes) maintain Physical clocks that are synchronized with one another.
- Since Physical clocks have a drift Problem, they must be periodically resynchronized.
- Such periodic synchronization becomes extremely difficult if the Byzatine failures are allowed.
- This is because a faulty process can report different clock values to different processes. (The description of clock synchronization is described in the Paper of Lamport & Millior-Smith. [H·W]

• The following assumptions regarding the System made-

$A_1$ → All clocks are initially synchronized to approximately the same value.

$A_2$ → A Nonfaulty process's clock runs the approximately the correct rate.

$A_3$ → A Nonfaulty process can read the clock value of another nonfaulty process with at most a small error $\epsilon$.

- A clock Synchronization algorithm should satisfy the following two conditions -

(1) At any time, the values of the clocks of all nonfaulty processes must be approximately equal.

(2) There is a small bound on the amount by which the clock of a nonfaulty process is changed during each resynchronization.

* There are 2 clock Synchronization algo, namely → (1) Interactive convergence algo.
(2) Interactive consistency algo.
- which are fault-Tolerant to the Byzantine failures.

Page No (190)

(1) Interactive Convergence algorithm :→ This algo assumes that the clocks are initially synchronized & that they are resynchronized often enough so that two nonfaulty clocks never differ by more than $\delta$. The algo works as follow -

Algo → Each Process reads the value of all other processes's clocks & set its own clock to the average of these values - However, if a clock value differs from its own clock value by more than $\delta$, it replaces that value by its own clock's value when forming the average.

Let
P, Q → Non faulty processes,
r → processes

$C_{pr}$, $C_{qr}$ → value used by P, Q.
when forming the average, respectively r's value →.
If r is nonfaulty - then $C_{pr} = C_{qr}$
If r is faulty then $C_{pr}$ & $C_{qr}$ will differs by (3δ)

$C_{pr} \to \delta$
$C_{qr} \to \delta$
$P_{kq} \to \delta$

Let n → total No of Processes
m → No of faulty processes
such that $n > 3m$

processes P & Q set their clocks to the average of n values $(C_{pr})$ & $(C_{qr})$ respectively, for $r = 1, \ldots n$.

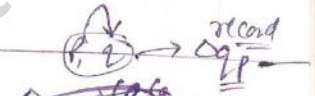
we have, → $C_{pr} = C_{qr}$ for the $(n-m)$ nonfaulty processes r, & $|C_{qr} - C_{pr}| \leq 3\delta$
for the m faulty processes r.

- It follows from this that the average computed by P & Q will differ by at most $\left(\frac{3m}{n}\right)\delta$.

- The assumption $n > 3m$ implies $\boxed{(3m/n)\delta < \delta}$, so the algo succeeds in bringing the clocks closer together.

- Therefor we can keep the Nonfaulty processes clocks synchronized to within $\delta$ of one other by resynchronizing often enough so that clocks which are initially within $(3m/n)\delta$ seconds of each other never drift further that $\delta$ seconds apart.

- If the clock reading error is $\epsilon$ then the difference in the clock values read by a process can be as long as $\delta + \epsilon$. Therefore, only clock differences larger than $\delta + \epsilon$ are replaced by 0 while computing the average increment.




$\Delta_{qp} \rightarrow c_q - c_p$

$\boxed{\Delta_{pp} = 0}$

$$\Delta_{pq} = \begin{cases} \Delta_{qp} & \text{if } |\Delta_{qp}| \leq \delta \\ 0 & \text{otherwise.} \end{cases}$$

(2) the Interactive Consistency Algorithms

odd no Median → $\frac{1}{2}(n+1)$ th value ⇒ -
even (8) M ⇒ $\left(\frac{4\text{th} + 5\text{th}}{2}\right)$

* This algo add two improvements —

① It takes the median of the clock values rather than the mean. the median provides a good estimate of the clock value, as the No of bad clocks will be low.

② It avoids the Problem of 2 faced clocks (which reports differ values to differ processes) by using a more sophisticated technique to obtain clock values of the processes.

(B) Atomic Commit in DDBS →

- In the Problem of atomic commit, sites of DBMS must agree whether to commit or abort the transaction.
- In the first Phase of the atomic commit, sites executes their Part of a distributed transaction & broadcast their decisions (Commit or abort) to all other sites.
- In the Second Phase each site, based on what it received from other sites in the first Phase, decides whether to commit or abort its Part of distributed transaction.

Since all sites receives an identical response from all other sites, they will reach the same decision. However, if some sites behaves maliciously, they can send a conflicting response to other sites, causing them to make Conflicting decisions.

- In this situations, we can use algorithms for the Byzatine agreement to prove that all nonfaulty processors reach a common decision about a distributed transaction.
- It works as follow— In the first Phase after a site has made a decision, it starts the Byzatine agreement, In the Second Phase Processors determine a common decision based on the agreed vector of Values.

# Distributed Resource Management

- A distributed file system is a Resource management component of a distributed Operating System. It implements a common file system that can be shared by all the autonomous computers in the system.

NFS (Sun's Net[K] File System)
- Windows NT, Zeno, XP
- AFS (Andrew File System)

- Two important goals of distributed file systems are — Net[K] transparency. / High availability.

① Network transparency — The primary goal of a DFS is to provide the same functional capabilities to access files distributed over a network as the file system of a timesharing mainframe system does to access files residing at one location.

- Access - Performance
- Location - Scaling — Transparency.
- Mobility

- Ideally, users do not have to be aware of the location of files to access them. this property of DS is known as network transparency.

② High availability :→ users should have the same easy access to files, irrespective of their physical location. System failures or regularly scheduled activities such as backups or maintenance should not result in the unavailability of files.

- In the recent years, several distributed files systems have been developed. Here we will ~~read~~ understand that how common mechanisms & design aspects shared by today's Distributed file systems.

## Architecture of Distributed File System

- Ideally, in a Distributed file system, files can be stored at any machine (or computer) and the computation can be performed at any machine (i.e the machine or peers).
- When a machine needs to access a file stored on a Remote machine, the remote machine performs the necessary file access operations & return data if a read operation is performed.
- However, for higher performance, several machines, referred to as file servers, are dedicated to storing files and performing storage & retrieval operations the rest of the machines in the system can be used solely for computational purpose. These machines are referred as clients & they access the files stored on the servers. (Client can use cache concept for fast access of servers)

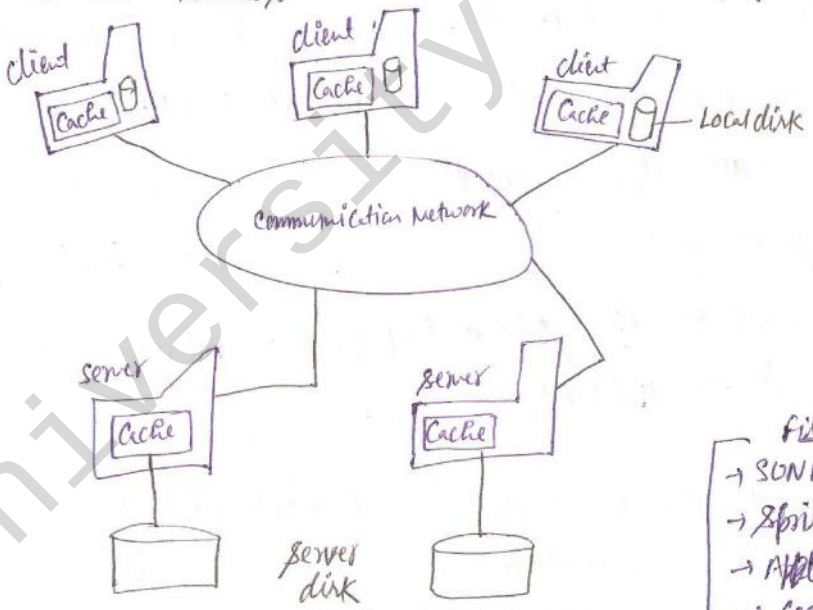



fig- Distributed File System Architecture

File System -
→ SUN File System
→ Sprite file System
→ Apollo Domain DFS
→ Coda FS
→ X-Kernel Logical FS
→ Log structured FS

Two Most important Services present in a distributed file System are the-

① Name Server (Map Name to stored objects) ② Cache Manager (Perform file caching) & Can be present at both side client & Server

① Name Server :→ It is a process that maps names specified by client to stored objects files & directories. The mapping (also referred as a name resolution) occurs when a process references a file or directory for the first time.

② Cache Manager :→ It is a process that implements file caching.

Cache on → client deals with Net^k latency
→ server deals with disk latency.

- In file caching, a copy of data stored at a remote file server is brought to the client's m/c when referenced by the client. Reducing the access delays due to Network latency. Cache managers can be present at both side of client's & Servers
- Cache managers at the servers cache files in the main memory to reduce delays due to disk latency.

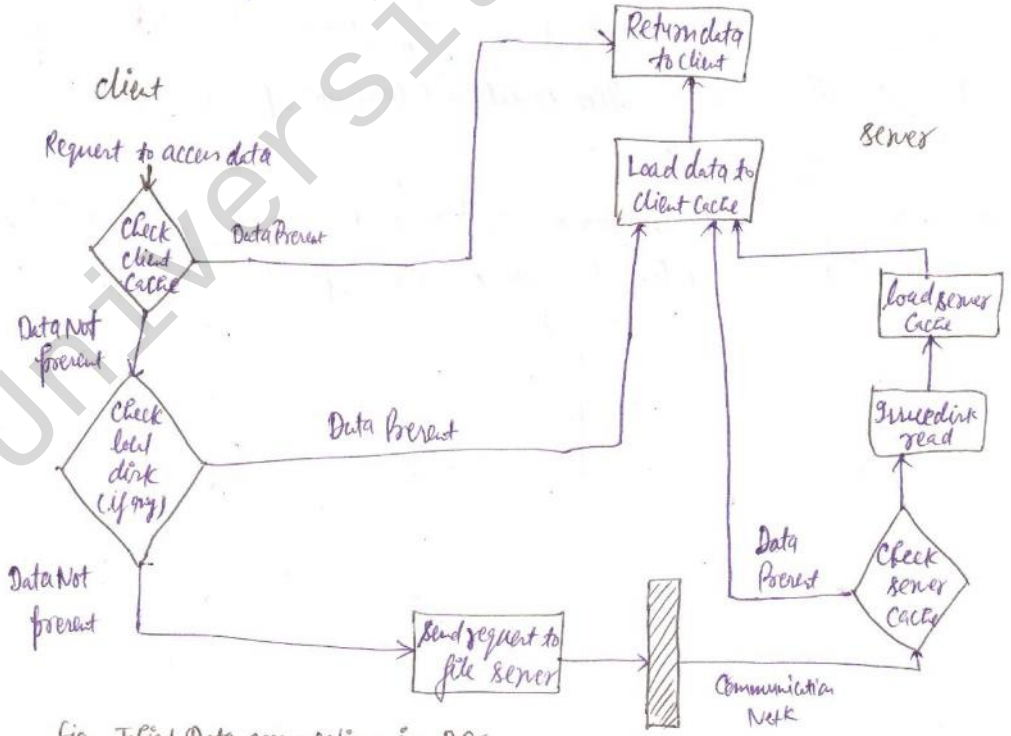



Fig- Typical Data access actions in DFS

Typical steps :→

- Check client cache, if present, Return Data.
- Check local disk, if present, load into local cache, return data.
- Send request to file server.
- . . . .
- Server checks cache, if present, load into client cache, return data.
- disk read
  - load into Server Cache
  - return data.

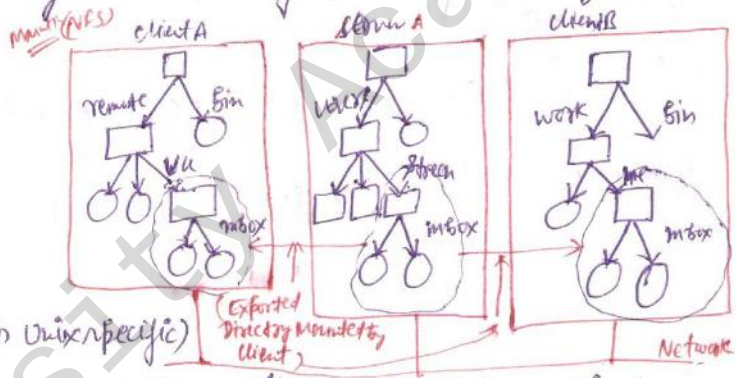
(name files are not unique)

SA – /users/stream/mbox
CB – /remote/vu/mbox
CC – /work/me/mbox

## Mechanisms for Building Distributed File Systems :

- The Basic Mechanisms for Building Distributed File systems involves following steps -

1. Mounting.
2. Caching.
3. Hints.
4. Bulk Data transfer.
5. Encryption.




① Mounting :→ (Mounting is Unix specific)

- It allows the binding together of different filename space to form a single hierarchical structured name space.
- A name space (or a collection of files) can be bounded to or mounted at an internal node or a leaf node of a name space tree.

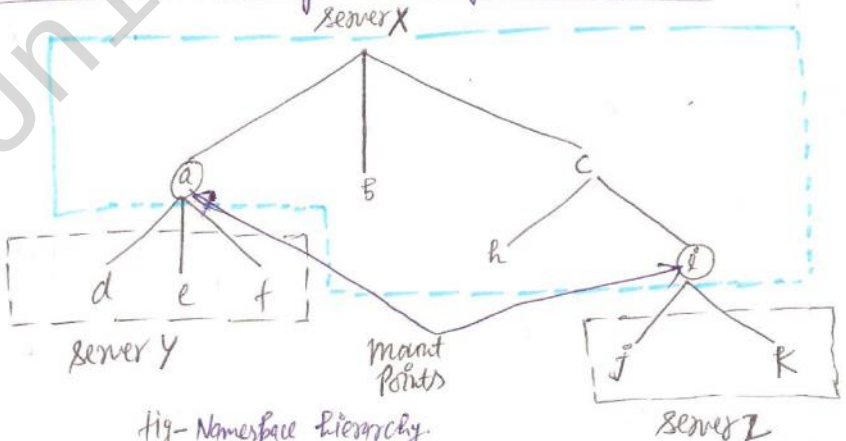



fig- Namespace Hierarchy.

- A Node onto which a name space is mounted is known as mount point.
- In this fig- Node a & Node i are mount points at which directories stored at Server Y & Server Z are mounted, respectively. [Namespace (directory)]
- Node a & i are internal nodes in the name space tree. the Kernel maintains a structure called the mount table, which maps mount points to appropriate storage devices.
- In the DFS, file system maintained by remote servers are mounted at the client.
- There are two approaches to maintain the mount information-
  - ✓ (1) At the client → (NFS) (Sun Net[K] File System)
    - Different clients may see different files. (Every client Need to update its mount table)
  - ✓ (2) At the Servers (Sprite file system)
    - all client may see the same filename space. good when file move around the servers.

(2) Caching → It implement in DFS to reduce delays in the accessing data.

- In file caching, a copy of data stored at a remote file server is brought to the client when referenced by the client.
- Subsequent access to the data is performed locally at the client, thereby reducing access delays due to Net[K] latency.
- Caching exploits the (temporal locality of reference) exhibited by programs.
  - (Refers to the fact that a file recently accessed is likely to accessed again in the near future.)
- Also data is cached ~~in the server~~ in the main memory (server cache) at the servers to reduce disk access latency.

③ Hints →

- Caching improves file system Performance by reducing the delay in accessing data.
- When Multiple client cache & modify shared data, the Problem of cache consistency arises.
- Specifically, it must be guaranteed that the cached data is valid (up-to-date) & that a copy of data - recently updated in some other client cache or in the file server - does not exist.
- Guaranteeing consistency is expensive in Distributed file system as it requires elaborate cooperation bet$^n$ fileservers & clients.

→ An Alternative approach is to treat the cached data as Hints. In this case, cached data are not expected to be completly accurate.

- Ex After the name of a file or directory is mapped to the Physical object, the address of the object can be stored as a hint in the cache. If the address fails to map to the object in the following attempt, the cached address is purged (removed) from the cache. the file Server consults the name server to determine the actual location of the file or directory & updates the cache.

④ Bulk Data transfer → Transfering Data in Bulk reduces the Protocol processing overhead at both servers & clients.

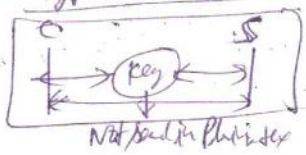
Communication cost = $S + C*B$

S → Startup cost, mostly S/w

C → Per Byte Cost

B → No of Bytes to Send.

⑤ Encryption → typical method to enforce security in Distributed system.



Design Issues → ① Naming & Name Resol$^{n}$ ② Name Server ③ Caching on Disk or Main Memory ④ writing Policy ⑤ Cache Consistency ⑥ Availability ⑦ Scalability ⑧ Semantics

(4)

① Naming & name resolution →

* A Name is associated with an object (such as file or directory)
* Name Resolution - Mapping a Name to an object.
* Name Space - (a collection of Names). three different methods to name files. → (directory)

① Concatenate the Host name to the name of the file on that Host.
- (A) Simple unique file Name.
- (B) No Network Transparency.
- (C) Moving a file from one machine to another may require modification to the application.
- (D) Name Resolution is simple. - Don't need a Name server.

② Mount remote directories onto local directory →
- (A) once mounted, accessing the file becomes location transparent.
- (B) Name Resolution is simple.

③ Have a Single Global Directory → (Sprite & Apollo FS)
- where all the files in the system blong to a single name space.
- The Main disadvantage of this scheme, however, is that it is mostly limited to one computing facility or to a few cooperating computing facilities.
- Not scalable how to maintain the unique filenames.

Name Server :→
- In a centralized system, name resolution can be accomplished by maintaining a table that maps names to objects.
- In distributed systems, name servers are responsible for name Resol$^{n}$.
- A name server is a process that maps names specified by clients to stored objects such as files & directories.
- The easiest approach to name resol$^{n}$ in DS is for all clients to send their queries to a single name server which maps names to object.

- This Approach has the following serious drawback -

① If the Name Server crashes - the entire system is drastically affected.

② Name server may become a bottleneck & seriously degrade the performance of the system. [a/b/c] → mapped to objects may require Remote mapping for [b/c]

② Caches on Disk or Main Memory :-> ✓ Virtual Memory → (taken from Hard disk) (Real term)
✓ Cache memory → (Internal memory of CPU)

The advantages of having the cache in the Main memory are as following -

- ✓ Can be used in Diskless workstation
- ✓ accessing a cache in main memory is much faster than accessing a cache on local disk.
- Same techniques for server cache & client cache.

✓ The main disadvantage of having client-cache in main memory is that it competes with the virtual memory system for Physical memory space.

→ Virtual Memory System more complex - data cannot be in both Vm & Cache.
→ larger files can't be cached completely - need to able to chop (cut in Pieces) the file. server s/w more complex.

✓ The Main Advantage of Caching on (local) Disk -
- large files can be cached.
- Virtual Memory mgt. is simple.

③ Writing Policy :-

- The writing Policy decides when a modified cache at a client should be transferred to the server.

① The simplest Policy is write-through, in this write-through, all writes requested by the applications at clients are also carried out at the servers immediately. (Synchronised way)

- The main advantage of write through is reliability. In the event of client crash little information is lost.
- A write-through policy does not take advantage of the cache.

② An Alternative writing Policy, delayed-writing Policy, delays the writing at the server. In this case modifications due to a write are reflected at the server after some delay.

- This approach can Potentially take advantage of the cache by performing many writes on a block present locally in the cache.
- Another motivation for delaying the writes is that some of the data (ex- Intermediate Results) could be deleted in a short time, in which case data need not be written at the server at all.
- while the delayed writing Policy takes advantage of the cache at a client, it introduces the Reliability Problem.
- In the event of client crash a significant Amount of Data can be lost.

④ Cache Consistency :-

- The schemes that can guarantee consistency of the data cached at clients.
- There are two approaches to guarantee that the data returned to the client is valid.

(1)

* In the server-initiated approach, servers inform cache managers whenever the data in the client caches becomes stale (old). Cache Managers at clients can then retrieve the new data or invalidate the blocks containing the old data in their cache.

(2)

* In the client-initiated approach, it is the responsibility of the cache managers at the clients to validate data with the server before returning it to the clients.

- Both of these approaches are expensive & unattractive as they require elaborate cooperation bet^n servers & cache managers. In both approach communication cost is High.
- The server-initiated approach requires the server to maintain reliable records on what data blocks are cached by which cache mangers.
- The client-initiated approach simply negates the benifits of having a cache by checking the server to validate data on every access.

(3)

- The third approach for cache consistency is simply not to allow file caching when concurrent-write sharing occurs. In concurrent write sharing, a file is open at multiple clients & at least one client has it open for writing.
- In this approach, the file server has to keep track of the clients sharing a file. When concurrent file sharing occurs for a file, the file server informs all the client to purge their cached data item belonging to that file. (Alternatively, concurrent file sharing can be avoided by locking files. (Apollo FS)

- Another issue ④ that a cache consistency scheme needs to address is sequential-write ④ ⑥ sharing, which occurs when a client opens a file that has recently been modified & closed by another client.

- Two Potential Problem with sequential-write sharing are—

  ① When a client opens a files, it may have outdated Blocks of the file in its Cache, & (use Timestamp associated with them)

  ② When a client opens a file, the current data blocks may still be in another client's Cache waiting to be flushed (clean or Remove),

  Server reges → (Client flushed the Modified blocks of a client Cache when over a new clients open the file for writing.

⑤ Availability :-

- Availability is the one of the important design issues of DFS. The failure of servers or the communication network can severely affect the availability of files.

- Replication is the Primary Mechanism used for enhancing the availability of files in DFS.

Replication → Under Replication, many copies or replicas of files are maintained at different servers. Replication is inherently expensive because extra storage space reqed to store the replica & the overhead incurred in maintaining all the replicas upto date.

- The Most Serious Problem of Replicas are-

  ① How to keep the Replicas of a file consistent &.

  ② How to detect inconsistent among replicas of a file & subsequent to Recover there inconsistencies.

UNIT of Replication :-

- A fundamental design issue in replication is the unit of Replication.
- The most basic unit is a file. This unit allows the Replication of only those files that need to have higher availability. It makes overall Replica Mgt harder. For ex- the protection rights associated with a directory have to be individually stored with each replica.
- Replication unit can be → a group of all the files of a single user or a files that in a server.
- The Group of files is referred as a volume. (The use of Volume Replication is that the Replica Mgt is easier)

Replica Mgt :-

- Replica Mgt is concerned with the maintenance of replicas & in making use of them to provide increased availability.
- Replica Mgt depends on whether Consistency is guaranteed by the Distributed File System
- Here we are concerned the Consistency with Replica only that is also known as Mutual Consistency, & not with the Consistency within a file.
- To ensure mutual Consistency among replicas, a special weighted Voting Scheme can be used.

Save No of (Votes + timestamp) as associated with → Each replica.

Vote = r or w (Must be obtain through read, write respectively).

Condition → $w > r$ & $r + w >$ "total No of votes" of all the Replica (only current update copies)

($r \rightarrow$ read, $w \rightarrow$ write)

(6) Availability:-

- Replication is the main mechanism.
- How to keep relicas consistent.
- How to detect inconsistencies among replicas.
- Consistency Problem may decrease the availability.
- Replica mgt - Voting Mechanism to read & write to replica.

(7)

(7) Scalability:-

* How to meet the demand of growing system?
* the biggest Problem - consistency issue -
  * caching - Reduce net$^k$ load & Server load.
    * Server-initiated Cache invalidation to maintain Cache Consistency: complexity increases as the System size increases.
  * exploiting the mode for file use - (most shared files are readonly)

(8) Semantics (Behaviour)

- what a user wants? strict consistency.
- users can usually tolerate a certain degree of errors in file Handling - No need to enforce strict consistency.

④ Writing Policy :>

- The writing Policy defined when a modified cache block at a client should be transferred to the Server.
- The Simplest Policy is write-through - In write through, all writes requested by the applications at clients are also carried out at the server immediately. (write is done synchronously both the cache & to the backing store)
- The main advantage of write-through is reliability. In the even of the client crash, little info is lost. A write-through does not take advantage of the cache.
- An alternating Policy, delayed writing policy, delays the writing at the server. In this case, modifications due to a write are reflected at the server after some delay. This approach can potentially take advantage of the cache by preforming many writes on a block present locally in the cache.

⑤ Cache Consistency :>

- Server-initiated - the server inform cache managers whenever the data in the client cache become stale (old).
- Client-initiated - the client makes sure the cache is clean (Validates) before returning the data to the Application.
- Both Server-initiated & client-initiated ~~scheme~~ Cache Consistency Schemes are expensive.
- Not allow caching when concurrent-write sharing occurs.
- Sequential-write sharing may also cause Problems - a client opens a files that has recently modified & closed by another client.
  - * outdated blocks in the cache: timestamp the cache copy. Contacting the server can find inconsistency.
  - * update data in another client's cache not in the server (delay write): flushing writes when other clients open.

# Distributed Shared Memory



## Introduction →

- Distributed computing has been based on the message passing model in which processes interact & share data with each other by exchanging data in the form of messages. (explicit message passing) (No Physical Shared Memory)
- DSM system is a Resource Mgt component of Distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory.
- The shared memory model provides a virtual address space that is shared among all nodes (computers) in a distributed system.

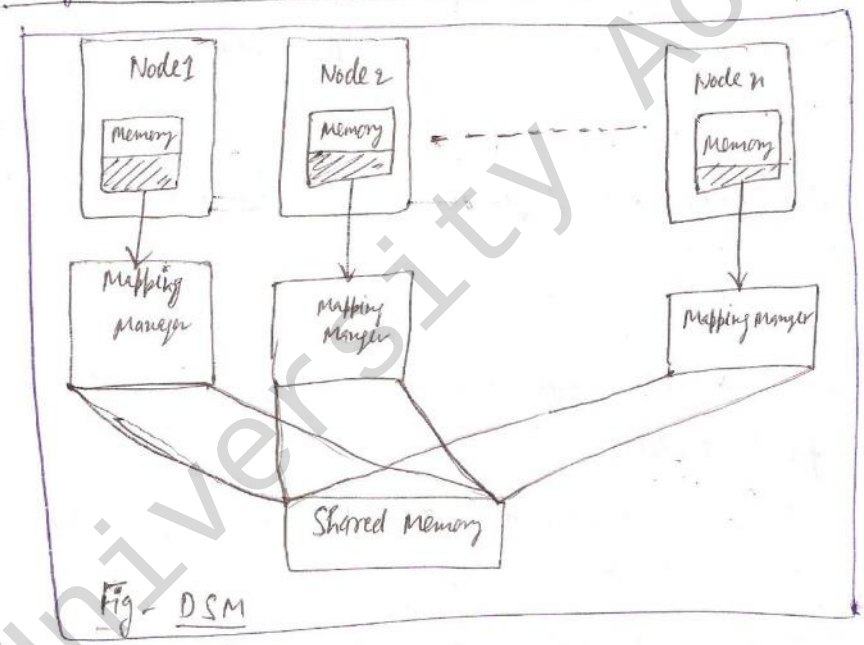



Fig - DSM

- To overcome this Problem (High Cost of communication) in DS, DSM system move data to the location of accen.

## Architecture & Motivation →

- With DSM, programs access data in the shared address space just as they access data in traditional virtual memory.
- In systems that supports DSM, data moves between secondary (HDD) memory & main memory (RAM) as well as betn main memories of different nodes.

- Each node can own (A Node which creates a data object owns the data object initially) data stored in shared address space, and the ownership can change when data moves from one node to another.

- When a Process access data in the shared address space, a mapping Manager maps the shared Memory address to the Physical Memory (which can be local or remote).

(RAM) Physical Block ⇒ PM ⇒ D ⇒ Frame Memory Unit
Logical Block ⇒ LM ⇒ D ⇒ Page ⇒ CPU gen

- The Mapping Manager is a layer of S/w implemented either in the OS Kernel or as a runtime library routine.

- To Reduce delays due to communication latency, DSM may move data at the shared Memory address from a remote node to the node that is accessing data (when the shared Memory address maps to a Physical Memory location on a Remote Node).

- In such case DSM Make use of the communication Services of the underlaying communication System.

Advantages of DSM are →

1. Data Sharing is implicit, hiding data movement (as opposed to 'send/receive' in message passing Model).
2. Passing data structures containing pointers is easier (In message passing Model Data Moves bet^n different address spaces). (use Pass By reference)
3. Moving entire object to user, DSM takes advantage of locality of reference.
4. Less expensive to Build than tightly coupled multiprocessor system: Off-the-self H/w, no expensive interface to shared Physical Memory.
5. very large total Physical Memory for all nodes - Large programs can run more efficiently.

⑥ No serial access to common bus for shared Physical Memory like in multiprocessor system. ②

⑦ Programs written for shared memory multiprocessors can be run on DSM system with minimum changes.

## ALGORITHMS FOR IMPLEMENTING DSM :->

1. The central-server Algorithm.
2. The Migration Algorithm :
3. The Read Replication Algorithm.
4. The Full Replication Algorithm.

- The central issue in the implementation of DSM are-
  - (A) How to keep track of the location of remote Data.
  - (B) How to Minimize Communication overhead when accessing Remote data.
  - (C) How to access concurrently remote data at several nodes (Computers). (Improve system performance)

① The central-server algorithm ->

- Central server Maintains all the shared data -
  - * Read Request : returns data item.
  - * write Request : Update data & returns acknowledge message.

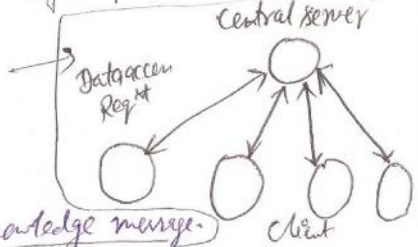



fig- The Central-server Alg.

- Implementation -
  - * A timeout is used to resend a request if acknowledgement fails.
  - * Associated Sequence Numbers can be used to detect duplicate write requests.
  - * If an Application's request to access shared data fails repeatedly, a failure condition is sent to the Application.
- While the central Server algo is easy to implement, the client server become a bottleneck.
- To overcome this problem the shared data can be distributed over several servers.
- Issue -> Performance & Reliability.
- Possible solⁿ -> - Partition shared data over several servers.
  - uses a mapping funⁿ to Distribute/locate data.

## ② The Migration Algorithm :-

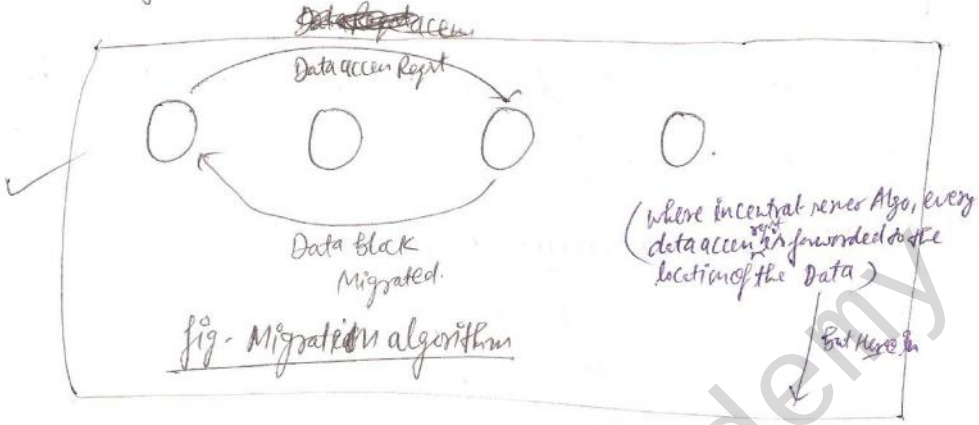



fig - Migration algorithm

**Operation** –

- Ship (Migrate) entire data object (page, block) containing data item to requesting location.
- Allow only one node to access a shared data at a time.

**Advantages** :-

- Takes the advantage of locality of reference.
- DSM can be integrated with virtual memory provided by the OS (Operating System) at individual nodes.
  - * Make DSM page multiple of VM page size.
  - * A locally held shared memory page can be mapped into the VM page address space.
  - * If page not local, fault-handler migrates page & removes it from the address space at remote node.
- To locate a remote data object :-
  - * use a location server.
  - * Maintain hints at each node.
  - * Broadcast query.
- Issues :-
  - * only one node can access a data object at a time.
  - * Thrashing can occur – to minimize it, set minimum time data object resides at a node.

→ (Processes on system require more memory than it has)
(Thrashing like high page activity)

## ③ The Read - Replication Algorithm →

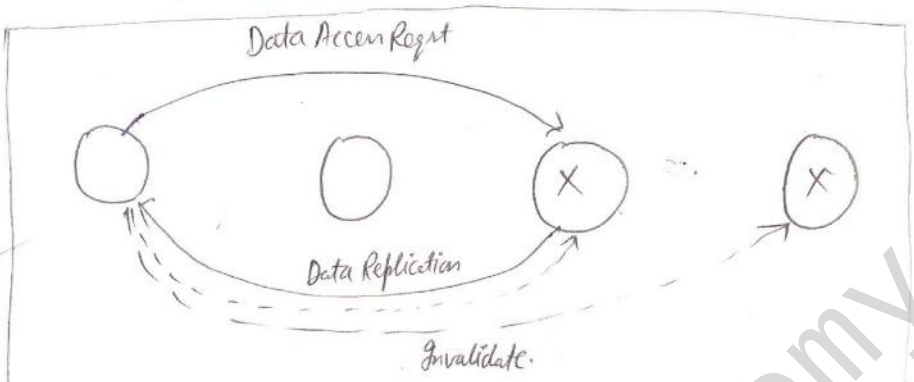



fig- write operation in Read - Replication Algo.

- In Previous approach, only processes on one node could access a shared data at any one Moment. The read-replication algorithm extends the Migration algo by replicating data blocks & allowing Multiple Nodes to have read access or one node have read-write access (the Multiple readers - one writer Protocol).

- Replicates Data objects to Multiple Nodes.
- DSM keeps track of location of Data objects.
- Multiple nodes can have read access or one node write access (Multiple Readers - one writer protocol)
- After a write, all copies are invalidated or updated the current value to maintain the Consistency of the Shared Data Block.
- DSM has to keep track of locations of all copies of data objects.

Example of implimentation (Integrate Shared virtual Memory at Yale)

* IVY :→ owner node of data object knows all nodes that have copies.
* PLUS :→ Distributed linkList tracks all Nodes that have copies.

Advantages

- The Read-Replication can lead to substantial performance improvements if the ratio of reads to write is large.

④ The Full-Replication Algorithm :→

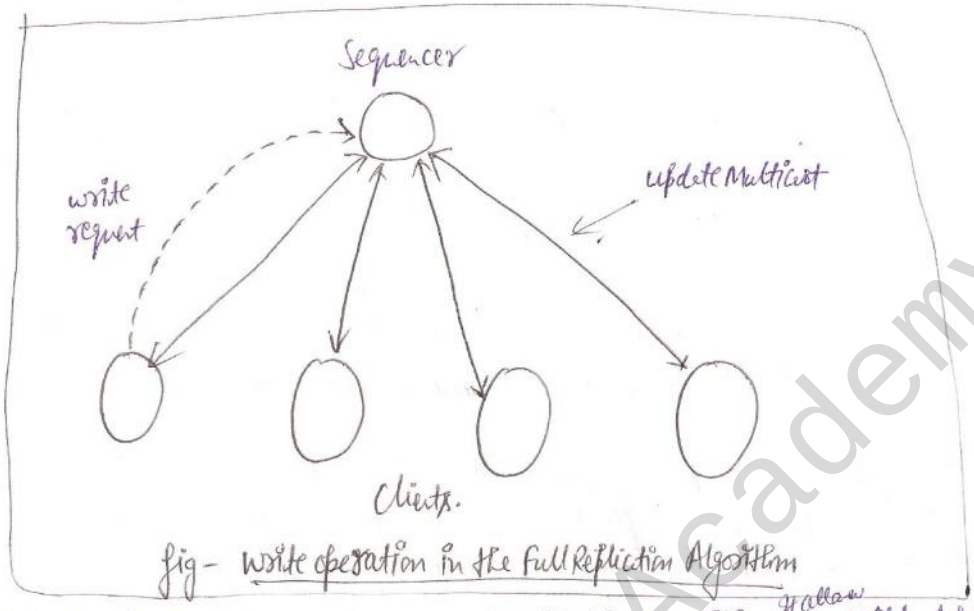



fig- write operation in the full Replication Algorithm

- This Algo is the, Extension of the read-replication Algorithm. Multiple Nodes (allow) can have both read & write access to shared data blocks (Multiple readers, Multiple writers protocol).
- Because Many Node can write shared data concurrently, the access to the shared data Must be Controlled to Maintain its Consistency.

Issues : Consistency of data for Multiple writers.

Sol$^{n}$ :

- All writes sent to Sequencer.
- The Sequencer will assigns (sequence No) & sends write request to all the sites that have copies. (If Multicast the Modification with the sequence No to all the Nodes that have a copies of shared data item)
- Each Node performs writes (Modification) according to sequence Numbers.
- A gap bet$^{n}$ the sequence No indicates that one or more Modification (write request missing) missed.
- Under such circumstances, Nodes will ask for retransmission of the Modification it has missed.

-Two important issues to be considered in the design of DSM system. They are important because the efficiency of DSM depends on the effectiveness of the size chosen for Granularity & the protocol used for Page replacement.

## Design Issues of Distributed Shared Memory-

1. **Granularity**- Refers to size of shared Memory unit.
2. **Page Replacement**

### 1. Granularity-

- A large page size for the shared memory unit will take advantage of the locality of reference.
  - By transferring large pages, less overhead is incurred due to paging activity and processing communication protocols.
- False sharing: occurs when two different data items, not shared but accessed by two different processes, are allocated to a single page.
  - More false sharing when the page size is large.
  - Smart compilers may partially solve the problem. However, if two processes share the same array, nothing can do about it.
- Another solution is to pre-fetch small pages.

### 2. Page Replacement-

- When there is no free space in the memory, a page may need to be replaced
  - Traditionally, we use Least recently used (LRU).
- In DSM, LRU may need to be modified, since data may be accessed in different modes such as shared, private, read-only, writable, etc.
  - Private pages may be replaced before shared pages, as shared pages would have to be moved over the network, from their owner.
  - Read-only pages can simply be deleted as their owners have a copy.
- Once a page is selected for replacement, the DSM must ensure that the page is not lost forever.
  - One option is to swap the page onto disk.
  - Another option is to use reserved memory, wherein each node is responsible for certain portions of the global virtual space and reserves memory space for those portions.

17

**IVY-**

- IVY (Integrated Shared Virtual Memory at Yale) was implemented in the Apollo environment.
- The address space is divided into pages, with pages being spread over all the processors in the system
  - When a processor references an address that is not local, a trap occurs, and the DSM software fetches the page containing the address and restarts the faulting instruction, which now competes successfully.
  - Use replication to improve performance, especially read.
- Achieving sequential consistency by write-invalidation.

18

**Sun Network File System-**

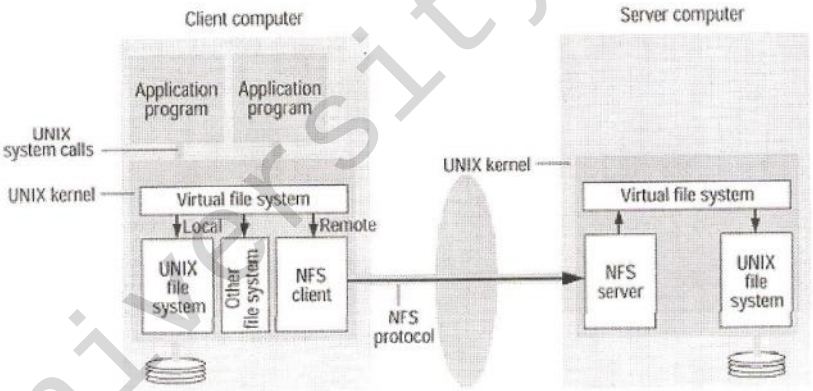
- Network file system (NFS) was developed by SUN Microsystem.
- It provides transparent access to remote files for client programs running on UNIX and other systems.
- Each computer has a client and server modules installed in its system kernel.
- Achieve a high level of support for hardware and OS heterogeneity.

**Virtual File System-**

- The access transparency is achieved by a virtual file system (VFS) module.
- VFS keeps track of the file systems that are currently available both locally and remotely, and it passes each request to the appropriate system.
- V-node contains a reference to show whether a file is local or remote
  - If the file is local, the v-node contains a reference to the index of the local file (i-node).
  - If the file is remote, it contains the file handle of the remote file. The file handle contains fields uniquely identifying the file system type, the disk, the i-node number of the directory, and security information.

57

## Network File System Architecture-



# Failure Recovery in Distributed System

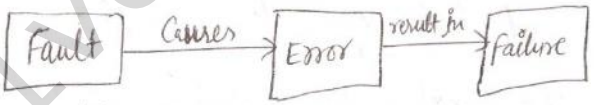
## UNIT-4

**Introduction :-** Recovery in Computer Systems refers to restoring a system to its normal operational state. Recovery may be as simple as restarting a failed computer or restarting failed processes. Recovery is a very complicated Process.

- In general, resources are allocated to executing processes in a computer. ex- a process has memory allocated to it and a process may have locked shared resources, such as files & memory. Under such circumstances, if a process fails, it is imperative (requiring attention or action) that the resources allocated the failed process be reclaimed (delivered from danger) so that they can be allocated to other processes.
  - If a failed process has modified the Database, then it is important that all the modifications made to the database by the failed process are undone.
  - If a process has executed for some time before failure, it would be preferable to restart the Process from the point of its failure & resume its execution.
  - By restarting from the Point of failure, the situation of having to reexecute the process from the begining is avoided, which may be a time consuming & expensive operation.

- Distributed systems provide enhanced performance & increased availability.
- one way to Relizing enhance performance is through the Concurrent execution of many processes, which cooperate in performing a task.
- If one or more cooperating Bocess fails, then the effect due to the interactions of the failed processes with the other processes must be undone, or every failed proces would have to restart from an appropriate state.
- Increased the availability in DS is realized mainly through replication (e.g. data, processes, & H/w, Computers can be replicated).
- If site fails, Copies of data stored at that site may miss updates, thus becoming in-consistent with the rest of the system when it becomes operational.
- Recovery in such Cases involves the question of how not to expose the system to data inconsistencies & bring back the failed site to an up-to-date state lead to failure & the types of failure.

Basic Concepts :->

- A System Consistents of a set of H/w & S/w components & is designed to provide a specific service
- The Components of a system may themselves be systems together with interrelationships.
- 


Fault -> is a defect within a system

Error -> is observed by a deviation from the expected behaviour of the system.

failure -> occurs when the system can no longer perform as required (does not meet specification)

Fault Tolerance -> is ability of system to provide a service, even in the presence of errors.

- From the above definitions, it can be seen that an error is a manifestation (expression) of a fault in a system, which could lead to system failure. failure recovery is a process that involves restoring an erroneous state (state which could lead to a system failure by sequence of valid state transition) to an error-free state.

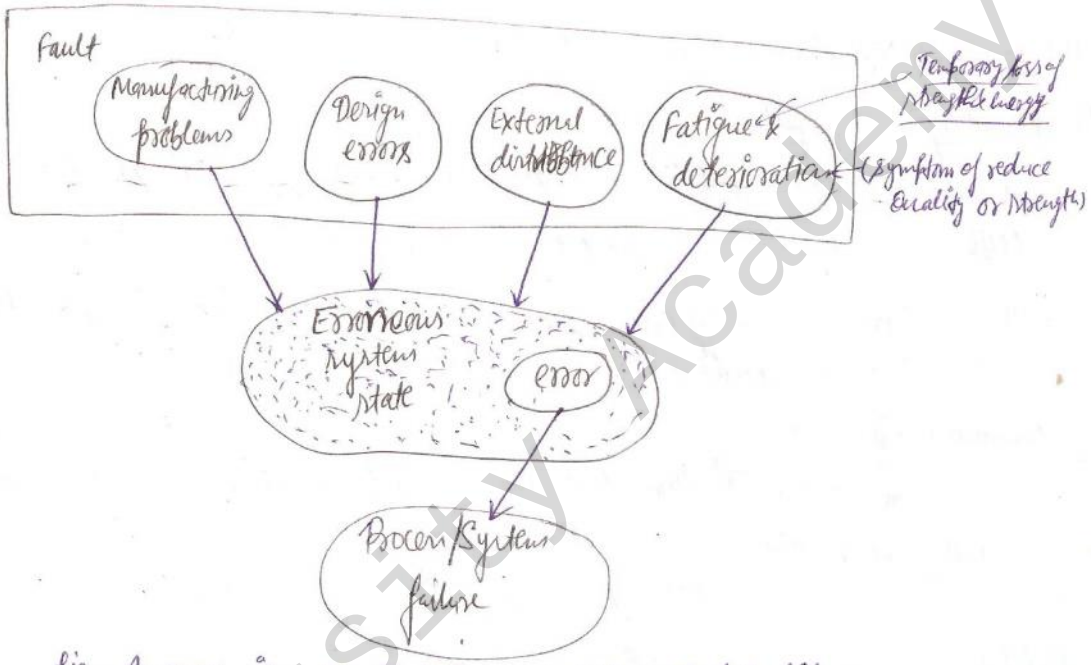



fig- An error is a manifestation of fault & can lead to failure.

Classification of failure:- Failure in a computer system can be classified as followed-

① Process failure → "The computation Results in an Incorrect outcome, the process causes the system state to deviate from specifications, the process may fail to progress." ex- (Deadlock)-

② System failure → System failure occurs when the processor fails to execute. (ex- S/w error, H/w Problems)

③ Secondary Storage failure → (SSF)

④ Communication Medium failure →

Tolerating/Recovery from SSF → {Archiving (Periodic Backup), Mirroring (continuous), activity logging}

→ Tolerating Com. Medium failure →
- → ack & resend
- → More complex fault Resistant Alg.

Recovering from System failure →
- "Restart in Predefined state" ← ① Amnesia failure
- "Reset the Part of the state to Predefined" → ② Partial-amnesia failure
- "Rollback to before failure" → ③ pause failure
- "give up" → ④ Halting failure.

(Partial or total loss of memory)

Amnesia failure :→ occurs when a system restarts on a predefined state that does not depend upon the state of the system before its failure.

Partial-amnesia :→ occurs when a system restarts in a state wherein a part of state in the same as the state before the failure & the rest of the state is predefined. (ex- file server has this type of failure)

Pause F :→ occurs when a system restarts in the same state it was before the failure.

Halting F :→ occurs when a crashed system never restarts.

(3) Secondary storage failure :→

- A secondary storage failure is said to have occured when the stored data (either some parts of it or in its entirety) cannot be accessed.
- This failure is usually caused by either parity error, head crash, or dust particles settled on the medium.

(4) Communication Medium failure :→

- A communication Medium failure occurs when a site cannot communicate with another operational site in the Net^k.
(ex- failure of switching Nodes)

## BACKWARD AND FORWARD ERROR RECOVERY :→

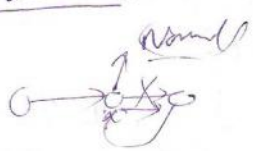
- Error is that part of the state that differs from its intended value & can lead to a system failure, & failure recovery is a process that involves restoring an erroneous state to an error-free state.
- There are two approaches for restoring an erroneous state to an error free state.
  - If the Nature of errors & damages caused by faults can be completely & accurately accessed, then it is possible to remove those errors in the process's (System's) state & enable the Process (System) to move forward. this technique is known as forward-error recovery. ~~(using check point)~~ (Correct the error)

- If it is not possible to foresee the Nature of faults & to remove all the errors in the process's (system's) state, then the process's (system's) state can be restored to a previous error-free state of the process (system). this technique is known as backward-error recovery. (~~correct the error~~) (use the check point) BR > FR

The Major problem associated with the backward-error recovery approach are →

- ✓ Performance Penalty:- The overhead to restore a process (system) state to a prior state can be quite high.
- ✓ There is no guarantee that faults will not occur again when processing begins from prior state.
- ✓ Some Component of the system state may be unrecoverable.
  ex- (Cash dispensed at an ATM can't be recovered)

## * BACKWARD ERROR RECOVERY :→



- It is based on Recovery Points.
- Two approaches are - (for implementing the Backward Recovery)
  1. operation based recovery.
  2. State-based Recovery.

System Model →
(Backward error Recovery)

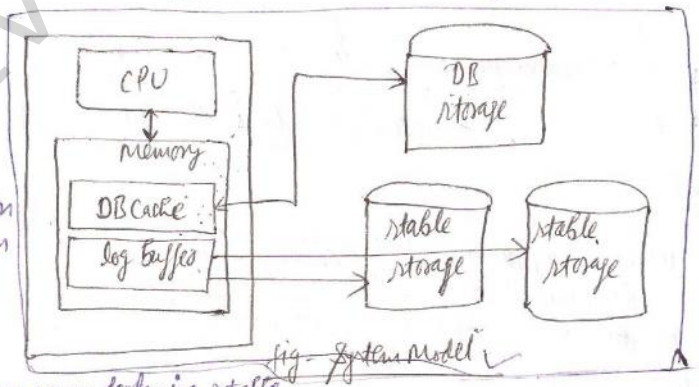



fig- System Model

Stable storage →
- Does not lose information in the event of system failure.
- Is used to keep logs & recovery points.
- Algorithm here we assume an underlaying stable storage system already exists.

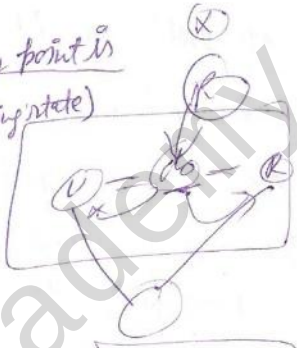
Two Approaches to fault Tolerance :-

① Operation Based Approach :- → UPDATING INPLACE SCHEME (write(update) recorded)
→ WRITE-AHEAD-LOG PROTOCOL (Both log is recorded) (redo, undo)

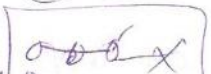
- Record a log (audit trail) of the operations performed.
- Restore previous state by reversing steps.

② State Based :- → Shadow Page (complete state of the process is saved when a recovery point is established.)

- Record a snapshot of the state (checkpoint) (the process of saving state)
- Restore state by reloading snapshot (rollback).
- use of Shado Pages can reduce size of checkpoints.



① Updating in Place :- under this scheme every update (write) operation to an object updates, the object & results, in a log to be recorded in a stable storage which has enough information to completely undo & redo the opr^n.



- The info recorded includes -
  - ✓ (1) the Name of the objects. (ID)
  - ✓ (2) the old state of the object (used for UNDO).
  - ✓ (3) the New state of the objects (used for REDO).

(Name/location of object) (old state/value of object) (New state/value of object)

DO - Operation which Does the actions writes a log
UNDO - gives log Record written by Do opr^n. Performed By Do opr^n
REDO -
DISPLAY - Specified by the Do opr^n → Display the log record.

② Write Ahead log Protocol :- In this a recoverable update opr^n is implemented by the following opr^n -

- ✓ Update an object only after the undo log is recorded.
- ✓ Before committing the updates, redo & undo logs are recorded.

- Both undo & Redo opr^n performed properly in both updating in placed write ahead log protocol.

① Shadow Page :- whenever a process wants to modify an object the Page Containing the object is Duplicated & is Maintained on the stable storage.

- ✓ The Process only updates one Copy. the unmodified copy is called the Shadow Page.
- ✓ If the Process fails, the Modified copy is discarded & the Shadow Page is used.
- ✓ If the Process Commits the Shadow Page is discarded.

## Recovery in Concurrent Systems :→ [IMP]

- In Concurrent Systems, several processes cooperate by exchanging information to accomplish a task. the information exchange can be through a shared memory in the case of shared memory machines (e.g multiprocessor systems) or through messages in the case of a distributed system.
- In such systems, if one of the cooperating processes fails & resumes execution from a recovery point, then the effects it has caused at other processes due to the information it has exchanged with them after establishing the recovery point will have to be undone.
- to undo the effects caused by a failed process at an active process, the active process must also rollback to an earlier state.
- Thus in Concurrent systems, all cooperating processes need to establish recovery points.
- rolling back processes in concurrent systems is more difficult than in the case of single process.

The following discussion shows how the rolling back of processes can cause further problems.

1. Orphan messages & the Domino Effect.
2. Lost Messages.
3. Problem of livelocks.

### ① Orphan Messages & the Domino effect :→

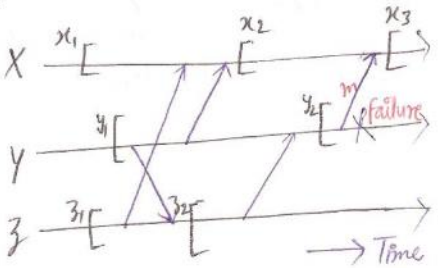


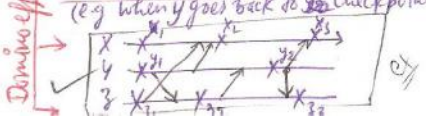

fig- Domino effect.

X, Y, Z → Processes (Cooperate by exchanging messages)

'[' → marks as a recovery point to which a process can be rolled back in the event of a failure.

Orphan messages & Domino effect -
In case receiving event is recorded in the checkpoint, but its sending event is not.

Domino effect → one rollback causes other to rollback (e.g when y goes back to checkpoint $y_2$)



- If Process X is to be Rolled back, it can be rolled back to the recovery point $x_3$ without affecting any other processes.
- Let, Y fails after sending message m & is rolled back to $y_2$. In this case, the receipt of m is recorded in $x_3$, but the sending of m is not recorded in $y_2$.
- Now we have a situation where X has received message m from Y, but Y has no record of sending it, which corresponds to an inconsistent state.
- under such circumstances, (m) is referred to as an orphan message & process X must also roll back.
- X must roll back because Y interacted with X after establishing its recovery point $y_2$. [Important]
- When Y is rolled back to $y_2$, the event that is responsible for the interaction is undone.
- Therefore, all the effects at X caused by the interaction must also be UNDONE.
- This can be achieved by rolling back X to recovery point $x_2$.
- Likewise, it can be seen that, if Z is rolled back, all three processes must roll back to their very first recovery Points, namely $x_1$, $y_1$ & $z_1$.
- This effect, where rolling back one process causes one or more other processes to roll back, is known as the domino affect & the orphan messages are the cause.

② Lost Messages :→

- A message whose sending event is recorded, but its receiving event is not recorded.

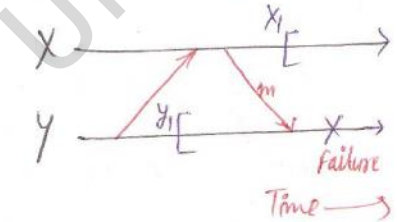
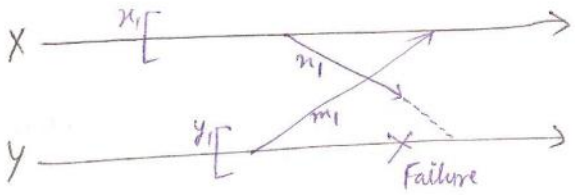


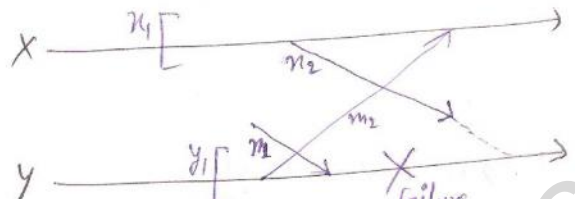

fig - Message loss due to roll-back recovery.

- Y fails & rolls back to $y_1$, & then X is in a state where it sent m but Y will never receive it.
- This situation can also happen if the communication channel is not reliable.

(3) Livelock :→ Livelock is a situation in which a single failure can cause an infinite No of rollbacks, preventing the System from Making Progress.




fig(a)




fig(b)

Fig -- Livelock ✓

- It is a situation in the Distributed System.

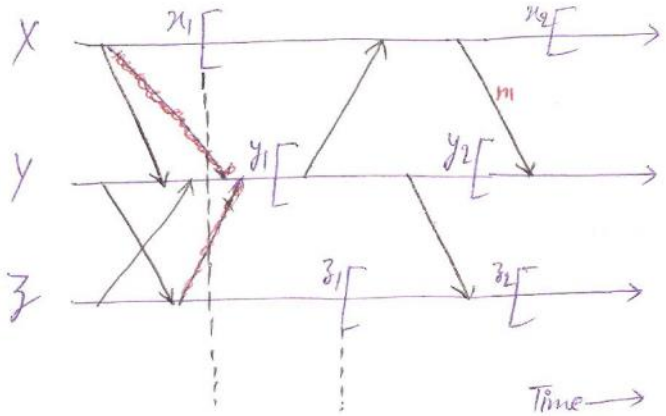
Consistent CheckPoints :→ Strongly consistency set of checkpoints / Consistent set of checkpoints (in Distributed System)

- A Process saves its local state on the stable storage, which is called a local checkpoint.
- The process of saving local states is called local checkpointing.
- All the local checkpoints, one from each site, collectively form a Global checkpoint.

(1) Strongly Consistent set of checkpoints :→

- The domino effect is caused by orphan messages, which themselves are due to rollbacks.
- To overcome the domino effect, a set of local checkpoints is needed (one for each process in the set) such that No info flow takes Place (No orphan Message) b/t any pair of processes in the set, as well as b/t any process in the set & any process outside the set during the interval spanned by the checkpoints.
- A set of check Points is Known as a recovery line or a strongly consistent set of checkpoints.

✓ A Global check Point is a strongly Consistent set of checkpoints if there is no orphan message & no lost Message.




$(x_1, y_1, z_1)$ → strongly Consistent Set of checkpoints

fig- Consistent set of ~~Reads~~ Checkpoints

- A Global checkpoint is a ~~strongly~~ Consistent set of checkpoints if there is no orphan message. $\{x_2, y_2, z_2\}$
- A Consistent set of checkpoints, ~~set~~ is similar to a Consistent global state in that it requires that each message recorded as received in a checkpoint (state) should also be recorded as sent in another checkpoint (state)

*A simple method ~~of~~ for taking a consistent set of checkpoints is

✓ If every process takes a checkpoints after sending every message, the set of the most recent checkpoints is always Consistent.

✓ However, it has a high overhead (expensive) that taking a checkpoint after every message has sent.

✓ So, one may attempt to reduce the overhead of in the above method by taking a ~~check~~ checkpoint after every K (K>1) message sent.

✓ This method suffers from domino effect.

✓ Assumption - Checkpoint, send/recv are atomic.
✓ Take a checkpoint after every message.
✓ the set of the most recent checkpoint is always consistent (why? Is it strongly Consistent)
✓ what is the main problem with this approach? Take a checkpoint after every K message sent? It is still Consistent?

Synchronous (IMP) checkpoint & Recovery :-> (Koo & Toueg) <- (1) The checkpoint Algo. (2) The Rollback Recovery Algo.

- A checkpointing & recovery technique proposed by Koo & toueg that takes a Consistent set of checkpoints & avoids livelock problems during recovery.
- The algorithm's approach is said to be synchronous, as the processes involved Coordinate their local checkpoint actions such that the set of all recent checkpoint in the system is guaranteed to be consistent.

(1) The checkpoint Algorithm :-> (Koo & Toueg)

The checkpoint Algorithm assumes the following characteristics for the Distributed System.

* Process communicate by exchanging messages through channels.
* Channels are FIFO in nature. End-to-End Protocols cope with message loss due to rollback recovery ((SWP) + Process have to log message in stable storage before sending them)
* Communication failures do not partition the network.

* This check Point Algorithm uses (2 kind of checkpoints) on stable storage.
  - (1) Permanent :-> (A local check Point at a process & is a part of a consistent Global checkPoint)
  - (2) Tentative :-> (A temporary checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint Algorithm).

This Algorithm has 2 Phase →

(1) Synchronous Checkpoint Phase-1 :->

(Initiator) :-> An initiating Process $P_i$ takes a tentative checkpoint & requests all the processes to take tentative checkpoints.

Other Processes : Can responds 'yes' or 'no'

Imp -> initiator :-> { Decides to make checkpoints permanent if every one has responded 'yes' } otherwise $P_i$ decides that all the tentative checkpoints should be discarded.

② Synchronous Checkpoint Phase-2 :->

- Initiator $P_i$ : informs all the processes of of Phase-1 (Decision at Phase-1) .
  (Commit or abort checkpoint)

- Others : act accordingly .

(Therefore either all or None of the processes takes permanent checkpoint.)

- Bet^n tentative checkpoint & Commit/abort of checkpoint process must hold back messages.

✓Synchronous checkpoint : Properties -

✓All or none of the Processes take permanent checkpoints.

✓ There is no record of a message being received but not sent.

Checkpoint may be taken unnecessarily e.g. (ex) → ✗ [Taking a checkpoint is an expensive thing]

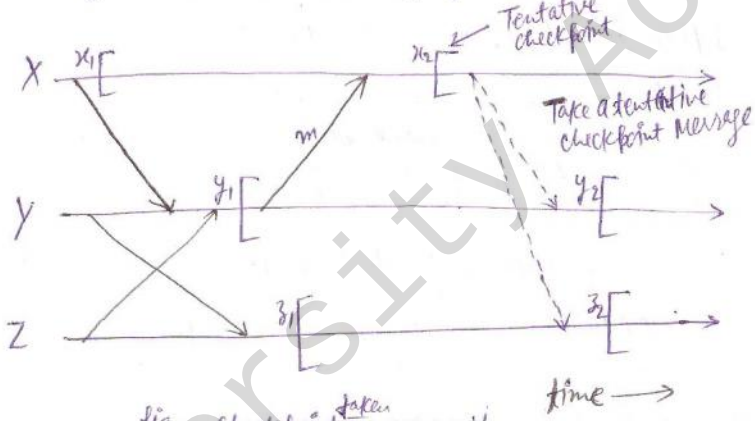



fig - Checkpoint taken unnecessarily

let,

- $\{x_1, y_1, z_1\}$ - Consistent set of checkpoint
- X decides to initiate the checkpoint after receiving message m.
- It takes the tentative checkpoint $x_2$ & sends "Take tentative checkpoint Message" to process Y & Z, causing Y & Z to take checkpoint $y_2$ & $z_2$ respectively.

Now, - $\{x_2, y_2, z_2\}$ - form a Consistent set of checkpoint.

- $\{x_2, y_2, z_1\}$ → Can also forms a Consistent set of checkpoint.

Can this unnecessary checkpoints to ~~a~~ avoided? A scheme is described as—

Main Idea → (Scheme)

* Record all Messages sent & received after the last check Point.
  - last_recv(x,y) -
  - First_sent(x,y) -

✓ * When X request Y to take a tentative checkpoint, X send the last message received from Y with the request.

✓ * Y takes tentative checkpoints only if the last message received by X from Y was sent after Y sent the first Message, after the last checkpoint → Page 310

$$(last\_recv(x,y) \geq first\_send(y,x))$$

✓ When a process takes a checkpoints, it will ask all other process that sent Messages to the processes to take checkpoint.

② The Rollback Recovery Algorithm :-

- It assumes that single process invokes the algorithm, as opposed to several processes concurrently invoking it to rollback & recover.
- It also assume that checkpoint & rollback recovery algorithm are not currently invoked.
- The rollback algo has two Phase — P-1, P-2

① Rollback Recovery : Phase (1) →

- Initiator (Pi) : check whether all processes are willing to restart from the checkpoints.
- ("May be yes" or "No") (A process may reply "no" to a restart req if it is already participating in a checkpointing or a recovering process initiated by some other process.)
- (If "yes" → by all Process to Pi, Pi decides to ~~restart~~ All process restart otherwise Pi will decide that the Process should continue with their ~~op~~ Normal operation).

② Rollback Recovery : Phase (2) →

- Initiator : Propogate go/nogo decision to all processes.
- others : Carry out the decision of the initiator.

Bet$^{n}$ Request to Rollback & decision, no one sends other Messages.

Rollback Recovery: Properties →

- All or none the Processes restart from checkpoints.
- After rollback, all process resume a consistent state.

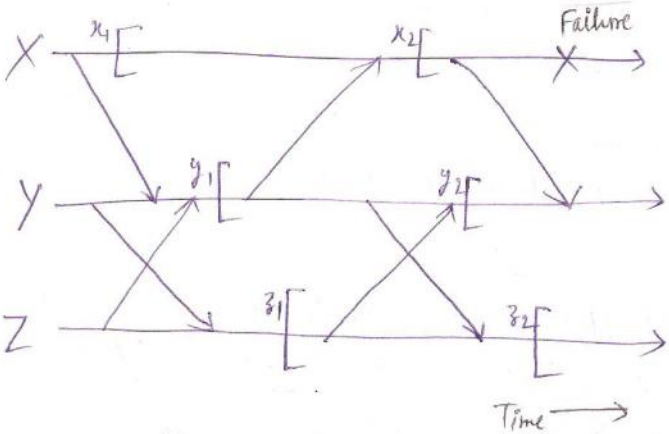



fig- Unnecessary rollback (Problem)

- In the event of failure of process X, would require processes X, Y, Z to restart from checkpoints $x_2, y_2, z_2$ respectively.
- However, that process Z need not have rolled back as there was no interaction bet$^n$ Z & the other two processes.

(To minimize) the No of process (rollbacks), A scheme is defined -

Main Idea - for Any two Process x, y. (m, mess)

- last_sent {x, y} → (X sent to Y, before X taken its latest permanent checkpoint)
- last_recvd {y, x} → (y will restart from its Permanent Checkpoint only if the condition holds.
- last_recvd {y, x} > last_sent {x, y} ← when X reqst to Y to checkpoint

Disadvantage of Synchronous Approach :→

- Checkpoint Algorithm generates message traffic.
- Synchronization delays are introduced.

these cost may seems high if failures bet$^n$ checkpoints are unlikely.

(X - Req$^t$ Y to restart from the Permanent checkpoint) at level


$i \geq j$

($\perp$ → Smallest level
$\top$ → largest level) → Process's level

## Asynchronous checkpointing And Recovery :-

- Takes multiple local checkpoints independently. (at each processor) without any synchronization among the processes. Because of the absence of synchronization there is no guarantee that a set of local checkpoints taken will be a Consistent set of checkpoints.
- After a failure, try to find a consistent set of checkpoints among those that have been taken recently.
- All incoming message bet$^n$ local checkpoints are logged. (message received can be logged in 2 ways) → (1) pessimistic approach: log each message before processing.
  (2) optimistic approach: Buffer messages & log in batches.

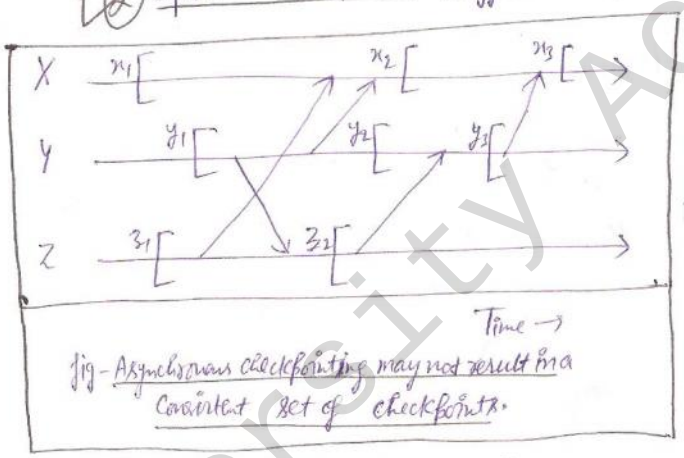



fig - Asynchronous checkpointing may not result in a Consistent set of checkpoints.

✓ $\{x_3, y_3, z_2\}$ → is not consistent.

✓ $\{x_2, y_2, z_2\}$ → is the most recent consistent set of checkpoints.

(GS) → Consistent global state. (for every received message a corresponding Send event is recorded in the Global state)

✓ A Scheme for Asynchronous Checkpointing & Recovery :- [Juang & S. Venkatesan]

Algorithm makes the following assumptions →

* Communication channels are reliable.
* Communication channels are FIFO.
* Communication channels have no Buffer limits (size).
* message transmission delay is bounded. [finite].
* underlaying Computation is event-driven, but locally timestamped event. (increasing no)

Each event consists of the following - waiting for a message, process the message, change process state, & send a no of messages

Asynchronous Check Point :->

Two types of log storage are assumed to be available for logging in the System.

1. Volatile log. (accessing volatile log takes less time than accessing stable log, but contents of the volatile log are lost if corresponding processor fails)
2. Stable log.

(the contents of volatile log are periodically flushed to the stable storage & cleared.)

Basic Idea :->

* At each event, a triplet { S, m, msgs_sent } is put in the log.

{ S → state
m → message causing the event
msgs_sent → is the set of message sent. }

* Two data structures used :
  - RCVD (i, j, checkpoint) - the No of message received by processor i from processor j at checkpoint.
  - SENT (i, j, checkpoint) - the No of message sent from i to j at checkpoint.

* use the message send/recv counts to determine the point to rollback.

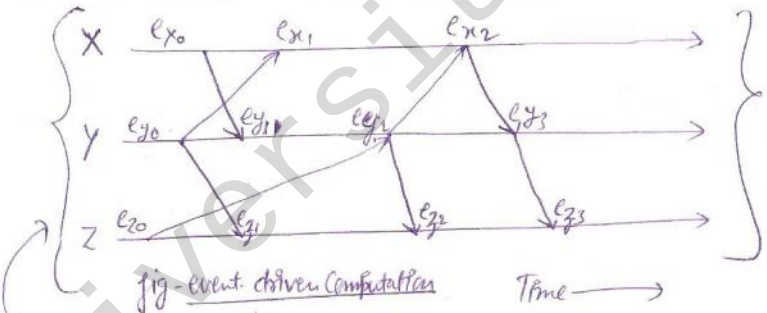



fig - event driven Computation

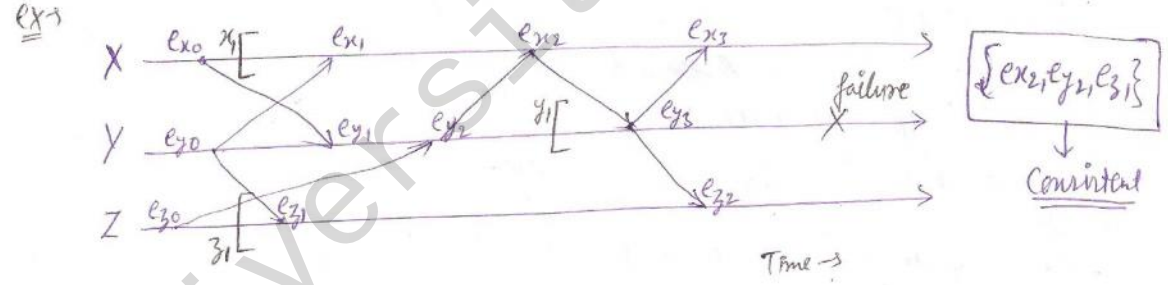
- If Y rolls back to a state corresponding to $e_{y_1}$, then according to this state Y has sent only one message to X. according to X's state, however, it has received two messages from Y thus far, therefore, X has to roll back to a state preceding $e_{x_2}$ to be consistent with Y's state. for the similar reasons, Z will also have to roll back.

meanwhile ($e_{x_1}$)

Algorithm:-

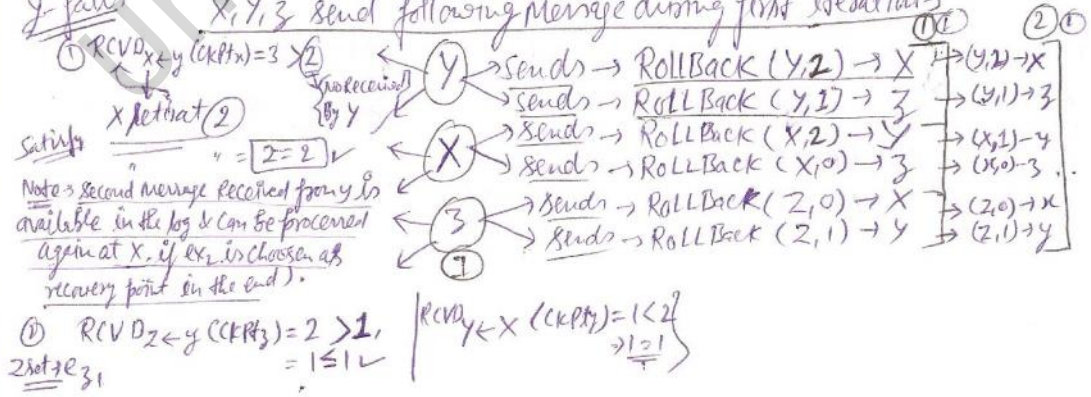
- All Node will be running the same recovering algorithm at each processor (i).
- If I is a processor that is recovering from failure, [checkpoint = the latest event logged in the stable storage.]
- else checkpoint = latest event that took Place.
- for K = 1 to N do — (N → No of processors in the system)
  - send [ROLLBACK (i, SENT (i, j, checkPoint))] to all neighbours j.
  - wait for [ROLLBACK messages] from all neighbours
  - for every ROLLBACK (j, c) received —
    - if (RCVD (i, j, checkpoint) > c) then (Presence of orphan message)
      - * find the latest event e such that RCVD (i, j, e) = c
      - * checkpoint = e

→ In each iteration, at least one processor will rollback to its final recovery point unless current recovery point is consistent.




If Y fails → Initially X, Y, Z set their CheckPoint at → $e_{x3}, e_{y2}, e_{z2}$

X, Y, Z send following message during first iteration →




① RCVD Z←y (CKPTy) = 2 > 1, 2 set → $e_{z1}$

RCVD Y←X (CKPTx) = 1 < 2 → 1 = 1

## Checkpointing for Distributed Database Systems :→

- In this we discuss on a technique for taking checkpoints in a distributed Database System (DDBS) where a set of data objects is partitioned among several sites.
- A checkpointing scheme for a DDBS should meet the following two basic objectives-
  1. Checkpoints should be highly derivable.
  2. All sites should take local checkpoints recording the state of the local database.

Issues →

1. How sites decide or agree upon update of what transactions are to be included in their checkpoints.
2. How each site can take a local checkpoint in a noninterfering way.

Son & Agrawala's Algorithm for checkpoint in a DDBS :→ (Noninterfering & takes globally consistent checkpoints).

Assumptions made by Algo are →

* The basic unit of user activity is a transaction.
* The Transactions follow some concurrency control Protocol.
* Lamport logical clocks are used to associate each transaction with a timestamp.
* Site failure are detectable either by net. protocols or by time out Mechanisms.
* Network Partitioning never occurs.

Basic Idea →

- All Participating sites agree upon a special timestamp known as the Global checkpoint Number (GCPN).
- The update of the transactions, which have timestamp ≤ GCPN, are included in the checkpoint.
- These transactions are called Before-checkpoint transaction (BCPTs).
- The update of the transaction which have timestamp > GCPN are not included in the Checkpoint.
- these transcations are called after-checkpoint transaction (ACPTs).

→

→ To avoide interfering with the normal operations while checkpointing is in progress, each site maintains multiple versions of data items in volatile storage that are being updated by ACPTs.

→ Thus the state of the database is not disturbed once all the BCPTs terminate (at which time the database is Consistent) until checkpointing completes.

DATA STRUCTURE:→

This Algo requires that each sites Maintains following variables—

* LC: The local clock maintained as per Lamport's logical clock rules
* LCPN: A No Determined locally for the current checkpoint.

Algorithm:→

- The Algo is initiated by a special process known as the checkpoint coordinator (CC).
- It takes a Consistent set of checkpoints with the help of processes known as checkpoint Subcordinators (CS), running at every participating site.

Phase-1 :→

(1) The checkpoint coordinator Broadcasts a checkpoint_Reqst message with local timestamp $LC_{cc}$.

(2) $LCPN_{cc} = LC_{cc}$

(3) $CONVERT_{cc}$ = false (use of convert will become clear later)

(4) The checkpoint coordinator waits for replies (obtaining LCPN's) from all the Subordinate sites.

At all the checkpoint Subordinates (CS) Sites →

(1) On receiving a checkpoint_Reqest message, a site m, updates its local clock as follows—

$$LC_m = MAX\{LC_m, LC_{cc} + 1\}$$

(2) $LCPN_m = LC_m$

(3) site m informs $LCPN_m$ to the checkpoint Coordinator.

(4) $Convert_m$ = false

(5) site m masks all the transactions with timestamp $\leq LCPN_m$ as BCPT, & mask rest of the transactions a temporary — ACPT.

Phase-2

At the checkpoint coordinator site:→

- Once all the replies for the checkpoint-request messages have been received, the coordinator broadcast GCPN, which is –

$$GCPN = MAX\{LCPN_1, LCPN_2 \cdots LCPN_n\}$$

(n → The No of sites in the system)

At all sites –

① On receiving GCPN, a site m marks all temporary ACPTs which satisfy the following condition as BCPT –

$$LCPN_m < \text{transaction's timestamp} \leq GCPN$$

the update of these transactions, newly converted as BCPTs, are also included in the checkpoint.

② $Convert_m = true$. When $Convert_m = true$, ~~since it taken a tentative checkpoint~~ it indicates that GCPN is known and all BCPTs have been identified.

③ When all BCPTs terminates & $Convert_m = true$, site m takes a local checkpoint by saving the state of data object.

④ When all the local checkpoint is taken, the database is updated with the commited temporary versions & then the commited temporary versions are deleted.

Note – if a site m receives a new "initiate transaction" message for a new transaction whose timestamp is $\leq GCPN_m$ & the site m has already executed steps 1 & 2 of Phase 2, then site m reject the "initial transaction".

## Recovery in Replicated Distributed Database System (RDDBS)

- To enhance the performance & availability, a distributed database system is replicated where copies of data objects are stored at different sites.
- Such a system is known as RDDBS. In RDDBS, transaction are allowed to continue despite one or more site failure as long as one copy of the Database is available.
- The copies of the database at the failed site may miss some updates while the sites are not operational. these copies will be inconsistent with the copies at the operation sites.
- The Goal of Recovery algorithm in RDDBS is to hide such inconsistencies from user transactions, bring the copies at recovering sites up-to-date with respect to the rest of the copies, & enable the recovering sites to start processing transactions as soon as possible.
- Two approaches have been proposed to recover failed sites-
  1. Merge spooler - used to save all the updates directed towards failed sites.
  2. Copier transactions - Read the up-to-date copies at the operational sites & update the copies at recovering sites. It runs concurrently with user transaction.
- The Recovery algorithm should Guarantee that-
  1. The out-of-date replicas are not accessible to user transactions. &
  2. once the out-of-date replicas are made up-to-date by copies transaction, they are also updated along the other copies by the user transactions.

An Algorithm for Site Recovery :→ [Bhargava & Ruan] → Based on Copier transaction

System Model→

- The Database is assumed to be manipulated through transactions whose access to the Database is controlled by Concurrency Control algorithm.
- Transactions either run to completion or have no effect on the database.
- The semantics of read & write operations on the Database are such that a read operation will read from any available copy & write operation updates all the available copies.
  - All the out-of-date copies in the database are assumed to be marked as "unreadable".
  - we also assume that database is fully replicated (every site has a copy of database).
  - A site may be any one of the following States—

(1) Operational / up :→ The site is operating normally & user transactions are accepted.

(2) Recovering :→ The Recovery is still in progress at the site & the site is not ready to accept user transaction.

(3) Down :→ No RDDBS activity can be performed at the site.

(4) Non operational :→ The site's state is either recovering or down.

- The operational session of a site is a time period in which the site is up. each operational session of a site is designated with a session no (an integer) which is unique in the site history, but not necessarily unique systemwide.
- The session No are stored on nonvolatile storage so that a recovering site can use an appropriate new session No.

DATA STRUCTURES:→

- Each site K maintains the following 2 data structures-

① The session No of site K is maintained in a variable $AS_K$. $AS_K$ is set to zero when site K is nonoperational.

② $PS_K$ is a vector of size n, where n is the No of sites in the system.

$PS_K[i]$ is the session no of site i as known to site K. $PS_K[i]$ is set to zero whenever K learns that site i is down or some other site informs K that site i is down.

How the System functions under the Normal, failure & recovery Conditions-

① User transaction →

- Each request that originates at a site i for reading or writing a data item at site K carries $PS_i[K]$.
- If $PS_i[K] \neq AS_K$ or $AS_K = 0$, then the request is rejected by site K. Otherwise there are 3 possibilities-
  ① the data item is readable - the request is processed at site K.
  ② the data item is marked unreadable & the oper$^n$ is a write operation - (the data item is modified & will be marked readable when the transaction commits.)
  ③ the data item is marked unreadable & the operation is a read operation - (a copier transaction is initiated by site K).

② Copier transaction :→

- Copier transaction may be initiated for all the data items marked as unreadable when a site starts recovering.
- On the other hand, a copier transaction may be initiated on a demand basis that is, whenever a read operation is received for individual data items marked unreadable.

③ Control transaction :->

- Control transactions are special transactions that update AS & PS at all sites (including any recovering sites).
- When a recovering site (say K) decides that it is ready to change its state from recovering to operational, it initiate a type-1 control transaction.
- type-1 transaction performs the following operations—
  - (1) It reads $PS_i$ from some reachable site i & refreshes $PS_K$.
  - (2) It chooses a new session No, sets $PS_K[K]$ to this new session no, & writes $PS_K[K]$ to all $PS_i[K]$ where $PS_K[i] \neq 0$.
    [all the sites perceived up by site K]
- When a sites discovers that one or more sites are down, it initiates a type-2 control transaction. (ex- if site K learns that site m & n are down, then it initiates a type-2 control transaction which performs following oprn.
  - (1) It sets $PS_K[m]$ & $PS_K[n]$ to zero.
  - (2) For all i such that $PS_K[i] \neq 0$, it sets $PS_i[m]$ & $PS_i[n]$ to zero.

The SITE RECOVERY PROCEDURE :->

- When a site K restarts from failure, the recovery procedure at site K performs the following steps.
  1. It sets $AS_K$ to zero. that is site K is recovering & is not ready to accept user transactions.
  2. It Marks all the copies of data item unreadable.
  3. It initiate a type-1 control transaction.
  4. If the control transaction of step-3 successfull terminates, then the site copies the New session No from $PS_K[K]$ to $AS_K$. [one $AS_K$ is $\neq 0$, the site is ready to accept user transaction].
  5. If step 3 fails because of discovering that another sites has failed, site K initiate type-2 control transaction to exclude the newly failed site & then restarts from step-3.

# Fault Tolerance UNIT-4 PART-2

System can be designed to be fault-tolerant in two ways →

① Mask failures, continue to perform, in degraded mode.
* Basic mechanism : redundancy.

② Or exhibit a well defined failure behaviour : may not perform the specified function, but facilitate actions suitable for recovery.
* e.g. changes made to a database by a transaction are made visible to other transactions only if the transaction successfully commits.

Issues in Fault Tolerance :→ (we must study the implication of certain types of failures)

① - Process Death - { Server/client Process's must inform to each other, if one dies after sending a reply to other).
- Must recover resources.
- Must undo resultant blockages. { when a process dies it is important to that the resources allocated to that process are recovered, otherwise they may be permanently lost.

② - Machine failure -
→ In the case of M/c failure all the processes running at the M/c will die.
→ The only difference bet" Process death & M/c failure, lies in how the failure is detected.

③ - Network failure -
* A communication link failure can partition a network into subnet, making it impossible for a m/c to communicate with another M/c in a different subnet.
* usually a fault-tolerant also assume that M/c may be operating & processes on that M/c are active.

Atomic Actions :→
- Appear to other processes as if they were -
  - indivisible.
  - instantaneous.
  - Serializability :→ the effect is the same as if atomic actions are interleaved, not concurrent.

Transaction →
- A set of sequence of actions that is implemented as if atomic.
- Commit → Guarantee transaction will be completed, so its effects are permanent.
  * Implemented by Commit protocols.
- Abort → guarantee to Back out of the transaction, so none of its effect persist.

(for backing out ↓ (using write ahead log Protocol or shadow page)

A transaction may abort due to following events →
* deadlock
* timeout
* protection violation
* Input data error
* Consistency violation.

Example :-> X - Sharable Resources -

| Process $P_1$ | Process $P_2$ |
|---|---|
| Lock (X); | Lock (X); |
| X = x + z; | X = x + y; |
| unlock (X); | unlock (X); |
| failure | |

$P_1$ should be atomic.

fig - Process iteration

| Process $P_1$ | Process $P_2$ |
|---|---|
| lock (x); Lock (y); | Lock (x); Lock (y) |
| Temp = X; | Temp = x; |
| X = Y; | X = Y; |
| Y = Temp; | Y = temp; |
| unlock (y); unlock (x); | unlock (y); unlock (x); |

Commit Protocols :-> [Gray] (2 Phase Commit Protocol)

- Commit Protocols are used to ensure atomicity across sites.
  - A transaction which executes at multiple site must either be committed at all the sites, or aborted at all the sites.
  - Not acceptable to have a transaction committed at one site & aborted at another.
- the 2-Phase Commit (2PC) Protocol is widely used.
- the 3-Phase Commit (3PC) Protocol is more complicated & more expensive, but avoides some drawbacks of 2PC Protocol. (This Protocol is not used in Practice).

2-Phase Commit :→

- A way to guaranteeing all-or-none agreement to commit a distributed transaction.

General's Paradox :→

This is an abstraction of a core problem in the design of commit Protocols.

- generals must attack simultaneously to capture hill.
- Some messengers may be killed or captured.
- the challenge is to use a protocol that allows the generals to agree on a time to attack, even though some messengers do not get through.

2-Phase Commit Protocol :→

This Protocol assume as →

* one of the cooperating process acts as a Coordinator.
* Other processes are refered to as Cohorts.
* Stable storage is available at each site.
* each site uses write-Ahead log Protocol, to achieve local fault recovery & rollback.
* At the begining of the transaction, the coordinator sends a start transaction message to every Cohort.

Phase-1→

At the Coordinator :→

① The Coordinator sends a COMMIT-REQUEST message to every Cohort requesting the the Cohorts to commit.

② the coordinator waits for replies from all the Cohorts.

At Cohorts :→

① On receiving a COMMIT-REQUEST message, a cohort takes the following actions.

- If the transaction executing at the Cohort is successful, it writes UNDO & REDO log on the stable storage & sends an AGREED message to the Coordinator.
- Otherwise, it send a ABORT message to the Coordinator.

Phase-2

(A) At the Coordinator :-

① If all the cohorts reply AGREED & the Coordinator also agrees, then the Coordinator writes a COMMIT record into the log. Then it sends a COMMIT message to all the cohorts. Otherwise the Coordinator sends an ABORT message to all the Cohorts.

② The Coordinator then waits for acknowledgements from each Cohorts. (Commit Message ACK)

③ If an acknowledgement is not received from any cohort within a timeout period, the Coordinator resends the Commit/Abort message to that cohort.

④ If all the acknowledgements are received, the Coordinator writes a COMPLETE record to the log (to indicate the completion of the transaction).

(B) At the Cohorts :-

① On receiving a COMMIT message, a cohort releases All the resources & locks held by it for executing the transaction, & sends an acknowledgement.

② On receiving a ABORT Message, a cohort undoes the transaction using the UNDO log Record, releases all the resources & locks held by it for Performing the transaction, & sends an acknowledgement.

- When there are no failure or message losses, it is easy to see that all sites will Commit only when all the participating (including the Coordinator) agree to Commit.
- In the case of lost messages (sent from either Cohorts or the Coordinator), the Coordinator simply resends messages after the timeout.

Site failure :->

(1) The Coordinator Crashes before the COMMIT record :-> on recovery, the Coordinator broadcast an ABORT message to all the Cohorts. All the Cohorts who had agreed to commit will simply undo the transaction using the UNDO log & abort. Other Cohorts will simply abort the transaction. all the Cohorts are blocked until they receive an ABORT Message.

(2) - The Coordinator Crashes after the COMMIT record but before COMPLETE :-> on recovery, the Coordinator broadcasts a COMMIT message to all the Cohorts & waits for acknowledgements. In this case also the Cohorts are blocked until they received a COMMIT message.

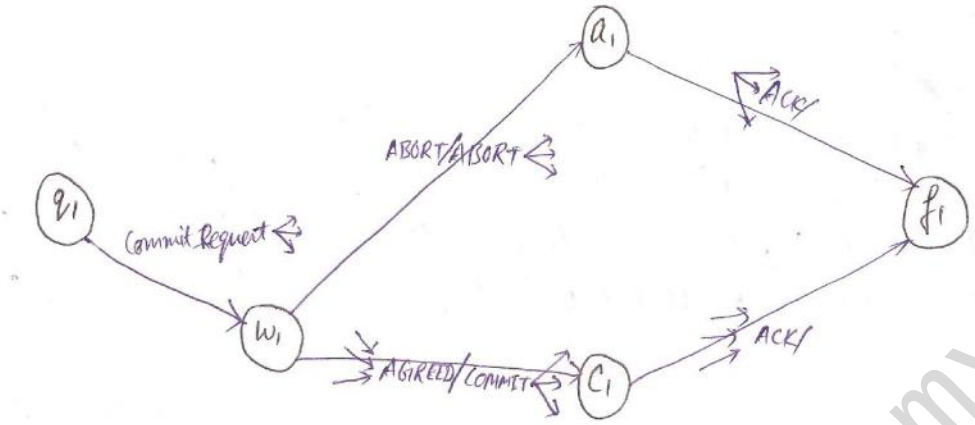
(3) - A coordinator crashes after COMPLETE :-> on recovery, there is nothing to be done for the transaction.

(4) - A Cohort crash Befor reply (in Phase 1) :-> the coordinator can abort the transaction because it did not receive reply from the crashed Cohort.

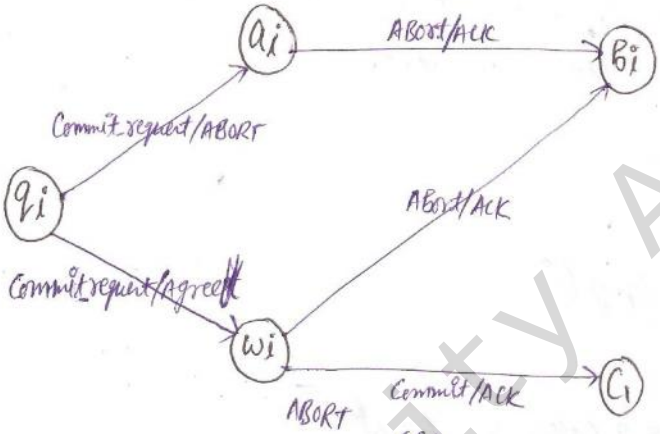
(5) - A Cohort Crash in Phase-2 (after UNDO/REDO log) :-> on recovery, the Cohort will check with coordinator whether to abort (i.e Perform an undo operation) or to commit the transaction. Note that commiting may require a redo operation because the Cohort may have failed Before updating the Database.

2-Phase Commit Protocol -

- guarantees global atomicity (its Biggest drawback is that it is a Blocking Protocol)
- if Coordinator fails all sites block until coordinator recovers




(a)

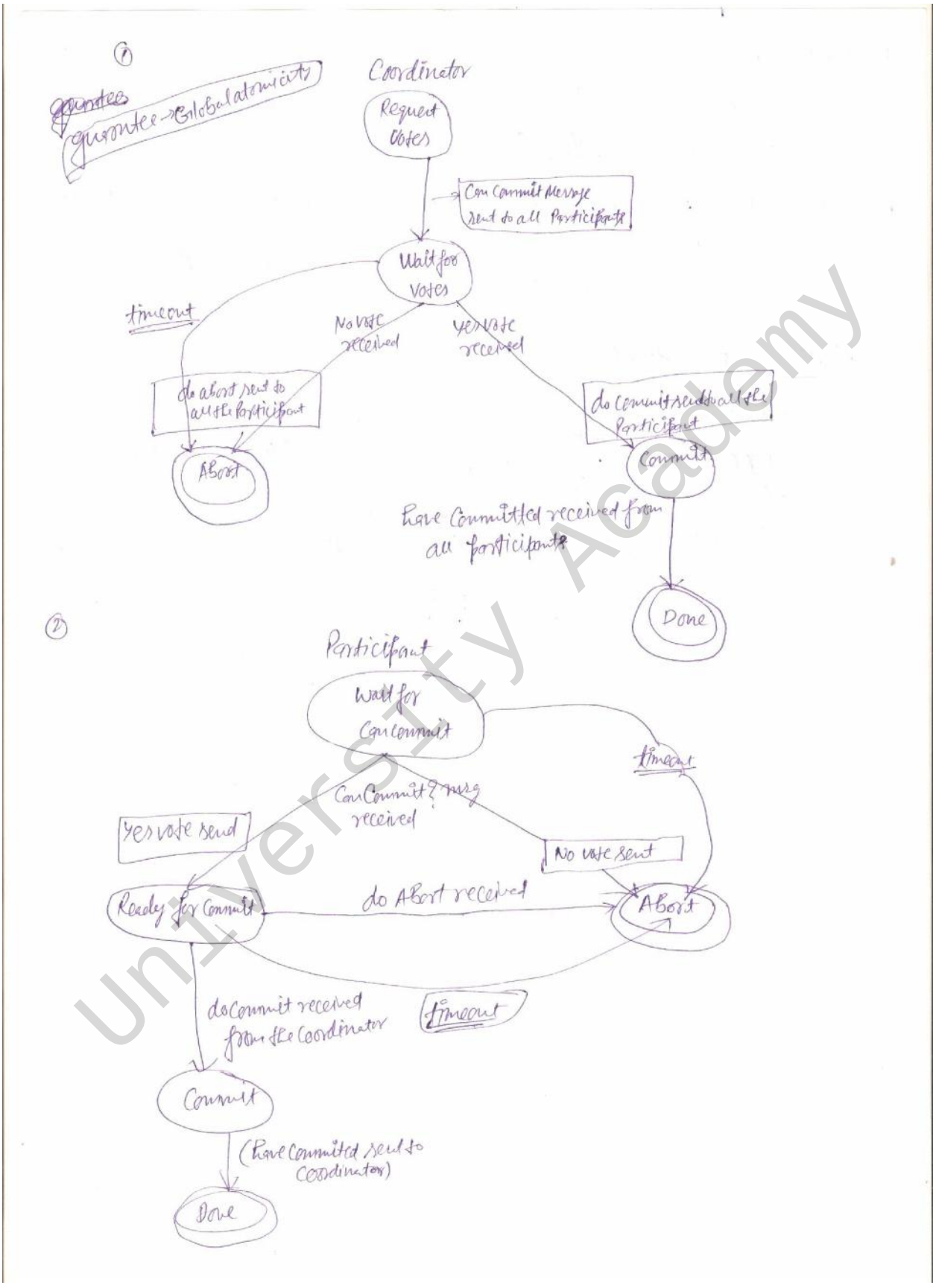



(b)

→ A level of the form "A/B" on a transition edge indicates that the transaction is taken when the set of message received matches the specification A, & causes the set of message specified by B to be taken.

→ "$\Rightarrow$ M" represents the receipt of M from all other sites.

→ "M $\Rightarrow$" represents the sending of M to all other sites.



## Voting Protocols:-

- Replicated data, at Multiple sites.
- Each site has some number of votes.
- access to replicated data requires a majority of votes.
- Votes determine which version is the current one.
- Voting mechanism is more fault-tolerant than a commit protocol in that it allows access to data under network Partitions, site failures, & Message losses without compromising the integrity of the data.

## Static Voting:- (Gifford)

- replicas of files (data), are stored at Multiple sites.
- Each files access operation requires that an appropriate lock is obtained.
- reader-writer locks are supported. { one writer no reader / Multiple reader no writers } to access the file simultaneously.
- Every site has a lock manager that performs the lock related operation.
- Every file has a version No = No of time the file has been updated (No of changes made) (version No are stored in stable storage)
- Every replica has some no of votes.
- Votes allocation is on stable storage.
- reads & writes require a quorum. (A minimum No of Vote that Must Present at a Meeting to make it valid)

## The Static Voting Algorithm:-

- When a process executing at site i issues a read or write request for a file, the following protocol is initiated.

1. Issue LOCK_Request to local lock manager.
2. local lock manager eventually grants requests & then sends Vote_Request to all other sites.

At site $j$ :-

(3) On receipt of (Vote Request from $i$), issue (Lock Request to local lock Manager.

(4) If the lock request is granted by the local lock Manager, send the (version NO ($VN_j$)) of its (replica) & the no of (votes $V_j$) of its replica to site $i$.

At site $i$ :-

(5) after votes are in, perform quorum test.

* Read quorum test :-

$$V_r = \sum_{K \in P} V_K \geq r$$

Where P = the set of sites that Replicated.

If $V_r \geq r$, where $r$ = read quorum, then site $i$ has succeeded in obtaining the read quorum.

* write quorum test →

$$V_w = \sum_{K=Q} V_K \geq w$$

If $V_w \geq w$, where $w$ = write quorum, the site $i$ has succeeded in obtaining the write quorum.

Set of site Q is determined as follows -

$M = \max \{ VN_j \mid j \in P \}$ is the largest version No reported in the vote, &

$Q = \{ j \in P \mid VN_j = M \}$ includes only the votes that correspond to that version Number.

(5) If quorum test fails, issue Release_lock to local manager & all sites in P that returned positive votes.

(6) If quorum test succeeds, check whether local copy is current, if not, obtain a fresh copy from another site.

(7) for a read, just use the local copy.
for a write, update the local copy.
- then update $VN_i$ & send the updates & $VN_j$ to all the sites in Q.

(8) Issue a Release_lock to local manager & all the sites in P.

at other sites:

- on receiving update messages, update own local copies.
- on Relase_lock, release all local locks.

Vote assignment:

If v is the total No of votes, we want to choose r & w such that

| |
|---|
| $r + w > v$ & |
| $w > v/2$ |

r → Read quorum.
w → write quorum.

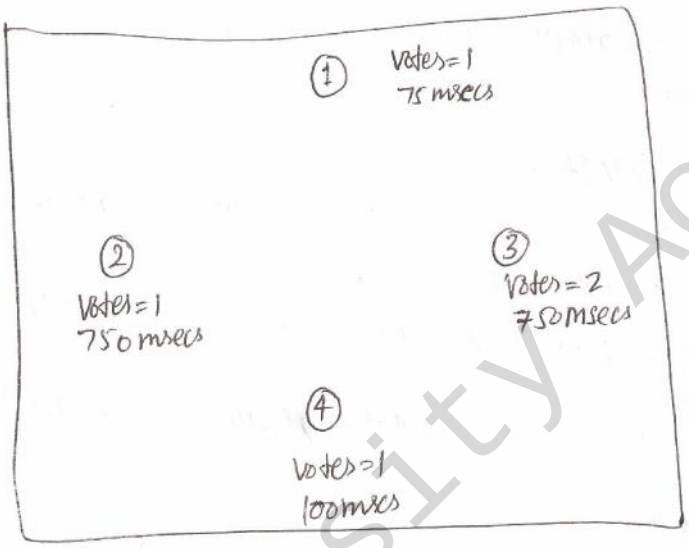



fig- vote assignment

$r=1$, $w=5$
$r_{msecs} = 75$ msec
$w \Rightarrow 750$ msecs
Read access time = 70 msecs (RAT)

$r=3$
$w=3$
RAT = 75 msecs

| Sites | Votes | Read Access time |
|---|---|---|
| 1 | 1 | 75 ms |
| 2 | 1 | 750 ms |
| 3 | 1 | 750 ms |
| 4 | 1 | 100 ms |

If $r=1$, $w=5$ RAT = 75 msec, WAT = 750 ms

If $r=3$, $w=3$ the times are unchanged, But writes are still possible

If site = 4, is more reliable we can further improve reliability by readjusting the votes as follows

| S | V | RAT |
|---|---|---|
| 1 | 1 | 75 ms |
| 2 | 1 | 750 ms |
| 3 | 1 | 750 ms |
| 4 | 2 | 100 ms |

**Dynamic Voting Protocols** :- (The No of vote reassignments protocols, the No of votes assigned to a site changes dynamically).

- Change the set of sites that can form a majority.
- Change the distribution of votes.

**Dynamic Vote Reassignment** :- → Majority based approach.
→ Dynamic vote reassignment.

- No of votes per site changes
- ~~two~~ Kinds -
  1. group Consensus on new assignment
  2. autonomous reassignment, ratified by majority of sites.

**Autonomous Vote Reassignment** :-

When site $i$ wants to increase $V_i[i]$ -
- Send $V_i$ & $N_i$ along with new vote value $x$ to all communicating sites.
- wait for a majority of sites to respond
- If a majority is collected, update $V_i[i]$ to the new value of increment $N_i[i]$.

When site $j$ receives a vote-increasing request from site $i$ with $V_i$, $N_i$, & $x$ -
- $V_j[i] = x$
- $N_j[i] = N_j[i] + 1$

**Vote Decreasing Protocol** :-

When a site wants to decrease $V_i[i]$
- Set $V_i[i]$ to the New value
- increment $N_i[i]$
- Send $V_i$ & $N_i$ to other sites.

When site $j$ receives a vote-Decreasing request from site $i$ with $V_i$ & $N_i$ -
- $V_i[i] = V_i[i]$
- $N_j[i] > N_i[i]$

## Vote Collecting Protocol →

- for each reply $V_j$ & $N_j$ received by site $i$;
  - * $V_i[j] = V_j[j]$
  - * if $V_j[j] > V_i[j]$ or $(V_j[j] < V_i[j]$ & $N_j[j] > N_j[j])$ then
    $V_i[i] = V_j[j]$;
    $N_i[j] = N_j[j]$;
    end if;
- if site $j$ did not respond to site $i$ -
  - * find $K \in G$ & $N_K[j] = \max\{N_p[j] : p \in G\}$;
    $G$ = the set of all sites that replied to $i$, that is find the site that has the latest information on the votes assigned to site $j$.
  - * $V_i[j] = V_K[j]$;
  - * $V_i[j] = V_K[j]$;
  - * $N_i[j] = N_K[j]$.

## Deciding the outcome →

- Let $K$ be the set of all sites, & $G$ be the set of sites that responded to the ballot.
  - The total no of votes in the system is computed as -

$$TOT = \sum_{K \in K} V_i[K]$$

  - The total no of votes received is computed as -

$$RCVD = \sum_{K \in G} V_i[K]$$

- Site $i$ has a majority iff $\boxed{RCVD > TOT/2}$.

UNIT-5

(CS-7) 66 B

5

# Transactions and Concurrency Control

**Q.1. What do you mean by Transaction in context of Distributed Systems ? What are its major properties ?**

**Ans.** In the database systems, multiple users concurrently access the data/records for various operations. This accessing is possible through the database transactions. The concurrency control is needed to maintain the correctness of database during concurrent access of data. In this way, the database system is assumed to be the set of data objects that are shared and accessed by the various users. The data objects include a data field value, a record, a file, etc. In database, every data value belongs to a specific domain set and comprises a data field.

When the user access a database, it means, access to the database value for some operations. The operations and processing on the database are called the Database Actions or Transactions. Various processing required by a user are grouped together in the form of a program and this program works as a unit for program execution with database. This logical unit of the interaction between user and database is called as Transaction.

A transaction may be any read, write or compute operation on any data object of a database. The transaction process does not affect the consistent state of a database. If the database is in consistent state before the transaction, it will remain in the consistent state after the transaction also. It means, the transaction is atomic in nature. Either, the transaction unit executes successfully or it will not execute at all.

The result of a transaction are seen to the user only after the completion of the transaction, *i.e.*, after transaction termination when a transaction is terminated, the user gets the notification of its success or failure through the resulting messages.

**Properties of Transactions :** Transactions have four essential properties, known as ACID properties :

(1) **Atomicity :** Atomicity means, that the transaction completes its processing as an indivisible and atomic unit either successfully or unsuccessfully. After the transaction, the database changes consistency or not changed at all.

(2) **Consistency :** Consistency means the database was in consistent state before transaction and after the termination of transaction, the database will also be in the consistent state.

(3) **Isolation :** Isolation property indicates that every action processed by a transaction is kept isolated until the completion of transaction. During the transaction processing, the processing will be hidden from outside the transaction.

(4) **Durability :** Durability means, any failure made after the commit operation will not affect the database. The commit action will reflect its results to the database after its termination.



**Q.2. Write down the States of Transaction.**

**Ans.** The transaction itself is an independent unit which is processed by the user. But this atomic operation has several different life states during its processing. In a database system, a transaction may be found in several working states as modify, abort, commit, rollback, etc.

When a transaction starts for a data object, it is called in the initiated state. This transaction moves forward in three different directions. The one way leads to the successful termination of the transaction where the commit operation is performed successfully. The second way forces a transaction to terminate. This is called as the Murderous Termination, which occurs on detecting an error during the transaction processing and the transaction is aborted at once. The another way for transaction termination is the Suicidal Termination where the transaction does not update the database but rollback itself and abort the operation.

The following figure illustrates the various transaction states in a system.

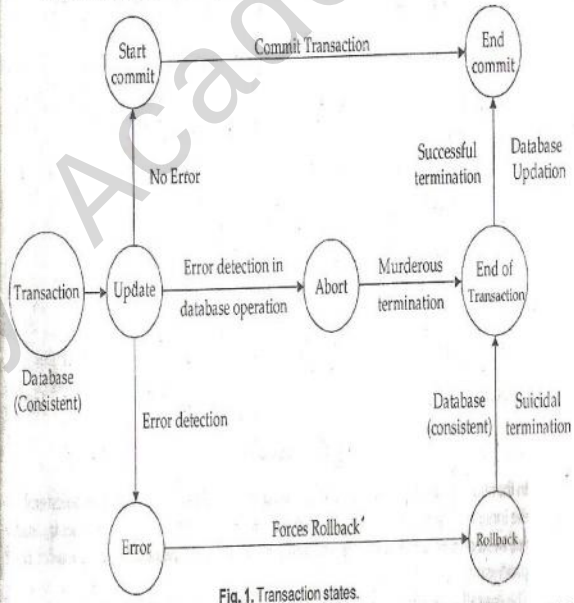



**Fig. 1.** Transaction states.

In the above figure, we can see that a transaction always reaches to its end either successfully or unsuccessfully. The database is assumed in consistent state before transaction state. When a transaction starts, it initiates the update operation, *i.e.*, modify transaction. This state of the database is called as the modify state. Then the transaction is executed till its termination. When a transaction executes successfully, it completes the modify state of itself. The commit transaction is started if no error is detected out of transaction. When the transaction is terminated, it is in the consistent state. The change made by the transaction are then reflected to database. When all the changes, made by any transaction are reflected to the database, then the transaction is said to be in consistent state. The transaction may be successful or unsuccessful. Then transaction either be successful or murderous or suicidal.

**Q.3. What is Nested Transaction ? What are the advantages of nested transactions ?**

**Ans.** In a database system, the transaction is said to be the query to the database management software program. Most often, the transaction is considered as the independent unit which

produces the result of one query at once. But sometimes, the result of a transaction may depend on the result of another transaction. In such case, the result is not independent. The database system uses a transaction to get a result database. Then this result value is used by any other transaction to get any other data value.

In this manner, the transaction (query) is based on other transaction (subquery). The subquery is the part of a transaction and so called as **nested transaction.** These are called nested because, these become the integral part of a transaction process. The following figure illustrates the processing of the nested transaction :

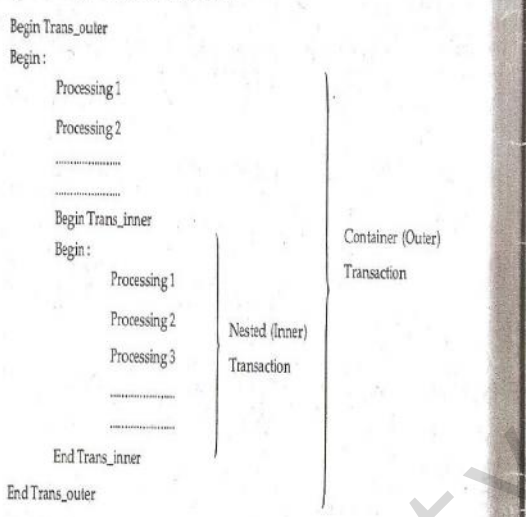



**Fig. 2.** Nested transaction.

In the above figure, two transactions are related with each other. The outer transaction embodies the inner transaction. When the transaction processing starts to execute, the inner transaction *i.e.*, nested transaction executes first and pass its result to the outer transaction for further processing.

The overall transaction result is based on the accumulated result of the two transactions.

**Advantages :** Nested transactions have the following main advantages :

1. Subtransactions at one level may run concurrently with other subtransactions at the same level in the hierarchy. This can allow additional concurrency in a transaction.
2. Subtransactions can commit or abort independently. In comparison with a single transaction, a set of nested sub transactions is potentially more robust.

**Q. 4. What do you understand by concurrency control in distributed transactions ?**

**Ans.** Each server manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions. Therefore, each server is responsible for applying concurrency control to its own objects. The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner.



This implies that if transaction T is before transaction U in their conflicting access to objects at one of the servers then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both T and U.

**Q. 5. What are Locks ? Explain the locking algorithms for Distributed Database.**

**OR**

**What do you mean by two phase locking ? How it is different from strict two phase locking ? Explain.** **[UPTU 2010, Marks 10]**

**Ans.** Every data object has a lock associated with the transaction with database can make the request for that lock as required. The transaction may hold or release the lock on the data object as required by the transaction. The transaction is said to have the locked data object if it holds a lock.

In a transaction, the data object can be locked in two modes as :

(i) **Exclusive Lock :** In case of Exclusive Lock, no other transaction can concurrently lock that data object at all.

(ii) **Shared Lock :** If a data object is locked by a transaction in shared node, the other transactions can also concurrently lock it (but only in shared node).

Thus, we can say that locking the data object means restricting it for other transactions. The locked data object becomes inaccessible to other database transactions. Locking the data objects is helpful for concurrency control.

**Lock Based Locking Algorithms :** The lock based concurrency algorithms are used for concurrency control using lock based techniques. All these algorithms allow the locking of data object for transaction. A transaction is considered as the sequence of *n* actions. The lock based algorithms allow read, write, lock and unlock actions. A transaction can be locked for an action and this lock is removed by an unlock action. For a well-performed transaction, the following points are considered :

- The transaction locks a data object before accessing it.
- The transaction does not lock the data object more than once.
- The transaction unlocks all the locked data objects before the completion of the transaction.

Various locking schemes are proposed for the distributed database. Some of them are :

(i) **Static Locking Scheme :** According to static lock scheme, all the required data objects are locked before the execution of an action. All the data objects are predeclared. All these objects are unlocked by the transaction after the execution of all related actions. This type of locking is very simple, but useful for handling the concurrency in distributed database transaction. The drawback of this scheme is its requirement of previous knowledge about database.

(ii) **Two-phase Locking Scheme :** This locking scheme is also called as '2PL'. This scheme is also called as **Dynamic Locking Scheme,** because the lock is applied, to data object only when the transaction needs the data object. A specific constraint (Rate) is applied to the lock acquisition and lock release. A transaction cannot request a lock on the data object after its unlocking. So, all the required data objects must be locked before unlocking the data objects. The two-phase locking scheme has two phases of its working as :

**Phase 1 : Growing Phase :** — locks are acquired & no locks are released.

(i) Transaction may obtain locks.

(ii) Transaction may not release locks.

**Phase 2 : Shrinking Phase :** — locks are released & no locks are required.

(i) Transaction may release locks.

(ii) Transaction may not obtain locks.

There is one lock point present in between these two phases. At the Lock Point Stage, the transaction holds locks on all the needed data objects. Figure below shows a two phase locking scheme.

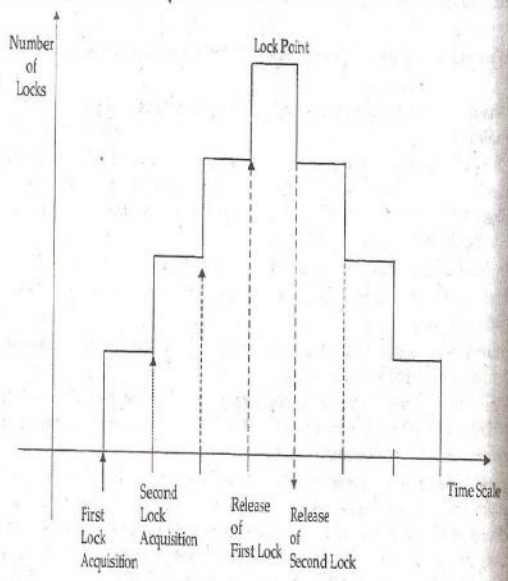



Fig. 3. Two Phase Locking Scheme.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

Two phase locking does not ensure freedom from deadlock. Cascading rollback may occur.

Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase locking protocol. The strict-two-phase locking protocol requires that in addition to locking being two-phase, all exclusive mode locks taken by a transaction must be held until that transaction commits.

(iii) **Timestamp Based Locking Scheme :** A transaction is assigned a unique timestamp when initiation. The timestamp is used to specify the order of the transactions and so helpful in transaction ordering. The another use of the time stamp is to solve the conflict problem between the transactions and so the time stamps are helpful for deadlock prevention.

A conflict is resolved by Wait Action, Restart Action, Die Action, Wound Action.

- The die action and the wound action are associated with the Restart of transaction.
- The die action aboris the requesting transaction and starts it as a fresh transaction.
- The requesting transaction is preceded only when the wounded transaction aborts completes.

(iv) **Non-Two Phase Locking Scheme :** This locking scheme is used for the database in which data objects are organized as an hierarchy, *i.e.*, inverted tree like structure. The Non-Two Phase locking is helpful for deadlock free transaction in hierarchical database. In such scheme



a data object cannot be locked for more than one time. When a transaction needs some data object, it is provided by the system after locking it. Sometimes, a transaction may try to lock such an object which is already locked. In such situation, that transaction will not be able to lock that data object and the blocking of that transaction takes place. But any other transaction seeking for that data object will try to get it and lock the data object for the further processing. Look at the following hierarchical tree of the database system.

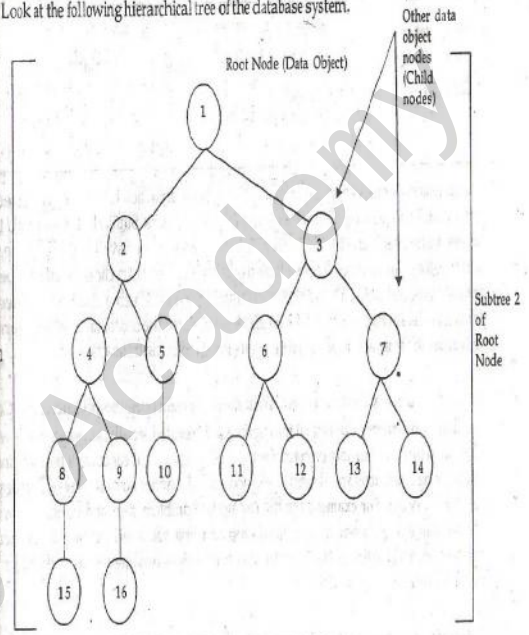



Fig. 4. Hierarchical Tree for Database System.

In such database system, the transaction initiates at a specific data object node and locks that data object. Furthermore, the transaction can only lock the data object in the subtree of that node. It means a data object node is locked first and then locks its child nodes only.

The Non-Two Phase locking scheme is more advantageous that the 2PL scheme, because it provides the freedom from deadlock in the system. One more advantage of this locking scheme is its dynamic nature, *i.e.*, the lock is released after the use of data object by the transaction. By this concept, the data objects used by a transaction will be allowed to be used by other transaction very soon after the completion of first transaction.

**Q. 6. Define Locking in Concurrency Control.**

**Ans.** The oldest and most widely used concurrency control algorithm is locking. In a distributed transaction, the locks on an object are held locally (in the same server). The local lock manager can decide whether to grant a lock or make the requesting transaction wait. However, it cannot release any lock until it knows that the transaction has been committed or aborted at all the servers involved in the transaction. When locking is used for concurrency control, the objects remain locked and are unavailable for other transactions during the atomic commit protocol, although an aborted transaction releases its locks after phase 1 of the protocol.

As lock managers in different servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions. Consider the following interleaving of transactions T and U at servers X and Y:

| T | | | U | | |
|---|---|---|---|---|---|
| *Write (A)* | at X | locks *A* | | | |
| | | | *Write (B)* | at Y | locks B |
| *Read (B)* | at Y | wait for U | | | |
| | | | *Read (A)* | at X | wait for T |

The transaction T locks object A at server X and then transaction U locks object B at server Y. After that, T tries to access B at server Y and waits for U's lock. Similarly, transaction U tries to access A at server X and has to wait for T's lock. Therefore, we have T before U in one server and U before T in the other. These different orderings can lead to cyclic dependencies between transactions and a distributed deadlock situation arises. When a deadlock is detected, a transaction is aborted to resolve the deadlock. In this case, the coordinator will be informed at will abort the transaction at the participants involved in the transaction.

**Q. 7. Why is Concurrency Control needed? Write down the main categories of Concurrency Control.**

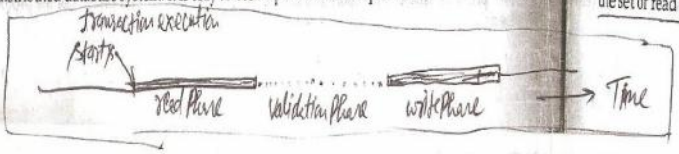
**Ans. Need of Concurrency Control :** If transactions are executed serially, *i.e.*, sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur. **For example :** In the lost update problem, a second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is last to other transactions running concurrently which need, by their precedence, to read the first value.

**Categories of Concurrency Control :** The main categories of concurrency control mechanisms are :

- **Optimistic :** Delay the checking of whether a transaction meets the isolation and other integrity rules (*e.g.*, serializability and recoverability) until its end, without blocking any of its (read, write) operations and then abort a transaction to prevent the violation, if the desired rules are to be violated upon its commit. An aborted transaction is immediately restarted and re-executed, which incurs an obvious overhead. If not too many transactions are aborted, then being optimistic is usually a good strategy.
- **Pessimistic :** Block an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction.
- **Semi-optimistic :** Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking (if needed) to transaction's end, as done with optimistic.

**Q. 8. Explain Optimistic Concurrency Control.**

**Ans.** Optimistic concurrency control states that the conflicts among the transactions are rare in distributed database system. It is only an assumption so called optimistic.






The optimistic concurrency algorithms are also based on this concept. These algorithms are developed by assuming that "the transaction conflicts are rare".

These algorithms apply a checking for the transaction. But this checking is not performed before transaction. When a transaction is executed, it is checked for its conflicts with other concurrent transactions. If any conflict is found, then the transaction will be aborted and will not forward the update operation. If no conflict is found, the transaction process to commit and do the updation in database as a result of transaction. The process of undone the transaction is also known as Rollback.

A transaction after starting leads to either commit or rollback. If the transaction is successfully executed, it leads to the transaction commit and causes the change in data of database as the result of transaction committed. On the other hand, the transaction leads to transaction rollback, if the transaction will not execute successfully due to some conflict with any other transaction.

The most common algorithm used for optimistic concurrency control was proposed by Kung and Robinson. They proposed that the transaction executes first with any synchronization and concurrency check. But it will not update the database finally. The transaction result is written to database only when it is not conflicted, otherwise not.

**For example :** Consider the following interleavings of transactions T and U, which access objects A and B at server X and Y, respectively.

| T | | U | |
|---|---|---|---|
| Read (A) | at X | Read (B) | at Y |
| Write (A) | | Write (B) | |
| Read (B) | at Y | Read (A) | at X |
| Write (B) | | Write (A) | |

The transactions access the objects in the order T before U at server X and in the order U before T at server Y. Now suppose that T and U start validation at about the same time, but server X validates T first and server Y validates U first.

**Q. 9. What do you mean by Timestamp Ordering?**

**Ans.** Each transaction is assigned a unique timestamp value when it starts. The timestamp defines its position in the time sequence of the transaction. Requests from transaction can be totally ordered according to their timestamp. The basic timestamp ordering rule is based on operation conflicts and it is very simple. A transaction's request to write an object is valid only if that object was last read and written by earlier transaction. A transaction's request to read an object is valid only if that objects was last written by an earlier transaction.

The timestamp ordering rule is refined to ensure that each transaction accesses a consistent set of versions of the objects. The write operations may be performed after the close transaction operation has returned. Without making the client wait, but the client must wait when read operations need to wait for earlier transactions to finish. The time stamps are assigned from server's clocks.

The write operations are recorded in tentative versions of objects and are invisible to other transactions until a close transaction request is issued and the transaction is committed. The write timestamp of the committed object is earlier than that of any of its tentative versions and the set of read timestamps can be represented by its maximum members.

Whenever a transaction's write operation on an object is accepted, the server creates a new tentative version of the objects with write timestamp set to the transaction timestamp. Transactions read operation is directed to the version with the maximum write timestamp less than the transaction timestamp. Whenever a transaction's read operation on an object is accepted, the timestamp of the transaction is added to its set of read timestamp.

When a transaction is committed, the value of the tentative versions becomes the value of the objects and the timestamp of the tentative versions becomes the timestamps of the corresponding objects.

**Q.10. Explain the methods for concurrency control in distributed transaction.** [UPTU 2010, Marks 5]

OR

**What are the essential differences in the lock-based protocols and time stamp based protocols ?** [UPTU 2004, 2005; Marks 5]

**Ans.** The three types of concurrency control algorithms are : the locking scheme, timestamp method and the optimistic control. All of these are applied to distributed database system. Their comparision depends on the necessity of data objects, synchronization, data updation and data consistency.

- Timestamp method is similar to two-phase locking in that both use pessimistic approaches in which conflicts between transactions are detected as each object is accessed.
- Timestamp ordering decides serialization order statically when transaction start. On the other hand two-phase locking decides the serialization order dynamically.
- Timestamp ordering is better than strict two phase locking for read-only transactions, while two-phase locking is better when the operations in transactions are predominantly updated.
- Timestamp ordering aborts the transaction immediately, whereas locking makes the transaction wait but with a possible later penalty of aborting to avoid deadlock.
- When optimistic concurrency control is used, all transactions are allowed to proceed, but some are aborted when they attempt to commit or forward validation transaction are aborted earlier. This results in relatively efficient operation when there are few conflicts, but a substantial amount of work may have to be repeated when a transaction is aborted.

**Q.11. Explain Multiversion Timestamp Ordering.**

OR

**What are the advantages and drawbacks of Multiversion Timestamp Ordering in comparison with the Ordering Timestamp Ordering.**

**Ans.** In multiversion timestamp ordering a list of old committed versions as well as tentative versions is kept for each object. This list represents the history of the values of the object. Each version has a read timestamp recording the largest timestamp of any transaction that has read from it in addition to a write timestamp. Whenever a write operation is accepted, it is directed to a tentative version with the write timestamp of the transaction. Whenever a read operation is carried out it is directed to the version with the largest write timestamp less than the transaction timestamp. If the transaction timestamp is larger than the read timestamp of the version being used, the read timestamp of the version is set to the transaction timestamp. When a read arrives late, it can be allowed to read from an old committed version, so there is no need to abort the read operations. In multiversion timestamp ordering, read operations are always permitted although they may have to wait for earlier transaction to complete (either commit or abort), which ensures that executions are recoverable.



**Advantages:** The main advantage of multiversion timestamp ordering is that a read request never fails and is never made to wait.

**Disadvantages:**

(1) The reading of a data item also requires the updating of the R timestamp field resulting in two potential disk accesses, rather than one.

(2) The conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive.

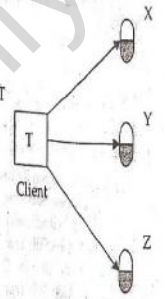
This multiversion timestamp ordering scheme does not ensure recoverability and cascadelessness. It can be extended in the same manner as the basic timestamp ordering scheme, to make it recoverable and cascadeless.

**Q. 12. Discuss Flat and Nested Distributed Transactions.**

**Ans.** A client transaction becomes distributed if it invokes operations in several different servers. There are two different ways that distributed transactions can be structured : as flat transactions and as nested transactions.

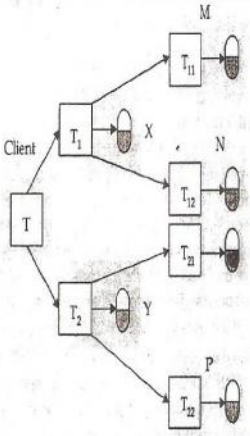
In a **flat transaction**, a client makes requests to more than one server. For example,

In the Fig. 5(a), transaction $T$ is a flat transaction that invokes operations on objects in servers X, Y and Z. A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction can only be waiting for one object at a time.




(a) Flat transactions

In a **nested transaction**, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting.




(b) Nested transactions

Fig. 5.

The Fig. 5(b), shows a client's transaction $T$ that opens two subtransactions $T_1$ and $T_2$, which access objects at servers X and Y. The subtransactions $T_1$ and $T_2$ open further subtransactions

$T_{11}$, $T_{12}$, $T_{21}$ and $T_{22}$ which access objects at servers $M$, $N$ and $P$. In the nested case, subtransactions at the same level can run concurrently, so $T_1$ and $T_2$ are concurrent, and as they invoke objects in different servers, they can run in parallel. The four subtransactions $T_{11}$, $T_{12}$, $T_{21}$ and $T_{22}$ also run concurrently.

Consider a distributed transaction in which a client transfers \$10 from account $A$ to $C$ and then transfers \$20 from $B$ to $D$. Accounts $A$ and $B$ are at separate servers $X$ and $Y$ and accounts $C$ and $D$ are at server $Z$. If this transaction is structured as a set of four nested transaction, as shown in figure below, the four requests (two deposit and two withdraw) can run in parallel and the overall effect can be achieved with better performance than a simple transaction in which the four operations are invoked sequentially.

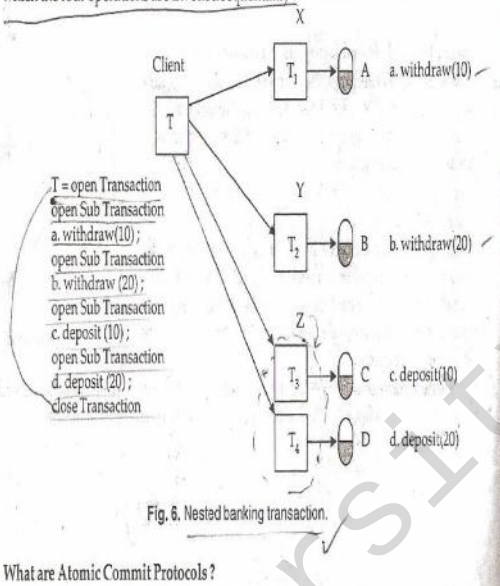



Fig. 6. Nested banking transaction.

**Q. 13. What are Atomic Commit Protocols?**

**Ans.** Every transaction is atomic in nature. In the distributed system, the sequence of such atomic operations execute to perform a specific work. A transaction is said to be atomic, if it is indivisible, instantaneous and cannot be interrupted. Generally, the machine level instructions are assumed atomic because of having all such characteristics and continue process unless the system fails. Such instructions (transactions) are grouped together to perform a task. Atomic transactions are the building blocks for constructing the fault tolerant system. An atomic transaction must possess the following characteristics :

(i) A transaction is atomic if it does not affected by an existing process.

(ii) A transaction is atomic if it does not communicate with other process.

(iii) A transaction is atomic if its process can detect no state change.

(iv) A transaction is atomic if it is considered as indivisible and instantaneous.

In the distributed system, a distributed transaction is a group of several transactions and these are processed in such a manner so that the database consistency is maintained. When a transaction starts, its execution leads to transaction commit or abort. There are various transaction states such as Abort, Commit, Rollback. The transaction may abort due to deadlock, wrong input, data inconsistency, etc.



The distributed system have multiple sites together on which many processes execute to perform a task. The distributed system states that a transaction must be processed at every site or at no site to maintain the database integrity. In the same manner, all sites of the distributed system are responsible for the atomicity of the transaction. This type of transaction state is called as the **Global Atomicity** for that transaction.

**Q. 14. What do you mean by two-phase commit protocol?** [UPTU 2005, Marks 5]

OR

**What is two-phase commit protocol? Explain its operation.**

**Ans.** Two-phase commit protocol is designed to allow any participant to abort its part of transaction. Due to the requirement for atomicity, if one part of a transaction aborted then the whole transaction must also be aborted.

**Phase 1 (voting phase):**

1. The coordinator sends a canCommit? request to each of the participants in the transaction.
2. When a participant receives canCommit? request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

**Phase 2 (completion according to outcome of vote):**

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are Yes the coordinator decides to commit the transaction and sends a doCommit request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends doAbort requests to all participants that voted Yes.
4. Participants that voted Yes are waiting for a doCommit or doAbort request from the coordinator. When a participant receives one of the these messages it acts accordingly and in the case of commit, makes a haveCommitted calls as confirmation to the coordinator.

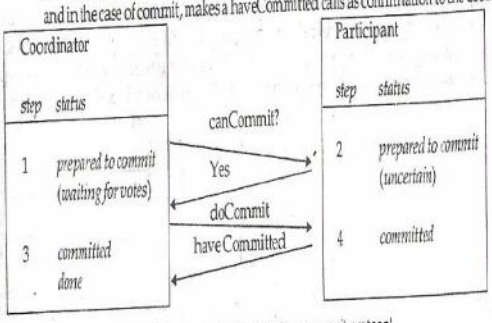



Fig. 7. Communication in two-phase commit protocol.

**Operations for two-phase commit protocol:**

canCommit? (trans) → Yes/No

Call from coordinator to participant to ask whether it can commit a transaction.

Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) → Yes/No

Call from participant to coordinator to ask for the decision on a transaction after it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages.

**Q. 15. In the two-phase commit protocol, why can blocking be completely eliminated, even when the participants elect a new coordinator? [UPTU 2005, Marks 5]**

**Ans.** The two-phase commit protocol is designed to allow any participant to abort its part of a transaction. Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must also be aborted. In the first phase of the protocol, each participant votes for the transaction to be committed or aborted. Once a participant has voted to commit a transaction, it is not allowed to abort it. Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim. A participant in a transaction is said to be in a **prepared** state for a transaction if it will eventually be able to commit it. To make sure of this, each participant saves in permanent storage of all the objects that it has altered in the transaction, together with its status-prepared.

In the second phase of the protocol, every participant in the transaction carries out the joint decision. If any one participant votes to abort, then the decision must be to abort the transaction. If all the participants vote to commit, then the decision is to commit the transaction.

**Q. 16. Explain the Two-Phase Commit Protocol for Nested Transaction. Explain its operations.**

**Ans.** The outermost transaction in a set of nested transactions is called the **top-level transaction**. Transactions other than the top-level transaction are called **subtransactions**. In Fig. 8, $T$ is the top-level transaction, $T_1$, $T_2$, $T_{11}$, $T_{12}$, $T_{21}$ and $T_{22}$ are subtransactions. $T_1$ and $T_2$ are child transactions of $T$, which is referred to as their parent. Similarly, $T_{11}$ and $T_{12}$ are child transactions of $T_1$ and $T_{21}$ and $T_{22}$ are child transactions of $T_2$. Each subtransaction starts after its parent and finishes before it. Thus, for example, $T_{11}$ and $T_{12}$ start after $T_1$ and finish before it.

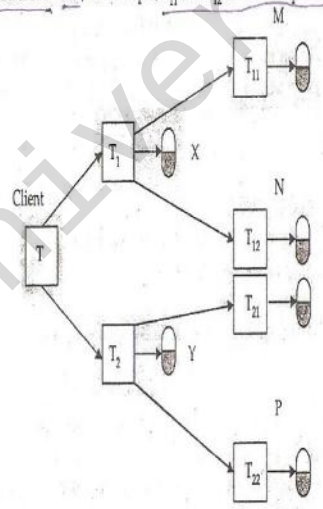



Fig. 8.



When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort. A provisional commit is different from being prepared to commit: it is not backed upon permanent storage. If the server crashes subsequently, its replacement will not be able to commit. After all subtransactions have completed, the provisionally committed ones participate in a two-phase commit protocol, in which servers of provisionally committed subtransactions express their intention to commit and those with an aborted ancestor will abort. A prepared commit guarantees a subtransaction will be able to commit, whereas a provisional commit only means that it has finished correctly – and will probably agree to commit when it is subsequently asked to.

A coordinator for a subtransaction will provide an operation to open a subtransaction, together with an operation enabling the coordinator of a subtransaction to enquire whether its parent has yet committed or aborted.

A client starts a set of nested transactions by opening a top-level transaction with an **openTransaction** operation, which returns a transaction identifier for the top-level transaction. The client starts a subtransaction by invoking the **openSubTransaction** operation, whose argument specifies its parent transaction. The new subtransaction automatically joins the parent transaction, and a transaction identifier for a subtransaction is returned.

**Operations in coordinator for nested transactions:**

*openSubTransaction(trans)* → *subTrans*

Open a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

*getStatus(trans)* → *committed, aborted, provisional*

Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed, aborted, provisional.*

**Q. 17. Explain how the two-phase commit protocol for nested transaction ensures that if the top-level transaction commits all the right descendents are committed or aborted. [UPTU 2008, Marks 10]**

**Ans.** Consider the top-level transaction $T$ and its subtransactions shown in Fig. 9. Each subtransaction has either provisionally committed or aborted. For example, $T_{12}$ has provisionally committed and $T_{11}$ has aborted, but the fate of $T_{12}$ depends on its parent $T_1$ and eventually on the top-level transaction, $T$. Although $T_{21}$ and $T_{22}$ have both provisionally committed, $T_2$ has aborted and this means that $T_{21}$ and $T_{22}$ must also abort. Suppose that $T$ decides to commit in spite of the fact that $T_2$ has aborted, also that $T_1$ decides to commit in spite of the fact that $T_{11}$ has aborted.

When a top-level transaction completes, its coordinator carries out a two-phase commit protocol. The only reason for a participant subtransaction being unable to complete is if it has crashed since it completed its provisional commit. When each subtransaction was created, it joined its parent transaction. Therefore, the coordinator of each parent transaction has a list of its child subtransactions. When a nested transaction provisionally commits, it reports its status and the status of its descendants to its parent. When a nested transaction aborts, it just reports abort to its parent without giving any information about its descendants. Eventually, the top-level transaction receives a list of all the subtransactions in the tree, together with the status of each. Descendants of aborted subtransactions are actually omitted from this list.

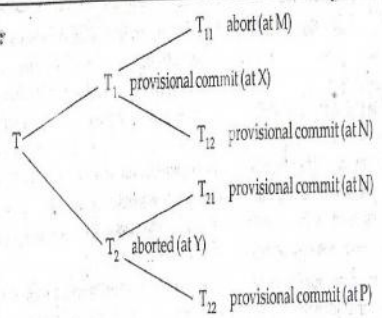



Fig. 9. Transaction *T* decides whether to commit.

**Q. 18. Discuss Concurrency Control in Distributed Transactions.**

**Ans.** The database system contains the three modules for concurrency control. These modules are the Transaction Manager, the Data Manager and the Scheduler. Any transaction from the user to database goes first of all to the transaction manager. Then the transaction manager interact with data manager for further execution. The transaction manager assigns a timestamp to a transaction. It also assigns the request for lock and unlock of data objects as per the user requirement.

The scheduler works between the transaction manager and data manager. It is used for concurrency control. The scheduler unlocks the data objects or release locks on to them as used by the transaction. The data manager manages the database and read from it or write into it according to the user's requirement. The data manager is also responsible for failure recovery.

The transaction manager executes the transaction sequentially. It takes one request and sends it to data manager through scheduler and then data manager gives the appropriate result. After that the data manager executes the transaction actions in the form of execution.

For concurrency control the scheduler modifies the stream of actions towards the data manager.

**Q. 19. What are the problems with Concurrency Control?**

**Ans.** In the database system the probability of the consistency in database can be increased by executing transaction in the sequential manner. However, the serial execution of transaction also gives the poor response to the user requirements and also there is poor utilization of system resources. By executing transactions concurrently the efficiency can be increased. But in concurrently execution, several accesses will be made to some data object and so any critical position can be made in the concurrent execution of transaction.

(i) **Inconsistent Retrieval of Data :** When many transaction read the data objects from the database through transaction manager, scheduler and data manager ; but before any another transaction has completed, its modification to these data objects, then the former transaction undergoes in a problem of retrieving incorrect values of data objects. This is known as Inconsistent Retrieval of Data.

(ii) **Inconsistent Update of Data :** When many transaction work on the command set of data objects of any database, although leaving it in inconsistent state, the inconsistent update occurs. Since if the concurrency of the transactions is not controlled then because of many inconsistent states the database may be in inconsistent situation. To overcome this problem the concurrency control method is used. These mechanisms control the relative orders of the conflicting works, so that every transaction has consistent state and on the completion of all



transactions the database is in consistent state. So the consistent state of the database system can be controlled by controlling the concurrency.

**Q. 20. Explain Distributed Transaction.**

OR

**How does the Distributed Deadlock affect the transactions?**

**Ans.** Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks. With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock.

Most deadlock detection schemes operate by finding cycles in the transaction wait-for-graph. In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for-graph can be constructed from the local ones. There can be a cycle in the global wait-for-graph that is not in any single local one and endash ; that is, there can be a distributed deadlock.

**For example :**

| *U* | | *V* | | *W* | |
|---|---|---|---|---|---|
| *d.deposit*(10) | lock D | | | | |
| | | *b.deposit*(10) | lock *B* at *Y* | | |
| *a.deposit*(20) | lock *A* at *X* | | | | |
| | | | | *c.deposit*(30) | lock *C* at *Z* |
| *b.withdraw*(30) | wait at *Y* | | | | |
| | | *c.withdraw*(20) | wait at *Z* | | |
| | | | | *a.withdraw*(20) | wait at *X* |

Fig. 10. Interleavings of transactions *U*, *V* and *W*.

The above figure shows the interleavings of the transactions *U*, *V* and *W* involving the objects *A* and *B* managed by servers *X* and *Y* and objects *C* and *D* managed by server *Z*.

The complete wait-for-graph, in Fig. 11(*a*), shows that a deadlock cycle consists of alternate edges, which represent a transaction waiting for an object and an object held by a transaction. As any transaction can only be waiting for one object at a time, objects can be left out of wait-for-graphs as shown in Fig. 11(*b*).

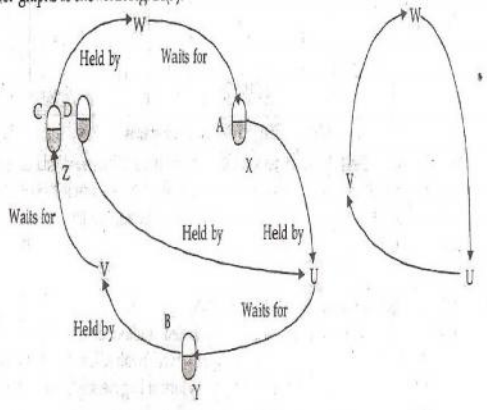



Fig. 11. Distributed deadlock

Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for-graph that is distributed among the servers that were involved in the transactions. Local wait-for-graphs can be built by the lock manager at each server. In this example, the local wait-for-graphs of the servers are :

Server Y : U → V (added when U requests b.withdraw(30))

Server Z : V → W (added when V requests c.withdraw(20))

Server X : W → U (added when W requests a.withdraw(20))

As the global wait-for-graph is held in part by each of the several servers involved, communication between these servers is required to find cycles in the graph.

A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for-graph to the global deadlock detector, which amalgamates the information in the local graphs in order to construct a global wait-for-graph. The global deadlock detector checks for cycles in the global wait-for-graph. When it finds a cycle, it makes a decision on how to resolve the deadlock and informs the servers as to the transaction to be aborted to resolve the deadlock.

**Q. 21. What do you mean by Phantom Deadlock and how it occurs in Distributed System ?**

**Ans.** A deadlock that is 'detected' but is not really a deadlock is called a phantom deadlock. In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another. If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected. As this procedure will take some time, there is a chance that one of the transactions that holds a lock will mean while have released it, in which case the deadlock will no longer exist.

**For example :** Consider the case of a global deadlock detector that receives local wait-for graphs from servers X and Y, as shown in Fig. 12. Suppose that transaction U then releases an object at server X and requests the one held by V at server Y. Suppose also that the global detector receives server Y's local graph before server X's. In this case, it would detect a cycle $T \rightarrow U \rightarrow V \rightarrow T$, although the edge $T \rightarrow U$ no longer exists.

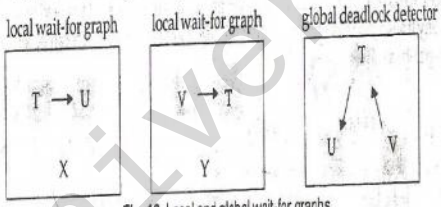



**Fig. 12.** Local and global wait-for graphs.

A phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure. For example, if there is a cycle $T \rightarrow U \rightarrow V \rightarrow T$ and U aborts after the information concerning U has been collected, then the cycle has been broken already and there is no deadlock.

**Q. 22. Define Edge Chasing in Distributed Deadlock.**

**Ans.** Edge chasing or path pushing is used in distributed deadlock detection. In this, the global wait for graph is not constructed, but each of the servers involved has knowledge about some of its edges. The servers attempt to find cycles by forwarding messages called probes, which follows the edges of the graph throughout the distributed system. A probe message consisting of transactions wait-for-relationship representing a path in the global wait-for-graph.



Edge chasing has three steps :

- **Initiation :** The server initiates to detect deadlock.
- **Detection :** Detection consists of receiving probes and deciding whether deadlock has occurred and forward probes.
- **Resolution :** When a cycle detected. A transaction in the cycle is aborted to break deadlock

**Q. 23. Discuss Transaction Recovery and different methods of Transaction Recovery.**

**Ans.** The atomic property of transactions requires that the effects of all committed transactions and none of the effects of incomplete or aborted transactions are reflected in the objects they accessed. This property can be described in terms of two aspects : durability and failure atomicity. Durability requires that objects are saved in permanent storage and will be available indefinitely thereafter. Therefore, an acknowledgement of client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's (volatile) objects. Failure atomicity requires that effects of transactions are atomic even when the server crashes. Recovery is concurrent with ensuring that a server's objects are durable and that the service provides failure atomicity.

Although file servers and database servers maintain data in permanent storage, other kinds of servers of recoverable objects need not do so except for recovery purposes. When a server is running it keeps all of its volatile memory and records its commited objects in a recovery file or files. Therefore, recovery consists of restoring the server with the latest committed versions of its objects from permanent storage. Databases need to deal with large volumes of data. They generally hold the objects in stable storage on disk with a cache in volatile memory.

The two requirements for durability and for failure atomicity are not really independent of one another and can be dealt with by a single mechanism—the recovery manager. The task of a recovery manager is :

- To save objects in permanent storage (in a recovery file) for committed transactions ;
- To restore the server's objects after a crash ;
- To reorganize the recovery file to improve the performance of recovery ;
- To reclaim storage space (in the recovery file).

**Q. 24. Write short note on :**

**(i) Logging.**

**(ii) Shadow versions.**

**Ans. (i) Logging :** In the logging technique, the recovery file represents a log containing the history of all the transactions performed by a server. The history consists of values of objects, transaction status entries and intentions lists of transactions. The order of the entries in the log reflects the order in which transactions have prepared, committed and aborted at that server.

During the normal operation of a server, its recovery manager is called whenever a transaction prepares to commit, commits or aborts a transaction. When the server is prepared to commit a transaction, the recovery manager appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction (prepared) together with its intentions list. When a transaction is eventually committed or aborted, the recovery manager appends the corresponding status of the transaction to its recovery file.

**(ii) Shadow Versions :** The shadow versions technique is an alternative way to organize a recovery file. It uses a map to locate versions of the server's object in a file called a **version store.** The map associates the identifiers of the server's versions in the version store. The versions written by each transaction are shadows of the previous committed versions.

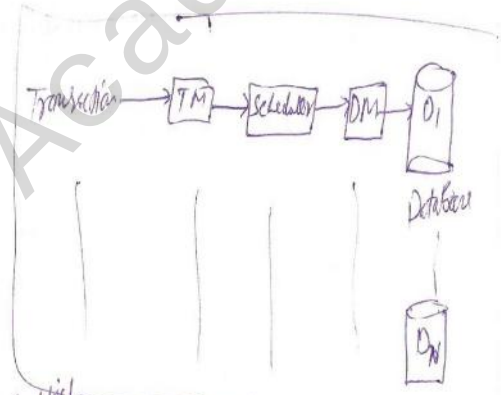
## DATABASE SYSTEM :→

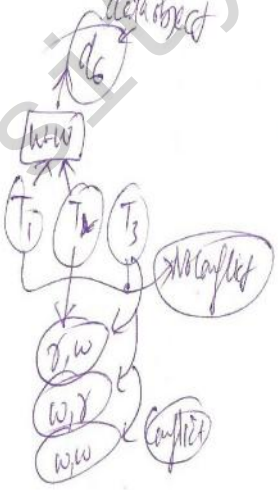
- In database System, users concurrently access data objects by executing transactions. Concurrency control is the process of controlling concurrent access to a database to ensure that the correctness of the database is maintained.
- A Database system consists of a set of shared data objects that can be accessed by users. A data object can be a page, a file, a segment or a record.
- For the purpose of concurrency control, we will view a database as a collection of data objects $(d_1, d_2, d_3 \ldots d_n)$.
- each data object takes values from a specified domain. the state of database is given by the values of its data objects.
- In database contain semantic relationship, called consistency assertions or integrity constraints must hold among its data objects.

$T_1 \rightarrow RS(T_1) = \{d_1, d_2, d_3\}$ $WS(T_1) = \{d_3, d_6\}$

$T_2 \rightarrow RS(T_2) = \{d_2, d_4, d_5\}$ $WS(T_2) = \{d_2, d_4, d_6\}$

$T_3 \rightarrow RS(T_3) = \{d_1, d_2, d_4\}$ $WS(T_3) = \{d_2, d_4\}$




Conflict $T_1 \rightarrow T_2$ $T_1, T_2$

RS(T) Readset.
WS(T) Writeset.

If Conditions

$RS(T_1) \cap WS(T_2) \neq \phi$

$WS(T_1) \cap RS(T_2) \neq \phi$

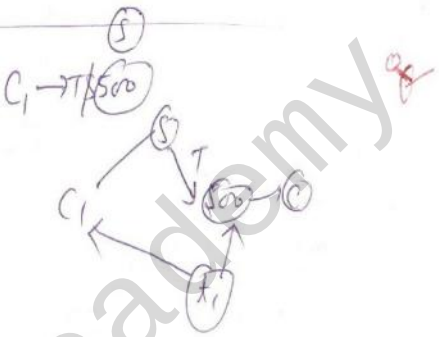
or

$WS(T_1) \cap WS(T_2) \neq \phi$

Slow Model - for the Response of Concurrency Control -

- Transaction Manager (TM)
- Data Manager (DM)
- Scheduler

Transaction — TM — Scheduler — DM — Database



Data objects

A → 10

B → 20

$T_1$ = if A = 0 then B = B + 1

$T_2$ = if B = 0 then A = A + 1

$T_1$ : lock A

$T_2$ : lock B

A held → A

$S = \bigcup_{i=0}^{n} S_i$ ②

$D_i = \{d_i / i \in [1 \cdots M]\}$

$D_i \cap D_j \Theta = \phi$ $(i \neq j)$

$D_1 \cup D_2 \cup D_3 \cdots \cup D_N = \{d_1, d_2 \cdots d_M\}$

$C_1 \rightarrow T$ 500 ①

S = 1000

C = 500

$C_1 \rightarrow$ 500 (1000 − 500) at ⑤

$T_1 \rightarrow$ S (= 500) & C (= 500) at ⑤

1000 → $T_1$

When a transaction is prepared to commit, any of the objects changed by the transaction are appended to the version store, leaving the corresponding committed versions unchanged. These new as yet tentative versions are called **shadow versions.** When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions. To complete the commit process, the new map replaces the old map.

To restore the objects when a server is replaced after a crash, its recovery manager reads the map and uses the information in the map to locate the objects in the version store.

The shadow version method provides faster recovery than logging because the positions of the current committed objects are recorded in the map, whereas recovery from a log requires searching throughout the log for objects.

Shadow versions on their own are not sufficient for a server that handles distributed transactions. Transaction status entries and intentions lists are saved in a file called the transaction status file. Each intentions list represents the part of the map that will be altered by a transaction when it commits.

**Q. 25. Explain Recovery of the Two-Phase Comnit Protocol.**

**Ans.** In phase 1, when coordinator is prepared to commit, its recovery manager adds a coordinator entry to its recovery file. Before participants votes yes, it must have already prepared to commit. When it votes Yes, its recovery manager records a participant entry and add uncertain transaction status to its recovery file as a forced write. When votes No, it adds abort transaction status to its recovery file.

In phase 2, of the protocol, the recovery manager of the coordinator adds either a committed or aborted transaction status to its recovery file, according to the decision. This must be a forced write. Recovery manager of the participants add a commit or abort transaction status to their recovery file according to message received from coordinator. When a coordinator has received a confirmation from all of its participants, its recovery manager adds a done transaction status to its recovery file—this need not to be forced.

The done status entry is not part of the protocol but is used when the recovery file is recognized.

**Q. 26. What do you understand by Replication?**

**Ans.** Replication is a key to providing high availability and fault tolerance in distributed systems. High availability is of increasing interest with the tendency towards mobile computing and consequently disconnected operation. Fault tolerance is an abiding concern for services provided in safety-critical and other important systems.

The replication is "the maintenance of copies of data at multiple computers. Replication is a key to the effectiveness of distributed systems in that it provides enhanced performance, high availability and high fault tolerance. For example, the caching of resources from web server, from web servers in browser and web proxy servers is a form of replication, since the data held in cache and at servers are replicas of one another.

Replication is a technique for enhancing a service. The motivations for replication are to improve a service's performance, to increase its availability, or to make it fault tolerant.

When data are replicated the replication transparency is required, *i.e.*, clients should not normally have to aware that multiple copies of data exist. The consistency of data is another requirement.

**Q. 27. Explain the concept of System Models and Group Communication.**

**Ans.** **System model :** The model involves replicas held by distinct **replica managers,** which are components that contain the replicas on a given computer and perform operations upon them



directly. This general model may be applied in a client-server environment, in which case a replica manager is a server. We shall sometimes simply call them servers instead. Equally, it may be applied to an application and application processes can in that case act as both clients and replica managers. For example, the user's laptop on a train may contain an application that acts as a replica manager for their diary.

We shall always require that a replica manager applies operations to its replicas recoverably. This allows us to assume that an operation at a replica manager does not leave inconsistent results if it fails part-way through. The set of replica managers may be static or dynamic.

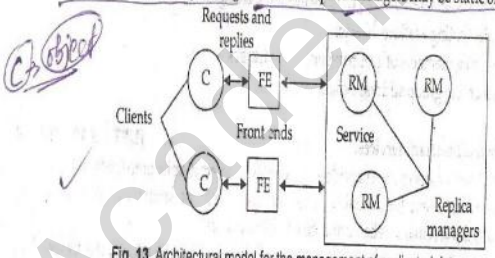



**Fig. 13.** Architectural model for the management of replicated data.

The general model of replica management is shown in above figure. A collection of replica managers provides a service to clients. The client see a service that gives them access to objects (for example, diaries or bank accounts), which infact are replicated at the managers. Clients each request a series of operations—invocations upon one or more of the objects. An operation involves no updates are called read-only requests; requested operations that update an object are called update requests (these may also involve reads).

**Group Communication :** Multicast communication is also known as group communication because process groups are the destinations of multicast messages. Groups are useful for managing replicated data and in other systems where processes cooperate towards a common goal by receiving and processing the same set of multicast messages. They are also useful where the group members independently consume one or more common streams of messages, such as messages carrying events to which the processes react independently.

In a service that manages replicated data, for example, users may add or withdraw a replica manager, or a replica manager may crash and thus need to be withdrawn from the system's operation. A full implementation of group communication incorporates a group membership service to manage the dynamic membership of groups, in addition to multicast communication.

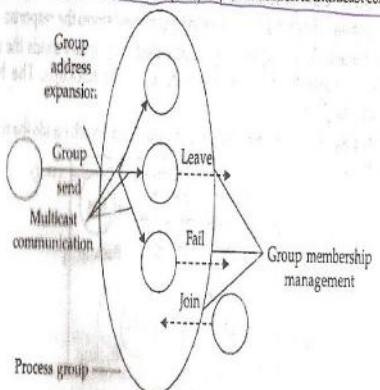



**Fig. 14.** Services provided for process groups.

Multicast and group membership management are strongly interrelated. Fig. 14. shows an open group, in which a process outside the group sends to the group without knowing the group's membership. The group communication services has to manage changes in the group's membership while multicasts take place concurrently.

**Q. 28. What are Roles of the Group Membership Service ?**

**Ans.** A group membership service has four main tasks which are as follows :

- Providing an interface for group membership changes.
- Implementing a failure detector.
- Notifying members of group membership changes.
- Performing group address expansion.

**Q. 29. Explain fault-tolerant services.** **[UPTU 2010, Marks 5]**

**Ans.** For fault tolerant services we provide a service that is correct despite upto $f$ process failures, by replicating data and functionality at replica managers. For simplicity, we assume that communication remains reliable and no partitions occurs.

Each replica manager is assumed to behave according to a specification of the semantics of the objects it manages, when they have not crashed. For example, a specification of bank accounts would include an assurance that funds transferred between bank accounts can never disappear, and that only deposits and withdrawals affect the balance on any particular account.

(1) **Passive (primary-backup) Replication :** In a passive model of replication for fault tolerance, there is at any time a single primary replica manager and one or more secondary replica managers—'backups' or 'slaves'. In the pure form of the model, front ends communicate only with the primary replica manager to obtain the service. The primary replica manager executes the operations and sends copies of the updated data to the backups. If the primary fails, one of the backups is promoted to act as the primary.

The sequence of events when a client requests an operation to be performed is as follows :

a. **Request :** The front end issues the request, containing a unique identifier, to the primary replica manager.

b. **Coordination :** The primary takes each request atomically, in the order in which it receives it. It checks the unique identifier, in case it has already executed the request and if so it simply re-sends the response.

c. **Execution :** The primary executes the request and stores the response.

d. **Agreement :** If the request is an update then the primary sends the updated state, the response and the unique identifier to all the backups. The backups send an acknowldgement.

e. **Response :** The primary responds to the front end, which hands the response back to the client.

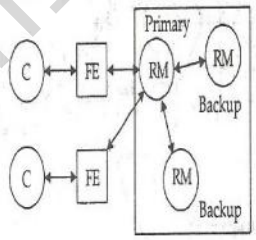



Fig. 15. The passive (primary-backup) model for fault tolerance.



(2) **Active Replication :** In the active model of replication for fault tolerance, the replica managers are state machines that play equivalent roles and are organized as a group. Front ends multicast their requests to the group of replica managers and all the replica managers process the request independently but identically and reply. If any replica manager crashes, then this need have no impact upon the performance of the service, since the remaining replica managers continue to respond in the normal way.

Under active replication, the sequence of events when a client requests and operation to be performed is as follows :

a. **Request :** The front end attaches a unique indentifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive.

b. **Coordination :** The group communication system delivers the request to every correct replica manager in the same (total) order.

c. **Execution :** Every replica manager executes the request.

d. **Agreement :** No agreement phase is needed, because of the multicast delivery semantics.

e. **Response :** Each replica manager sends its response to the front end.

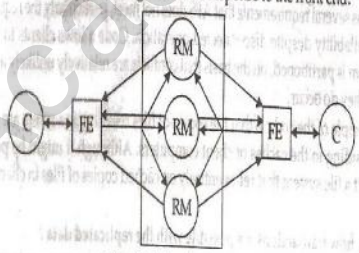



Fig. 16. Active replication.

**Q. 30. Discuss Highly Available Services.** **[UPTU 2010, Marks 5]**

OR

**Write a short note on :**

**(i) Gossip architecture.**

**(ii) Bayou and operational transformation approach.**

**(iii) Coda file system.**

**Ans.** The systems that provide highly available services are :

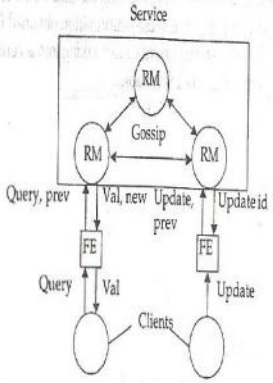



Fig. 17. Query and update operations in a gossip service.

(i) **Gossip architecture :** Gossip architecture as a framework for implementing highly available services by replicating data close to the points where groups of clients need it. The name reflects the fact that the replica managers exchange 'gossip' messages periodically in order to convey the updates they have each received from clients.

A gossip service provides two basic types of operation : queries are read-only operations and updates modify but do not read the state. A key feature is that front ends send queries and updates to any replica manager they choose—any that is available and can provide reasonable response times.

(ii) **Bayou and the operational transformation approach :** The Bayou system provides data replication for high availability with weaker guarantees than sequential consistency. Bayou replica managers cope with variable connectivity by exchanging updates in pairs, in what the designers also call an anti-entropy protocol. But Bayou adopts a markedly different approach in that it enables domain-specific conflict detection and conflict resolution to take place.

(iii) **Coda file system :** The coda file system is a descendent of Andrew File System (AFS) that aims to address several requirements that AFS does not meet, particularly the requirement to provide high availability despite disconnected operation. Code allows clients to update data while the system is partitioned, on the basis that conflicts are relatively unlikely and that they can be fixed if they do occur.

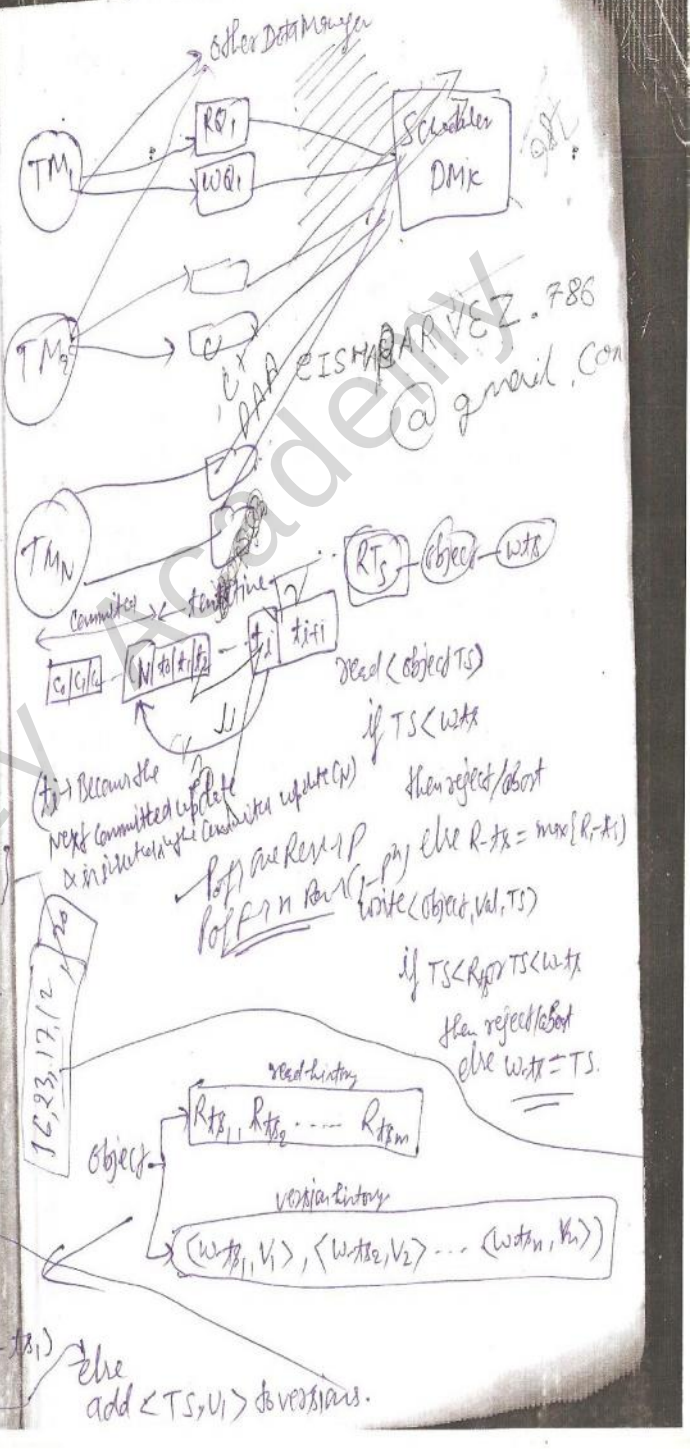
The principle of the coda is that the copies of files residing on servers are more reliable than those residing in the caches of client computers. Although it might be possible logically to construct a file system that relies entirely on cached copies of files in client computers.

**Q. 31. Explain, how transactions are possible with the replicated data ?**

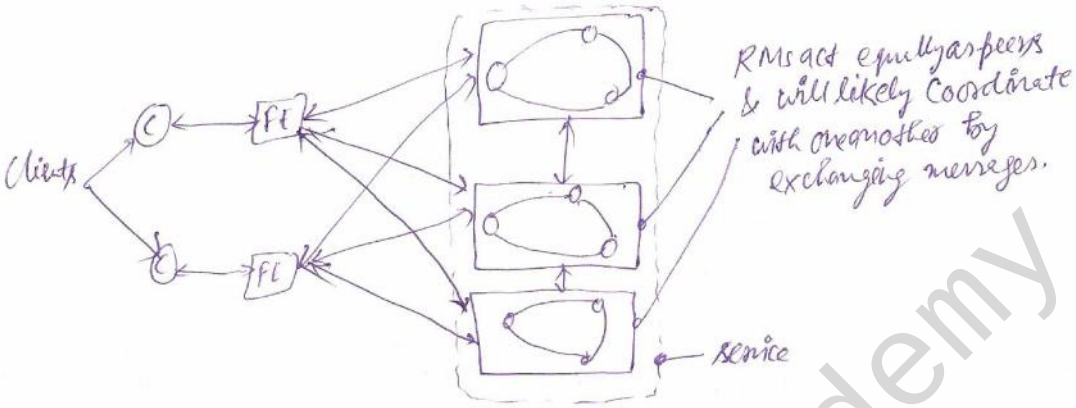
**Ans.** Objects in transactional systems may be replicated to increase both availability and performance. From a client's view point, a transaction on replicated objects should appear the same as one with non-replicated objects. In a non-replicated system, transactions appear to be performed one at a time in some order. This is achieved by ensuring a serially equivalent interleaving of client's transactions. The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set or objects. This property is called **one-copy serializability.**

Each replica manager provides concurrency control and recovery of its own objects. Recovery is complicated by the fact that a failed replica manager is a member of a collection and that the other members continue to provide a service during the time that it is unavailable. When a replica manager recovers from a failure, it uses information obtained from the other replica managers to restore its objects to their current values, taking into account all the changes that have occurred during the time it was unavailable.

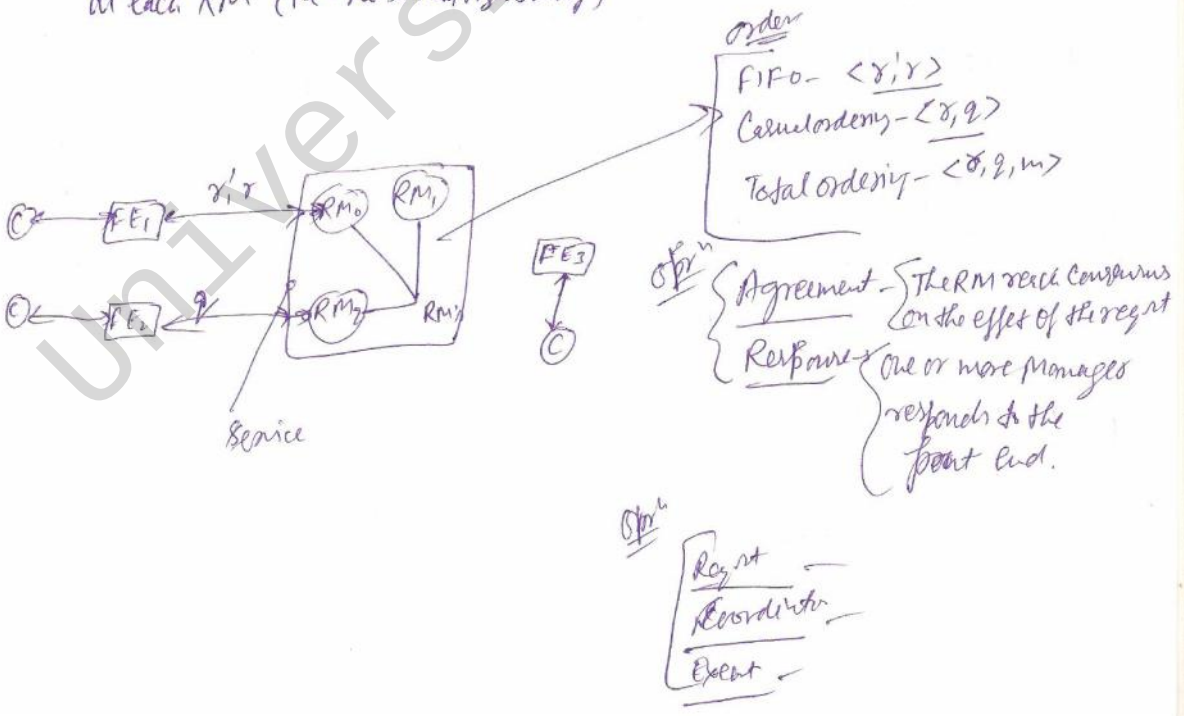
□□□



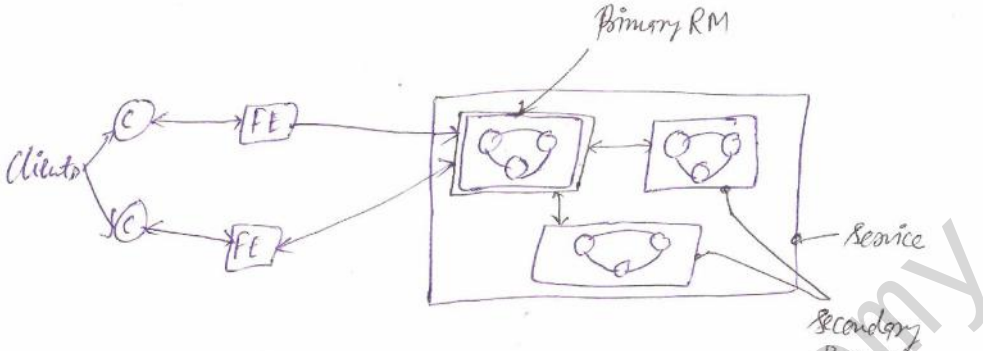
Active Replication →




- Client Requests are Multicast (usually totally ordered Multicast) by the FE to All RMs in the Service Group.
- FEs Collect responses from all non-failed RMs & pass a single response to the client.
- Although client requests are handled in order (i.e with Sequencial Consistency), due to arbitrary delays, they may interleave with other Clients' requests differently at each RM (i.e. no linearizability)



## Passive (Primary backup) Replication:-




- Clients communicate, via their FE, with the Primary RM (e.g reads or updates to objects)
- Primary sends updates to the secondary RMs so that they can replace the primary if it fails, perhaps through some pre-arrangement or an election.
- To maintain consistency during primary replacement, RMs can implement view synchronization, so they know which options have been processed up until the failure.
- Since all of the client operations are fielded by the primary RM, it can ensure that exactly the same sequencing is executed by its secondary RMs, so this technique can achieve linearizability.

op^n
→ Reqst
→ Coordn
→ Execution

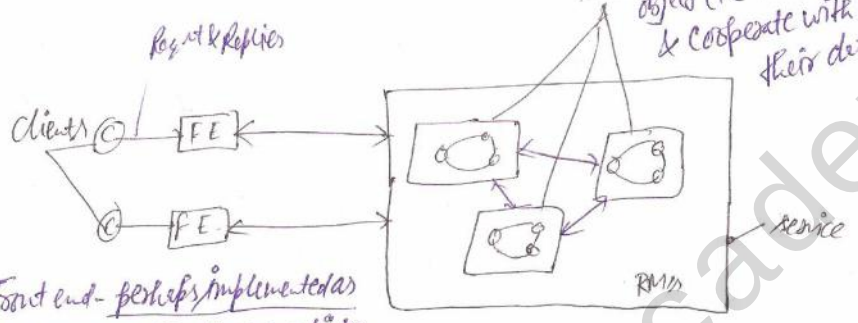
✓ Replica Mangers (RM)
- one replica manager per replica
- Receive FE's reqst apply op^n to its replica atomically.

✓ Frontend (FE)
- one front end per client
- Receive client's reqst Communicate with RMs by message passing.

# Replication

Replicated service –

- Model of a Replicated service.
- View Synchronization Communication.
- Linearizability & Sequential Consistency.




Replica Managers — Maintains Recoverable object (i.e that support tentative oprns) & cooperate with one another to keep their data consistent.

Front end - perhaps implemented as Middleware at the client, mediates betⁿ the client's oprⁿ & the replicated services to provide transparency.

In general, there are 5 Phase involves in Handling the client Request –

① Request - the FE communicates the req^st to one or more of the RM's (depending on the particular replication system) on behalf of the client.

② Coordination - the RM's coordinates to handle the request consistently (e.g agreement on the ordering of reqsts, whether or not the request is valid.)

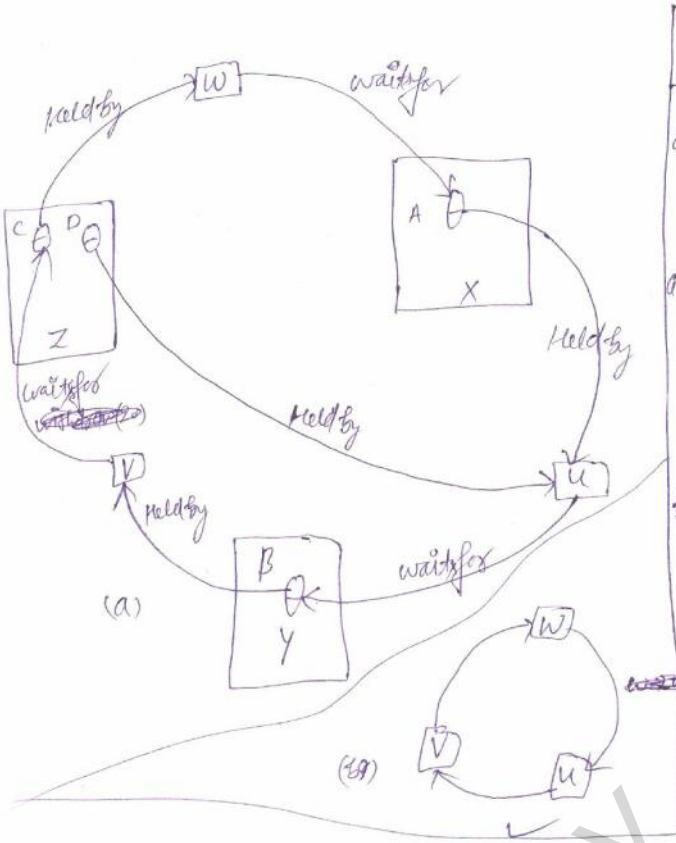
③ Execution - the RMs executes the request, perhaps tentatively (i.e to allow for recovery).

④ Agreement - If appropriate (e.g for transactions), the RM's will reach a Concensus on whether to commit or abort the updates.

⑤ Response - one or More of the RMs return a response to the FE.
the FE will the delivers the Response to client.

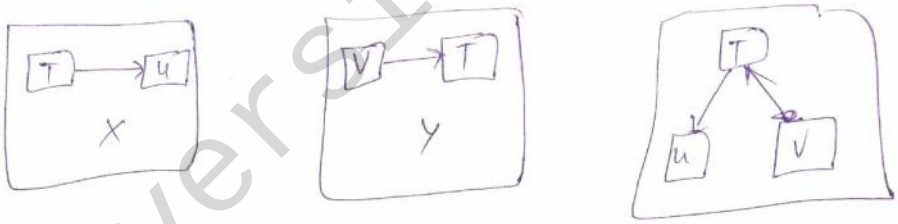
Group Membership views

- An RM may join may leave or fail.
- RMs need to agree on the current membership of the group so that they can be confident not only of who they should be communicating with but also who the other RMs are communicating with.
- this can be achieved by the notion of a (group membership) view, which is simply an agreed list of current RMs (e.g IP addresses processIDs, etc).
- when a RM notices that another RM has failed (or receives a join Request from a new RM), it multicasts a new view to the other RMs.
- Importantly, though, the RMs do not immediately treat the new view as the current view until all RMs in the new view have acknowledged to other RMs that they got it.




| U | V | W |
|---|---|---|
| a.deposit(10) lock D | | |
| | b.deposit(6) lock B at Y | |
| a.deposit(20) lock A at X | | |
| | | c.deposit(3) lock C at Z |
| | b.withdraw(30) wait at Y | |
| | | c.withdraw(20) wait at Z |
| | | a.withdraw(20) wait at X |

Local wait for graphs




WFG of the scenario.

At server Y: U → V (added when U requests B.withdraw(4)) → A, B Managed by X, Y.

Z: V → W (" " V " c.withdraw(3)) → C, D Managed by Z.

X: W → U (" " W " a.withdraw(2)) → Transaction U, V, W involves object A, B, C, D.

① W → U (X → Y)

② W → U → V (Y → Z)

③ W → U → V → W

~~① Server x initiate detection of sending probe <w→u>, node that B is held by V & appends V to the pr~~

① server x initiates detection by sending probe $\langle w \to u \rangle$ to the server of B {server y}

② server y receives probe $\langle w \to u \rangle$, notes that B is held by V & appends V to the probe to produce $\langle w \to u \to v \rangle$. If Notes that V is waiting for c at server z. this probe is forwarded to z.

③ Server z receives probe $\langle w \to u \to v \rangle$, notes c is held by w & appends w to the probe to produce $\langle w \to u \to v \to w \rangle$.

→ This Path Contains cycle. the server detects a deadlock. one of the transactions in the cycle must by aborted to break the deadlock.

| Coordinator of Transaction | Child transactions | Participant | Provisional Commit list | Abort list |
|---|---|---|---|---|
| T | $T_1, T_2$ | yes | $T_1, T_{12}$ | ~~A~~ $T_{11}, T_2$ |
| $T_1$ | | | | |
| $T_2$ | | | | |
| $T_{11}$ | — | yes (aborted) | | |
| $T_{12}, T_{21}$ | — | $T_{1.2}$ by Not $T_{2.1}$* | $T_{2.1}$*, $T_{12}$ | ~~P~~ - |
| $T_{22}$ | — | No (Parent aborted) | | |

*($T_{2.1}$ Parent has aborted)

Replica Managers:

- one Replica Manager per replica
- Receives FE's regst, apply operations to its replicas automatically.

Front Ends: Regst communication (Regst can be Made to a single RM or to Multiple RMs).

- one FE per client
- Receives client's regst, communicate with RM by Message Passing.

Coordination: RM's decide

- The order of regst
  - FIFO - [r, r'] First r then r' regardless of issued by FE
  - Causal ordering - [r, r'] → r happens before r'
  - Total ordering - [r, r'] → Correct r & then r'

## A Concurrency Control Model of Database Systems :-

- For the Purpose of concurrency control, we can view a database system as consisting of three S/w modules-
  - Transaction Manager (TM)
  - Data Manager (DM)
  - Scheduler

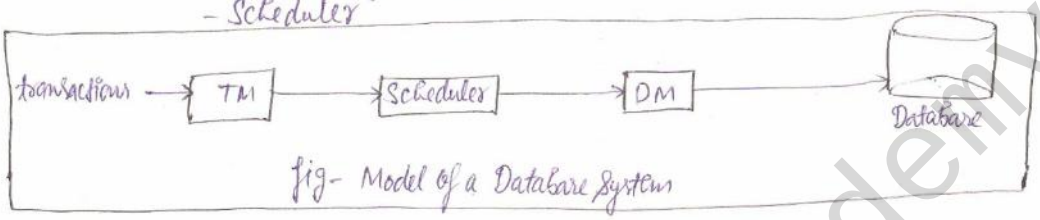



fig- Model of a Database System

- The transaction Manager Supervises the execution of a transaction. It intercepts & executes all the submitted transactions.
- A TM interacts with the DM to carry out the execution of a transaction.
- It is the responsibility of the TM to assign a timestamp to a transaction or issue requests to lock & unlock data objects on behalf of a user.
- Thus we can say that TM is an interface bet$^n$ user & the database system.
- The Scheduler is responsible for Concurrency Control. It grants or releases locks on data objects as requested by a transaction.
- The Data Manager(DM) Manages the database. ~~It operates them~~ It carries out the read-write requests issued by the TM on behalf of a transaction by operating them on the database.
- DM is an interface bet$^n$ the Scheduler & the database. DM executes a stream of transaction actions, directed toward it by the TM.

## The Problem of Concurrency Control :->

- Typically, in a Database System, several transactions are under execution simultaneously.
- Since a transaction preserves database consistency, a database system can guarantee consistency by executing transaction serially. i.e one at time.
- Such a serial execution of transactions is inefficient as- it results in poor response to user requests & poor utilization of system resources.
- efficiency can be improved by executing transactions concurrently, that is by executing read & write actions from several transactions in an interleaved manner.
- Because the actions of concurrently running transactions may access the same data objects, several anomalous situations may arise if the interleaving of actions is not controlled in some orderly way-

Some situations are-

① Inconsistent Retrieval-

- It occurs when a transaction reads some data objects of a database before another transaction has completed with modification of those data objects.
- In such transaction, the former transaction faces the risk of retrieving incorrect values of the data objects.

ex-

S = 1000 (- 500)  C = 500

→ Customer $C_1$ ⇒ reads S into its workspace & subtract 500 from it & writes S.
→ teller $t_1$ ⇒ reads S (= 500) & C (= 500) into its workspace
$C_1$ ⇒ reads C (= 500) into its workspace & add 500 to it, & write back to C (= 1000).
$t_1$ ⇒ output 1000 as the balance

Here, $t_1$ reads S after $C_1$ has modified it & reads C before $C_1$ has modified it resulting the incorrect retrieval of the the total balance.

② Inconsistent update :- occurs when many transactions read & write onto a common set of data objects of a database, leaving the database in an inconsistent state.

Ex (inconsistent state) →

Let, data object – A, B

Consistent assertion → "(A=0) OR (B=0)"

Concurrent Modified transaction –

"$T_1$ : if A=0 then B = B+1"

"$T_2$ : if B=0 then A = A+1".

Now,

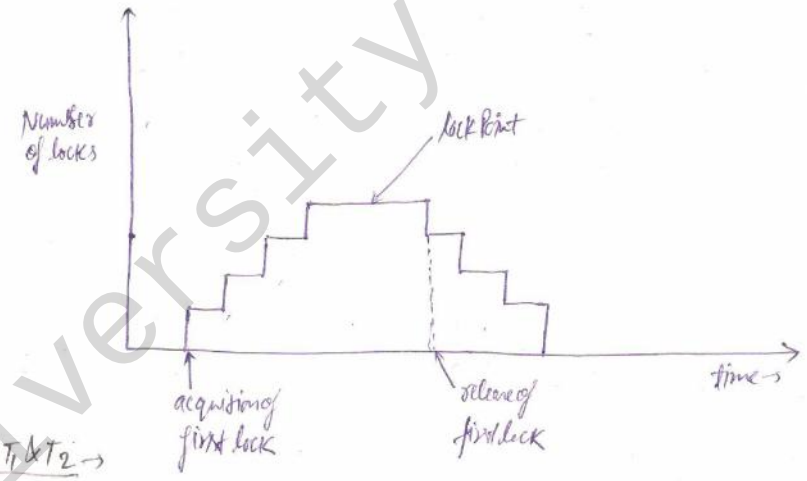
– A Possible execution trace is –

initially (A=0 & B=0)

$T_1$ reads A(=0) & B(=0), since A=0 in $T_1$ so it increment B By 1 & write in the Database (B=1)

$T_2$ reads A(=0) & B(=0), Since B=0, in $T_2$ — — — A = 1 — — — (A=1).

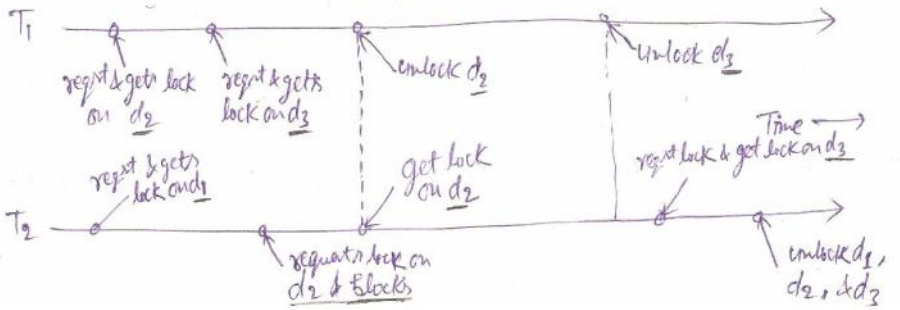
– The final Database state "(A=1) & (B=1)" is inconsistent.




Concurrent execution of $T_1$ & $T_2$ →

$RS(T_1) = \{d_2, d_3\}$ $WS(T_1) = \{d_3\}$

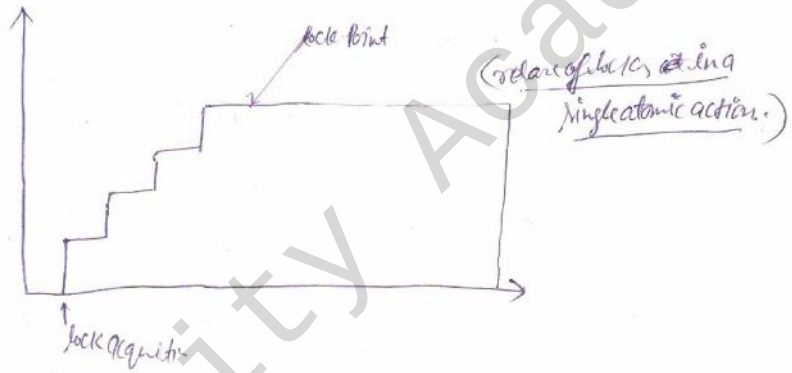
$RS(T_2) = \{d_1, d_2, d_3\}$, $WS(T_2) = \{d_1, d_2, d_3\}$



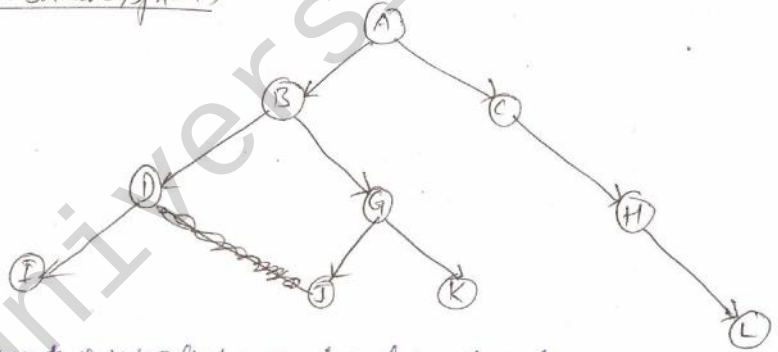
Cascaded Roll-Backs → when a transaction Roll-back (for any reason-a user kills it, the system crashes, or it becomes deadlocked), all the Data objects modified by it, ~~the system~~ are restored their original states.

- In this care, all transaction that have read the backed up data objects must also be rolled back & the data objects modified by them must also be restored & so on.
- this Phenomenon is called the cascaded roll-back.
- 2PL suffers from the problem of cascaded roll-back.

Strict 2PL →




Hierarchical Database System →




→ $T_1$ must the Node B first so that it can lock all the req$^{d}$ data object (D, G & I).
$T_2$ must lock node A befor accessing any other node. befor $T_2$ can lock H, it must first lock node C.

→ $T_i$ can lock data object $R \neq E(T_i)$ (Root Node) iff $T_i$ is holding a lock on R's ancestor.
→ After unlocking a data object, $T_i$ cannot lock it again.
→ $T_i$ can only access those data objects for which it is holding a lock.

# OPTIMISTIC CONCURRENCY CONTROL :→ (KUNG-ROBINSON)

- In optimistic algo, no synchronization is performed when a transaction is executed, but at the end of transaction's execution, a check is performed to determine if the transaction has conflicted with any other concurrently running transactions.
- In the case of conflict, the transaction is aborted, otherwise it is commited.
- In this technique, a transaction always executes (tentatively (Provisionally)) concurrently with other transactions without any synchronization check, but before its writesset written in the database (& become accessible to other transactions), it is validated.
- In validation Phase, it is determined whether actions of the transaction have conflicted with those of any other transaction. If found in conflict, then the tentative writes of the transaction are discarded & the transaction is restarted.
- The execution of transaction is decided into 3 Phase-
  (1) read Phase (2) validation Phase (3) write Phase.

- In read Phase, appropriate data objects are read, the intended computation of the transaction is done, & writes are made on a temporary storage.
- In validation Phase it is checked if the writes made by the transaction violate the consistency of the database.
- If check Passes, then in the write Phase, all the writes of the transaction are made to the database.

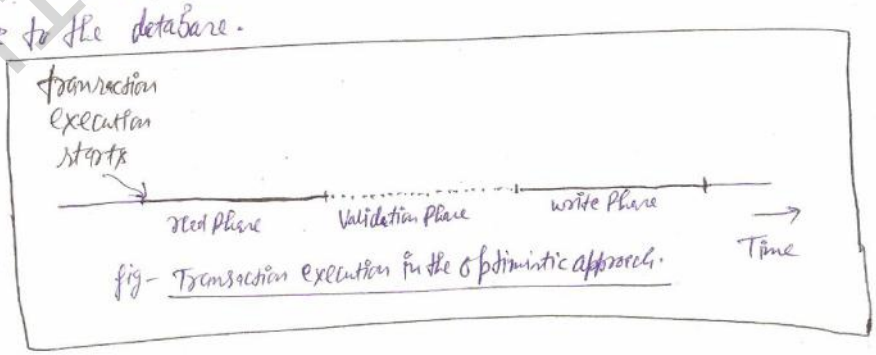
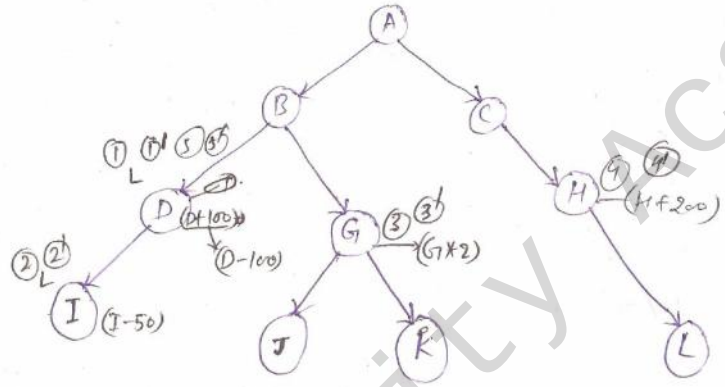
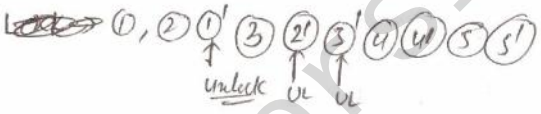
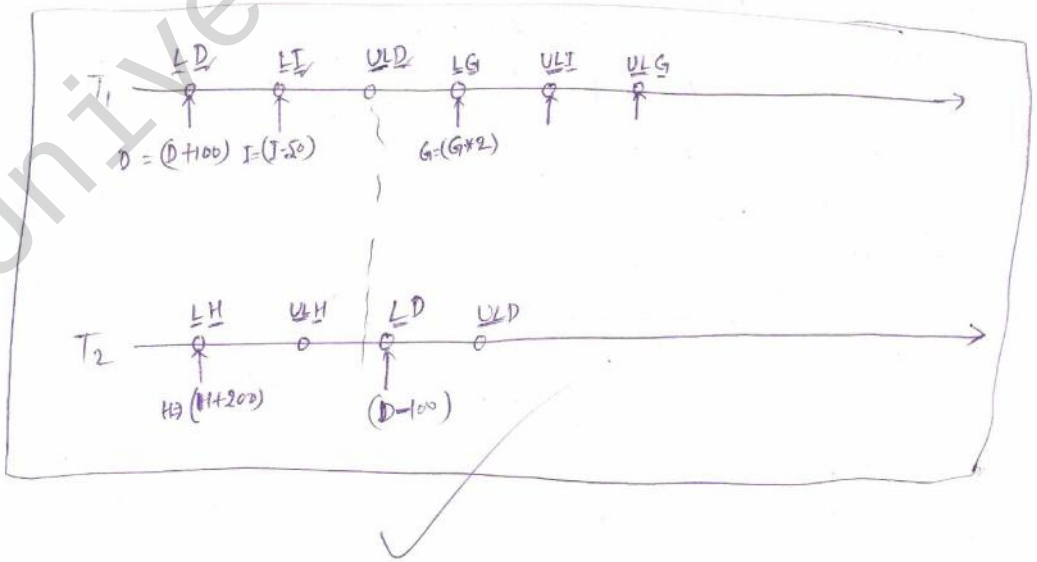



fig- Transaction execution in the optimistic approach.

| $T_1$ | $T_2$ |
|---|---|
| lock D | LOCK H |
| D+100 → D | H+200 → H |
| lock I | Unlock H |
| I-50 → I | lock D |
| unlock D | D-100 → D |
| lock G | unlock D |
| G*2 → G | |
| unlock I | |
| unlock G | |



## Timestamp Based Algorithms →

TM → attaching timestamp to each transaction
DMs → process conflicting opr$^{n}$ in timestamp order

- Timestamp based concurrency Control algorithms, every site maintains a logical clock that is incremented when a transaction is submitted at that site & updated whenever the site receives a message with a higher clock value (every message contains the current clock value of its sender site).

- Timestamps can be used in two ways –
  - Determine the outdatedness of a request with respect to data objects.
  - used to order events (read-write req$^{st}$) with respect to one another.

### ① Basic Timestamp ordering Algo → (BTO)

- The Scheduler at each DM keeps track of the largest timestamp of any read & write processed thus for the each data object.

let
① – { R-ts (object) – read-timestamp. }
      { W-ts (object) – write timestamp. }

② { read (X, TS) } { read & write request · with timestamp TS on a data
  { write (X, V, TS) } { object X. In write operation, V is the value to be assigned to X. }

- A read (X, TS) request is handled in the following manner –
  - If TS $<$ W-ts(x), then the read req$^{st}$ is rejected & the corresponding transaction is aborted, otherwise it is executed & R-ts(x) is set to Max{R-ts(x), TS}.

- A write (X, V, TS) request is handled in the following manner –
  - If TS $<$ R-ts(x) (of Transaction (T)) or TS $<$ W-ts(x), then write request is rejected otherwise it is executed & W-ts(x) is set to TS.

- If a transaction aborted it is ~~aborted~~ started with new Timestamp.

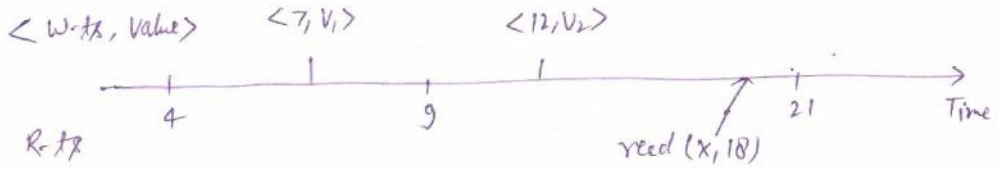
② Thomas write Rule (TWR) → (Suitable for only write actions)

- for a write (X, V, TS), if TS < W-ts(x), the TWR says that instead of rejecting the write simply ignore it.
- this is sufficient to enforce synchronization among writes because the effect of ignoring an obsolete (No longer true) write request is the same as executing all writes in their timestamp ordering.
- TWR takes care of only write-write synchronization.
- TWR is an improvement over the BTO algorithm because it reduces the No of transaction aborts.

③ Multiversion Timestamp ordering Algo → (MTO)

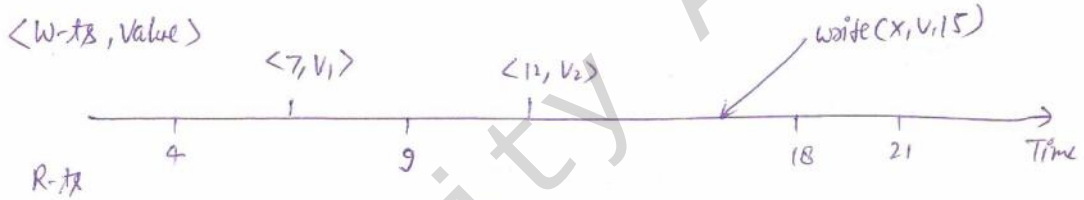
- A history of a set of R-ts's & (W-ts, value > pairs (called versions) is kept for each data object at the respective DM's.
- The R-ts's of a data object keep track of the timestamps of all the executed read operations, & the versions keep track of the timestamps & the value of all the executed write opr^n.
- Read & write req^t are executed in following manner.
  - A read (X, TS) request is executed by reading the version of X with the largest timestamp less than TS & adding TS to the X's set of R-ts's. A Read request is never rejected.
  - A write (X, V, TS) request is executed in the following way.
    - If there exist a R-ts(x) in the interval from TS to the smallest W-ts(x) that is larger than TS, then the write is rejected, Otherwise it is accepted & a new version of x is created with timestamp TS.

ex→ (a) a read (x,18) is executed by reading the version <12, $V_2$> & the resulting History is shown as –




(a)

ex (b) write (x, V, 15) is rejected because a read with ~~a~~ timestamp 18 has already been executed. However, a write (x, U, 22) is accepted & is executed by creating a version <22, V> in the history.




(b)

④ Conservative timestamp ordering Algorithm → (CTO)

- It eliminates the abort & restarts of transactions by executing the requests in strict timestamp order at all DM's.
- A Scheduler processes a request when it is sure that there is no other request with a smaller (older) timestamp in the system.
- Each Scheduler maintains 2 queues –
  - R-queue
  - W-queue

  } per TM, these queue hold read & write requests.
- A TM sends requests to Schedulers in timestamp order & the communication medium is order preserving.
- A Scheduler Puts a new read or write request in the corresponding queue in timestamp order.

- This algo executes read & write actions in the following way -

① A read(x, TS) request is executed in following way -

* If every W-queue is nonempty & the first write on each W-queue has a timestamp greater the TS, then the read is executed, otherwise the read(x, TS) request is buffered in the R-queue.

② A write(x, v, TS) request with timestamp TS is executed in following Manner.

* If all R-queues & all W-queues are nonempty & the first read on each R-queue has a timestamp greater than TS & the first write on each W-queue has a timestamp greater than TS, then the write is executed, otherwise the write (x, v, TS) request is buffered in the W-queue.

③ When any read or write req^t is buffered or executed, buffered req^t are tested to see if any of them can be executed.

Problem with CTO →

① Termination is not guaranteed.

② The algorithm is overly conservative.