Fundamental Models

- Some of the properties are common in all architectural models.
- The fundamental models are concerned with the formal description of such properties.
- A model contains only the essential ingredients that we need to consider in order to understand and reason about some aspects of a system's behaviour.
- A system model has to address the questions:-
  - What are the main entities in the system?
  - How do they interact?
  - What are the characteristics that effect their individual and collective behaviour?
- The purpose of a model is :-
  - To make explicit all the relevant assumptions about the systems we are modeling.
  - To make generalizations concerning what is possible or impossible, given those assumptions.
- The generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed.

The fundamental models are classified as—

(i) The interaction model
(ii) The failure model
(iii) The security model

(1) Interaction model :→

- Computation occurs within processes; the processes interact by passing messages, resulting in communication (i.e. information flows) and coordination (synchronization and ordering of activities) between processes.
- They send and receive messages among each other.
- The interaction model must be designed in such a manner that the communication between various processes is not delayed.

There may be various servers providing specific services.
- various file servers
- Domain name server
- Network information service provider

- Many processes may communicate with each other for specific purpose such as video conferencing.
- The rate at which each process proceeds and the timing of the transmission of messages between them can not in general be predicted.
- Each process has its own state and it is impossible to maintain a single global notion of time.

Important factors in interaction model -

(a) Performance of communication channels
- delay
- Bandwidth
- Jitter

(b) computer clocks and timing events

Variants of the interaction model

(a) Synchronous distributed systems
(b) Asynchronous distributed systems

~~(a) Synchronous distributed systems~~

- In a distributed system it is hard to set time limits on the time taken for process execution, message delivery or clock drift.
- Two opposing extreme positions provide a pair of simple models

(a) Synchronous distributed systems →

Synchronous distributed system to be one in which the following bounds are defined -

- the time to execute each step of a process has known lower and upper bounds
- Each message transmitted over a channel is received within a known bounded time
- Each process has a local clock whose drift rate from real time has a known bound.

* The term clock drift rate refers to the relative amount that a computer clock differs from a perfect reference clock.

- For eg. classical PARAM model

(b) Asynchronous distributed system :→

An asynchronous distributed system is one in which there are no bounds on :-

- Process execution speeds - for example, one process step may take only a picosecond and another a century; all that can be said is that each step may take an arbitrarily long time.

- Message transmission delays - for example, one message from process A to process B may be delivered in negligible time and another may take several years.
  - In other words, a message may be received after an arbitrarily long time.

- Clock drift rates - The drift rate of a clock is arbitrary.

- Internet is the exactly model of asynchronous distributed system.
- The asynchronous model allows no assumptions about the time intervals involved in any execution.
- In internet, there is no intrinsic bound on server or network load and therefore on how long it takes.
- For example to transfer a file using ftp.
- Some times an email message can take days to arrive.

~~for ex: classical PRAM model~~

(ii) Failure model :→

- The correct operation of a distributed system is threatened whenever a fault occurs in any of the computers on which it runs or in the network that connects them.
- This model defines and classifies the faults.
- This provides a basis for the analysis of their potential effects and for the design of systems that are able to tolerate faults of each type while continuing to run correctly.

Types of failures -

(a) Omission failures →

- Refers to cases when a process or communication channel fails to perform actions that it is supposed to do, it is called as process omission failure and communication omission failures respectively.

(b) Arbitrary failures →

- The term arbitrary is used to describe the worst possible failure semantics, in which any type of error may occur.
- For example, a process may set wrong values in its data items, or it may return a wrong value in response to an invocation.
- Therefore arbitrary failures in processes cannot be detected by seeing whether the process responds to invocations because it might arbitrarily omit it wrong

Class of failure	Affects	Description
Fail-stop	Process	Process halts and remains halted; other processes may detect this state
Crash	Process	" but other processes may not be able to detect this state
Omission	channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
send omission	Process	A process completes a send, but the message is not put in its outgoing message buffer.
Receive	Process	A message is put in a process's incoming message buffer, but omission that process does not receive it.
Arbitrary	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Omission and arbitrary failures

Timing failures –

Timing failures are applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

(d) Masking failures –

- Each component in a distributed system is generally constructed from a collection of other components.
- It is possible to construct reliable services from components that exhibit failures.

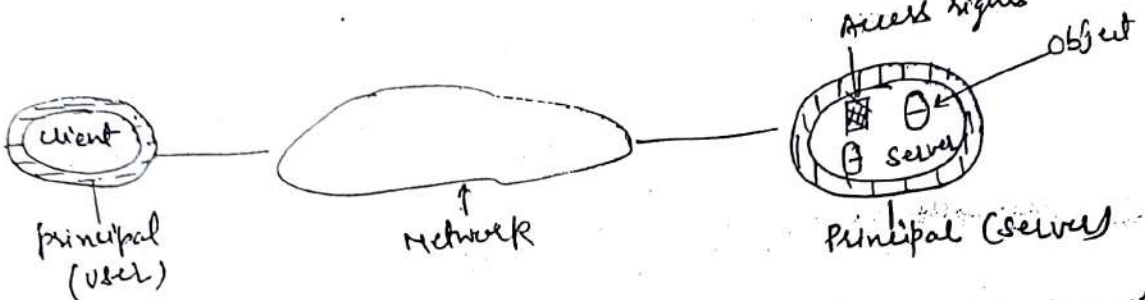
(iii) Security Model →

- The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents.
- Security model defines and classifies the forms that such attacks may take, providing a basis for the analysis of threats to a system and for the design of systems that are able to resist them.
- The security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access.

(a) Protecting objects :-

- Objects are intended to be used in different ways by different users.

- For example, some objects may hold a user's private data such as their mailbox, and other objects may hold shared data such as web pages.
- To support this, access rights specify who is allowed to perform the operations of an object - for example, who is allowed to read or to write its state.



A principal (user) is a user or process that invoke request

A principal (server) is a process or server that give responses.

(b) Securing processes and their interactions →

- We can secure interaction by securing channel with the help of VPN, SSL, HTTPS etc.
- For data integrity and authentication we use cryptography. Cryptography can also be used for confidentiality.

Limitation of Distributed system

There are basically two broad limitations of distributed system.

(i) Absence of global clock i.e. no synchronization among processes.

(ii) Absence of dynamic memory i.e. at a particular time a process can only get partial & coherent state of complete & incoherent state of the distributed system.

coherent - recorded state of all the processes at any given time.

(1) Absence of a Global clock :→

- In a distributed system there exists no system wide common clock (Global clock).
- This problem can be solved by either having a clock common to all the computers (processes) in the system or having synchronized clock, one at each computer.
- Unfortunately these can not work because
  - In the case of global clock, two different processes can observe a global clock value at different instants due to unpredictable message delays.
  - In second case physical clocks can drift from the physical time and the drift rate may vary from clock to clock.

Impact of the absence of global time :-

- Due to the absence of global time, it is difficult to reason about the temporal order of events in a distributed system.
- Hence algorithm for a distributed system are more difficult to design and debug compared to algorithms for centralized system.
- The absence of a global clock makes it harder to collect up-to-date information on the state of the entire system.

(ii) Absence of shared memory :→

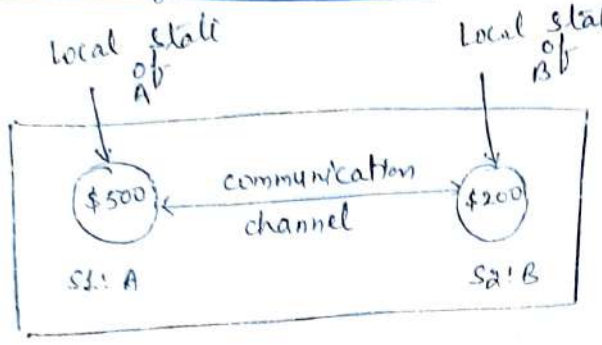
- Computers in distributed system do not share common memory, therefore an up-to-date state of the entire system is not available to any individual process.
- Up-to-date state of the system is necessary for reasoning about the system's behaviour, debugging, recovering from failures.
- A process in a distributed system can obtain a coherent but partial view of the system or a complete but incoherent view of the system.
- A complete view encompasses the local view (local state) at all the computers and any messages that are in transit in the distributed system.
- A complete view is also referred to as a global state.
- Similarly, the global state of a distributed computation encompasses the local states of all the

सामंजस्य पूर्ण

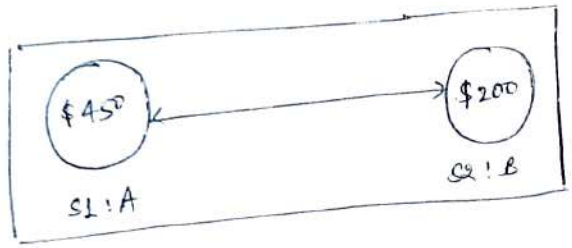
processes and any messages that are in transit between the process.

Because of the absence of a global clock, obtaining a coherent global state of the system is difficult.

Ex.



(a) S1 & S2 are two entities maintaining bank a/c of A and B



(b) states immediately fund transfer from A and before reaching to B

(c) states before fund transfer from A and after reaching to B

There are two schemes that implement an abstraction of virtual time to order events in a distributed system.

(i) Lamport's logical clocks :→

- Lamport proposed the following scheme to order events in using logical clocks.

Definitions -

- Under certain conditions, it is possible to ascertain the order in which two events occur based solely on the behaviour exhibited by the underlying computation.
- There is happened before relation that orders events based on the behaviour of the underlying computation.

Happened Before Relation (→) :-

- The happened before relation defines whether two events are causally related or not.
- The relation → is defined as follows -
  - ↪ $a \to b$, if a and b are events in the same process and a occurred before b.
  - ↪ $a \to b$, if a is the event of sending a message m in a process and b is the event of receipt of the same message m by another process.
  - ↪ If $a \to b$ and $b \to c$, then $a \to c$, i.e. "→" relation is transitive.
- Events that can be ordered by '→' is referred to as Causal affects.

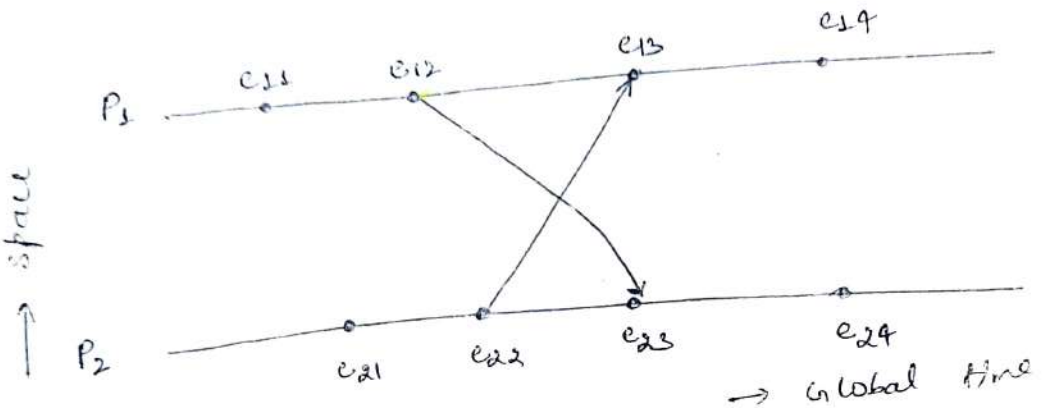
Causal — A has caused B

Causally Related Events → Event a causally affects event b if $a \rightarrow b$.

Concurrent Events → Two distinct events a and b are said to be concurrent ($a \| b$) if $a \not\rightarrow b$ and $b \not\rightarrow a$.

- That means concurrent events do not causally affect each other.

Ex -



- According to above diagram process P_1 contains event e_{11}, e_{12}, e_{13} and e_{14}.
- And e_{21}, e_{22}, e_{23} & e_{24} are the events of process P_2.
- P1 & P2 process represent message transfer b/w the processes.
- We can see that $e_{22} \rightarrow e_{13}$, $e_{23} \rightarrow e_{14}$ and therefore $e_{22} \rightarrow e_{14}$.
- That means event e_{22} causally affects event e_{14}.
- From above diagram all the events move only forward along the time axis in the space-time region.

Logical Clocks

- In order to realize the relation $\rightarrow$, Lamport introduced the following system of logical clocks.
- There is a clock C_i at each process P_i in the system.
- The clock C_i can be thought of as a function that assigns a number $C_i(a)$ to any event a, called the time stamp of event a, at P_i.
- The numbers assigned by the system of clocks have no relation to physical time, and hence the name logical clocks.
- The logical clocks take monotonically increasing values.
- The time stamp of an event is the value of the clock when it occurs.

Conditions statisfied by the system of clocks →

if $a \rightarrow b$, then $C(a) < C(b)$

The happened before relation $\rightarrow$ can now be realized by using the logical clocks if the following two conditions are met:

(i) [C1] For any two events a and b in a process P_i, if a occurs before b, then

$C_i(a) < C_i(b)$

[C2] If a is the event of sending a message m in process P_i and 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 clocks guarantee that the clocks satisfy the correctness conditions C1 and C2:

→[IR1] clock C_i is incremented between any two successive events in process P_i:

$$C_i := C_i + d \quad (d>0) \qquad \text{— (1)}$$

If a and b are two successive events in P_i and $a \to b$, then $C_i(b) = C_i(a) + d$.

→[IR2] If event a is the sending of message m by process P_i, then message m is assigned a time stamp $t_m = C_i(a)$ (value of $C_i(a)$ is obtained after applying rule IR1).

On receiving the same message m by process P_j, C_j is set to a value greater than or equal to its present value and greater than t_m:

$$C_j := \max(C_j, t_m + d) \quad d>0 \qquad \text{— (2)}$$

Note that message receipt event at P_j increments C_j as per rule IR1.

- The updated value of C_j is used in equation and usually d in equation ① & ② has a value of 1.

→ Lamport's happened before relation, $\rightarrow$, defines an irreflexive partial order among the events.

- The set of all the events in a distributed computation can be totally ordered ($\Rightarrow$) using the above system of clocks as follows:

i) If a is any event at process P_i and b is any event at process P_j then $a \Rightarrow b$ if and only if either

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

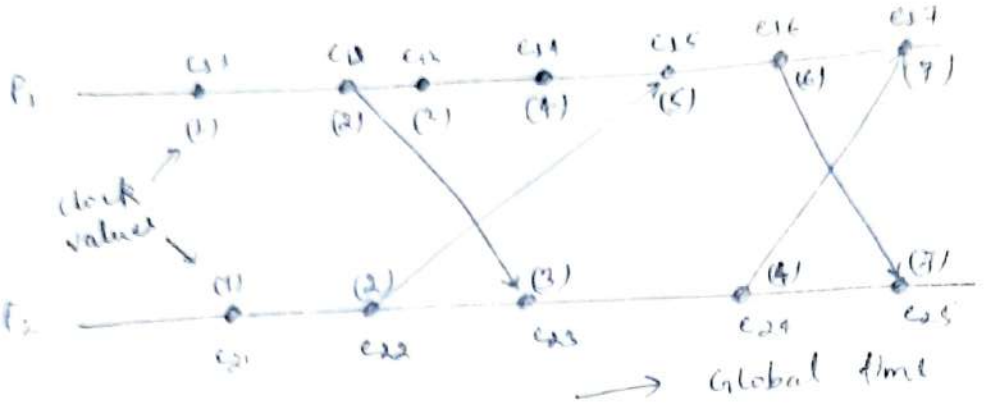
or $C_i(a) = C_j(b)$ and $P_i \prec P_j$

where $\prec$ is any arbitrary relation that totally orders the processes to break ties.

- A simple way to implement $\prec$ is to assign unique identification numbers to each process and then $P_i \prec P_j$, if $i < j$.

Ex - In diagram shown on next page shows how logical clocks are updated under Lamport's scheme. Both the clock values CP_1 and CP_2 are assumed to be zero initially and d is assumed to be 1.

- Let b the event of process P_j which causes C_j to



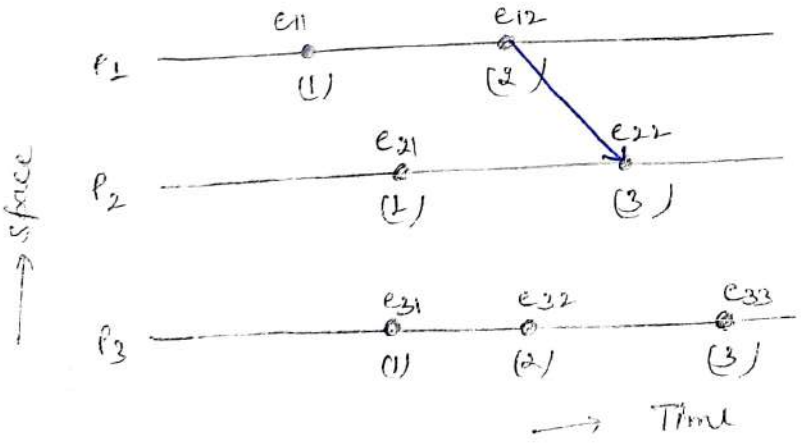
- Similarly, e_{21} and e_{22} are two events in P_2 resulting in $CP_2 = 2$ due to IR1.
- e_{16} is a message send event in P_1 which increments CP_1 to 6 due to IR1.
- The message is assigned a timestamp = 6.
- The event e_{25}, corresponding to the receive event of the above message, increments the clock CP_2 to 7 $(\max(4+1, 6+1))$ due to rule IR1 and IR2.
- Similarly e_{17} is a send event in P_2 is assigned a timestamp of 7 corresponding to the receive event e_{26} of the message, increments the clock CP_1 to 8 $(\max(6+1, 7+1))$ due to rules IR1 and IR2.

A Limitation of Lamport's clocks →

- According to condition, if $a \rightarrow b$ then $C(a) < C(b)$.
- However, the reverse is not necessarily true if the events have occured in different processes.
- That means if a and b are events in different processes and $C(a) < C(b)$ then $a \rightarrow b$ is not necessarily true.
- Event a and b may be causally related or may not be causally related.
- Thus Lamport's system of clocks is not powerful enough to capture such situations.

Ex - Figure shows a computation over three processes.
Clearly, $C(e_{11}) < C(e_{22})$ and $C(e_{11}) < C(e_{32})$

- According to figure event e_{11} is causally related to event e_{22} but not to event e_{32}, since a path exists from e_{11} to e_{22} but not from e_{11} to e_{32}.



- Initially clock values are zero and value of d is equal to 1 (by according to Condition).

In Lamport's system of clocks, we can guarantee that if $C(a) < C(b)$ then $b \not\rightarrow a$.

- However we can not say whether events a and b are causally related or not. (whether path exists b/w a and b that moves only forward along the time axis) by just looking at the timestamps of the events.
- The reason for the above limitation is that each clock can independently advance due to the occurrence of local events in a process.
- And the Lamport's clock system can not distinguish between the advancements of clocks due to local events from those due to the exchange of messages between processes.
- Therefore using the timestamps assigned by Lamport's clocks, we can not reason about the causal relationship b/w two events occurring in different processes by just looking at the timestamps of the events.

Vector Clocks

- The system of vector clocks was proposed by Fidge and Mattern
- This concept is for keeping track of transitive dependencies among processes for recovery purpose.
- Let n be the number of processes and each process P_i is equipped with a clock C_i, which is an integer vector of length n.
- The clock C_i can be thought of as a function that assigns a vector $C_i(a)$ to any event a.
- $C_i(a)$ is referred to as the time stamp of event a at P_i.
- $C_i[i]$, the ith entry of C_i, corresponds 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.
- More specifically, at any point in time, the jth entry of C_i indicates the time of occurrence of the last event at P_j which 'happened before' the current point in time at P_i.

The implementation rules for the vector clocks are as follows -

[IR1] clock C_i is incremented between any two successive events in process P_i

$$C_i[i] := C_i[i] + d \qquad (d > 0) \quad - (3)$$

[IR2] If event a is the sending of the message m by process P_i, then message m is assigned a vector

timestamp $tm = C_i(a)$; on receiving the same message by process P_j, C_j is updated as follows -

$$\forall k.\ C_j[k] := \max(C_j[k], tm[k]) \quad \text{--- (4)}$$

In rule IR1, we treat message send and message receive by a process as events

- In rule IR2, a message is assigned a timestamp after the sending process has implemented its clock due to IR1
- If it is necessary to allow for propagation time for a message, then IR2 can be performed after performing the following step.

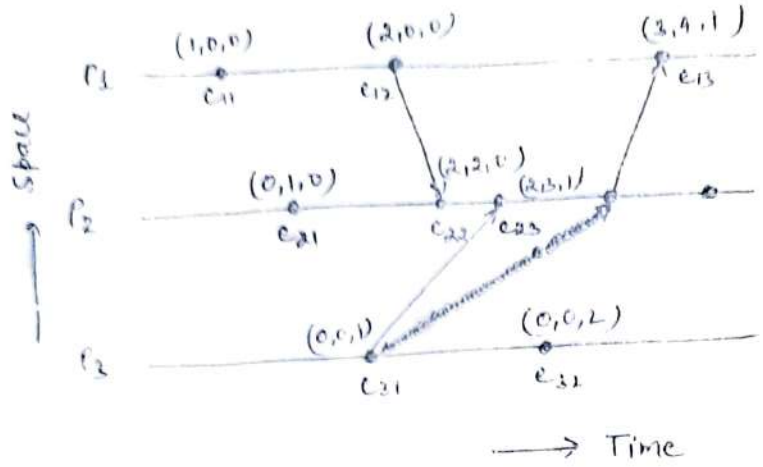
$$\text{If } C_j[i] \leq tm[i] \text{ then } C_j[i] := tm[i] + d \quad (d > 0)$$

However, the above step is not necessary to relate events causally and hence, we do not make use of it in the following discussion.

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

This proof is obvious because no process $P_j \neq P_i$ can have more up to date knowledge about the clock value of process P_i than P_i itself. Clocks are monotonically nondecreasing.

Q. Next figure shows how clocks advance and the different cases of time events in a system using vector clocks ($d = 1$ & all clock values are initially zero).



- Event e_{11} of process P_1 that causes $C_1[1]$ to be incremented to 1 due to IR1.
- e_{12} is a message send event in P_1 which causes $C_1[1]$ to be incremented to 2 due to IR1.
- e_{22} is a message receive event in P_2 that causes $C_2[2]$ to be incremented to 2 due to IR1, and $C_2[1]$ to be set to 2 due to IR2.
- Event e_{23} a receive event in P_2, causes $C_2[2]$ to be incremented to 3 due to IR1, and $C_2[3]$ to be set to 1 due to IR2.
- e_{32} is a send event in P_2 and e_{13} is the corresponding receive event.
- $C_1[3]$ is set to 1 due to IR2, and process P_1 has learned that the local clock value of P_3 is at least 1 through a message from P_3.

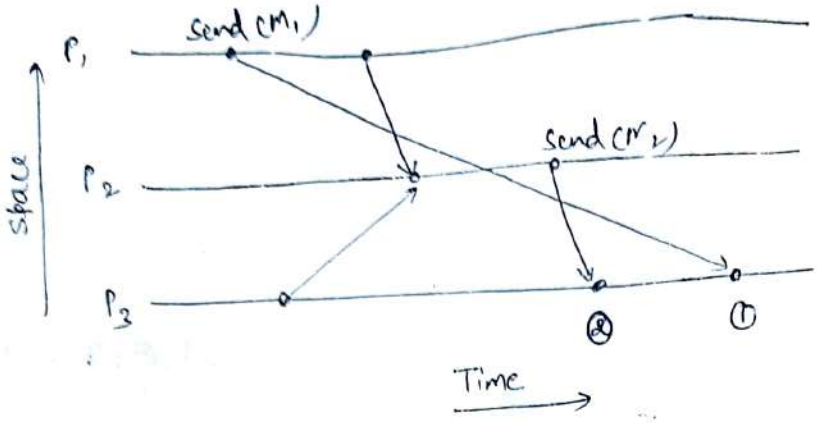
29/25. ✓
07/20 ✓

Causal Ordering of Messages

- The causal ordering of messages was first proposed by Birman and Joseph.
- The causal ordering of messages deals with the notion of maintaining the same causal relationship that holds among "message send" event with the corresponding "message receive" events.
- In other words if send(M_1) → send(M_2) where send(M) is the event sending message(M) then every recipient of both messages M_1 and M_2 must receive M_1 before M_2.

a distributed system the causal ordering of messages is not automatically guaranteed.

for eg. figure shows a violation of causal ordering of messages in a distributed system.



- In this example, send $(M_1) \rightarrow$ send (M_2). However, M_2 is delivered before M_1 to process P_3. (The numbers circled indicate the correct causal order to deliver messages)

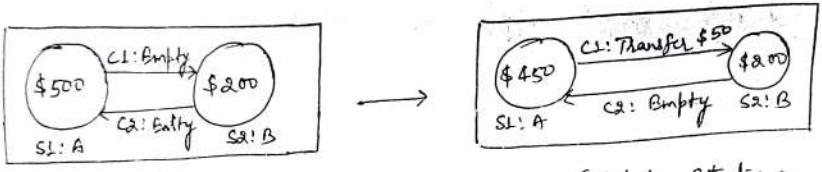
Techniques for The Causal Ordering of Messages

- Techniques for the causal ordering of messages are useful in developing distributed algorithms and may simplify the algorithms themselves.
- for example, for applications such as replicated data base systems it is important that every process in charge of updating a replica receives the updates in the same order to maintain the consistency of the database.

Global State

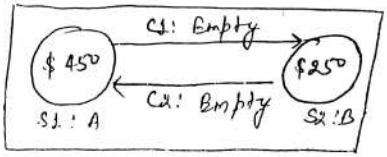
- A global state, GS of a system is a collection of local states of its sites; That is $GS = \{LS_1, LS_2 \ldots$ where n is the no. of sites in the system.
- Note that any collection of local states of sites need represent a consistent global state.

According to diagram, suppose the state of account A at site S1 was recorded when the global state was 1.



Global State: 1 Global State: 2

- Now assume that the global state changes to 2, and the state of communication channels C1 and C2 and of account B are recorded when the global state is 2.
- Then the composite of all the states recorded would show account A's balance as $500, account B's balance as $200 and a message in transit to transfer $50.
- Hence an extra amount of $50 would appear in the global state.



Global State: 3