Data Analytics

→ Weightage :

Quiz - 1	→	3 - 15 %
Quiz - 2	→	3 - 15 %
Mid sem	→	10 - 55 %
End sem	→	10 - 55 %

→ Book : Han, Kamber and Pei.

Distributed Systems

Topics

1.> Foundations :

↳ Characterization of Distributed Systems, System models, Network and Inter-Networking, Inter-process communication

2.> Logical Times :

↳ Framework for a system of logical clocks, Scalar Time, vector Time, Efficient implementation of vector clocks, synchronization of physical clocks, Network Time Protocol (NTP)

3.> Global State and Snapshot Recording Algorithms :

↳ System model and Definition, snapshot Algorithm for FIFO Channel.

4.> Middle ware :

↳ Distributed objects and Remote Method Invocation (RMI)

5.> Termination detection :

↳ Termination detection using distributed snapshots, a spanning tree based termination detection algorithm.

6.> Distributed Mutual Exclusion Algorithms

↳ Lamport's Algorithm, Rickard Agarwala Algorithm, Singhal's Dynamic Information-Structure Algorithm, Quorum-based mutual exclusion algorithm, Maekawa's Algorithm

7.) Deadlock Detection in Distributed Systems:
↳ Models of deadlock, Knapp's classification of distributed deadlock detection algorithms, Mitchell and Merritt algorithm for single resource model

8.) Consensus and Agreement Algorithm:
↳ Problem definition, Agreement in a failure-free system (synchronous or asynchronous), Agreement in synchronous systems with failures, Agreement in asynchronous message passing system with failures

9.) Distributed File Systems.

BOOKS

1.) Ajay D. KshemKalyani and Mukesh Singhal: "Distributed computing Principles; Algorithms and systems"; Cambridge Univ. Press, 2008

2.) Sukumar Ghosh: "Distributed systems and Algorithmic Approach"; Chapman and Hall/CRC, II Edi. 2015
I Edi. 2007.

3.) M.L. Liu: "Distributed computing Principles and Applications"; Pearson, 2004

4.) George Coulouris, Jean Dollimore, Tim Kindberg and Gordan Blair: "Distributed systems concepts and Design"; Pearson, V Edition 2011

5.) Mukesh Singhal and Niranjan G. Shivaratri: "Advanced concepts in operating systems"; Tata McGraw Hill, '94 & 2010

6.) Andrew S. Tanenbaum and Maarten Van Steen : "Distributed Systems Principles and Paradigms" ; II Edi PHI 2007
III Edi. 2017

7.) Sape Mullender (Edited) : "Distributed systems" ; II Edi A/W

8.) Hagit Attiya and Jennifer Welch : "Distributed Computing Fundamentals, simulation and Advanced Topics" ; II Edi 2004 Wiley Int.

9.) Gerard Tel : "Introduction to Distributed Algorithms" ; Cambridge Univ. Press 2000

10.) Nancy Lynch : "Distributed Algorithms"

11.) Joel M. Crichlow : "Distributed systems computing over Network" ; PHI 2004

12.) Vijay K. Garg : "Elements of Distributed computing" ; Wiley 2002

⇒ Notes, can be downloaded by a professor : Notes on Theory of Distributed systems → cs-www.cs.yale/edu/homes/aspnes/classes/465/notes.pdf

Assignments on (To be done in Java)

1) Socket Programming : connection-oriented / connectionless Datagram socket ; stream-mode socket ; API → Book by Liu

2) Remote Method Invocation.

3) MPI-3 (Message Passing Interface) Programming → McGraw Hill book, open MP
↳ Michael Quinn (2004)

** Erlang Language for concurrent programs & Distributed programs.

** top500.com → website to be visited.

Definitions

→ A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.

→ A distributed system is a collection of independent computers that appear to the user as a single coherent computer.

→ A distributed system is one in which the failure of a computer system you did not know even existed can render your computer unusable. - Leslie Lamport

→ A term that describes a wide range of computers from weakly coupled systems such as WANs to strongly coupled systems such as LANs and to very strongly coupled systems such as multi-processor systems.
↳ (NUMA systems)

→ A collection of computers that do not share common memory or common physical clock, that communicate by a message-passing over a communication network, where each computer has its own memory and runs its own OS.
e.g.: Mobile phone networks, corporate networks, factory networks, campus networks, home networks, in-car networks

Significant consequences of Distributed Systems

→ concurrency,
no global clock and independent failures.

Features of Distributed systems

→ No common physical clock & ~~Incorrect ordering~~
→ Inherent Asynchrony among processors (because they have their own clocks)
→ No shared memory, so requires message passing
→ Geographical separation: Network of workstations (NOWs)/ Cluster of workstations (COWs)
→ Processors need not be on WAN.
→ Autonomy and heterogeneity of processors.
→ Processors can be loosely coupled
→ Resource sharing, database sharing, etc.

Motivation for Distributed systems.

* Bank transactions (may be intra-city or inter-city), corporate file systems, etc. are all distributed.
→ Inherently Distributed computations.
↳ Money transfer in banking.
↳ Reaching consensus among parties that are geographically distinct.

→ Resource sharing.
↳ Peripherals and complete datasets in databases and special libraries (cannot be replicated at every place, and also cannot be put at a single site).

→ Access to geographically remote data and resources.

→ Enhanced reliability.
↳ Possible because of replicating resources and execution (if one process fails, others will not, they can take over although a delay might happen).

↳ Reliability:
- Availability: Resources should be accessible at all times.
- Integrity: value & state of resources should be correct.

- Fault Tolerance : Ability to recover from system failures.

→ Increased performance / cost ratio.

→ Scalability

→ Modularity and incremental expandability.
↳ Modularity as in, in a distributed system, computers can be grouped with each group working on a different problem.

Relation To Parallel Multi-processor and Multi-computer Systems

* Omega-Network & Butterfly-Network for 16 inputs (4 stages, 4 switches in each) (Ass-1 on this Read Kshankalyani)

* Multiprocessor : All processes are homogeneous & UMA and shared memory.

* Assignment 1 : Omega & Butterfly Networks for 16 inputs (4 stages, 4 switches in each stage).

→ In multi-computer system, memory space is local to computers. (NUMA).

* Toroidal & hypercube connections of processors.

→ Architecture of parallel systems.

↳ Taxonomies for multiprocessor / multicomputer systems

→ Characteristics of parallel system:

↳ A multiprocessor system is a parallel system in which multiple processes have direct access to shared memory. All processors usually run on the same OS. (UMA).

- These processors can be on a:
  * bus
  * multi-stage switch : Omega Network and Butterfly Network.

↳ A multicomputer parallel system is a parallel system in which multiple processors do not have access to shared memory. (NUMA)

- The memory of the multiple processors may or may not form a common address space.
- The processors are in close proximity and are usually very tightly coupled and also, connected by an inter-connection network.
- The processes communicate either via a common address space or via message passing.

- A multiprocessor system that has a common address space usually corresponds to NUMA architecture.

↳ Array Processors

- These are very tightly coupled and have a common system clock.
- Used for Digital signal & Image Processing applications.

→ Flynn's Taxonomy.

↳ SISD : Single Instruction Single Data
↳ SIMD : Single Instruction Multiple Data
↳ MISD : Multiple Instructions Single Data
↳ MIMD : Multiple Instructions Multiple Data

Coupling, parallelism, concurrency & granularity

granularity ↓ computation to communication ratio.

* low granularity : tightly coupled systems.

→ Degree of coupling among a set of modules, whether hardware or software is measured in terms of the inter-dependency and binding and/or homogeneity among the modules.

* Tightly coupled multiprocessors with UMA shared memory:
  Eg: Switch-based NYU ultra computer.

* Tightly coupled multiprocessor with NUMA shared memory or that communicate by message passing:
  Eg: Silicon SGI origin 2000 ; Sun ultra HPC servers (NUMA shared memory)

* Loosely coupled multicomputers (without shared memory), physically co-located:
  ↳ Bus-based or using a more general communication network
  ↳ Processes may be heterogeneous.

** Distributed systems involve both parallelism and concurrency.

* Parallelism: Divide problem into modules
Concurrency: Each process progress autonomously.

→ Parallel and distributed programs are concurrent.
↳ In distributed computing, a program may need to cooperate with other programs to solve a problem.
↳ In parallel computing, a program is one in which multiple tasks cooperate closely to solve a problem.
↳ In concurrent computing, a program is one in which multiple tasks can be in progress at any instant.

→ Granularity of a program is the ratio of the amount of computation to the amount of communication within the parallel / distributed program.
↳ Fine-grain programs: Frequent communication
: Tightly coupled systems.
↳ Coarse-grain applications/programs: Less frequent communicat"
: Loosely coupled systems.

Design Issues and Challenges

→ Primary issues in the design of the distributed system:
↳ Providing access to remote data in the face of failures.
↳ File system design
↳ Directory structure design.

* Design Forces

→ Categorization of Design and Challenges:

1. Having greater component related to system design and OS design.
2. Having greater component related to algorithm design.
3. Emerging from recent technology advances and/or driven by new applications.

There is some overlap between these categories.

Distributed system Challenges from a System Perspective

→ Communication:
  - ↳ RPC (Remote Procedure Calls)
  - ↳ ROI (Remote Object Invocation) - In object-oriented
  - ↳ Message-oriented communication vs. stream-oriented communication.

→ Processes:
  - ↳ Management of processes and threads at clients and servers.
  - ↳ code migration (code movement from one system to another)
  - ↳ Design of software and mobile agents.

→ Naming.
  - ↳ Easy to use and robust schemes for names, identifiers and addresses.
  - ↳ Naming in mobile systems

→ synchronization
↳ Mutual exclusion
↳ synchronizing physical clocks and devising logical clocks that capture the essence of physical time.

→ Data Storage and Access.
↳ For accessing the data in a fast and scalable manner across the network.

→ Consistency and Replication.
↳ To provide fast access to data and to provide scalability.

** cache-coherence protocols.
↳ Required in case of file sharing or so.

→ Fault Tolerance.
↳ Involves maintaining correct and efficient operations inspite of any failures of the communication channels (or so called as links), nodes and certain processes.

→ Security:
↳ various aspects of cryptography, secure channels, and access control, key management

→ API and Transparency:
↳ Transparency deals with hiding the implementation policies from the user.
- Access transparency: hides differences in data representation on different systems.
- location transparency: makes locations of resources transparent to the user.
- migration transparency: allows re-locating resources without changing names.

- Relocation transparency: ability to re-locate resources as they are being accessed.
- Replication transparency: does not let the user become aware of any replication.
- Concurrency transparency: deals with masking the concurrent use of shared resources for the user.
- Failure transparency: refers to the system being reliable and fault tolerant.

→ Scalability and Modularity

Algorithmic challenges in Distributed computing

→ Designing useful Execution models and frameworks
  ↳ Inter-leaving model.
  ↳ Partial order model
  ↳ Input-output Automata model
  ↳ the TLA (Temporal logical Action)

→ Dynamic distributed algorithms and distributed routing algorithms.

→ Time and global state in a distributed system.
  ↳ Physical time and logical time.

* Each processor has its own clock, which might not be in sync with clocks of all other processors.

→ Synchronization / coordination mechanisms:
↳ Resource management and concurrency management
- Physical clock synchronization
- Leader election : to coordinate activities of various processes, and resource management
- Mutual exclusion : resources are shared
- Deadlock detection and resolution.
- Termination detection.
- Garbage collection

→ Group communication, multicast and ordered message delivery.

→ Monitoring distributed events and predicates

→ Distributed program design and verification tools.

→ Debugging distributed programs.

→ Data replication, consistency model, & caching

→ World-wide web - design : caching, searching, scheduling.

→ Distributed shared memory abstraction.

↳ Wait-free Algorithm
- Even in concurrency case, the program runs.

↳ Mutual exclusion

↳ Register construction.

↳ consistency models.

→ Reliable and fault tolerant distributed systems:

↳ Concencus algorithms
↳ Replication and replica management (Triple Modular Redundancy)
↳ voting and Quorum systems.
↳ Distributed databases and distributed commit
↳ self-stabilizing system.
↳ Checkpointing & recovery algorithms.
↳ failure detectors.

Algorithmic Challenges

→ Load balancing to gain throughput and reduce user percieved latency.

↳ Data Migration
↳ computation migration
↳ Distributed scheduling.

→ Real time scheduling (Mission critical applications)

→ Performance metrics

↳ Measurement methods / tools

Applications of Distributed computing and Newer Challenges

→ Mobile systems → cellular (base station)
→ ad-hoc

↳ Issues:

- Range of translation; power of transmission
- Battery power conservation: interfacing with wired

- Signal processing and indifference : Routing
- Location management : channel allocation
- Localization : position estimation : overall management of mobility

→ Sensor Networks
  ↳ Mobiles in static
  ↳ Sensors have to self configure to form ad-hoc networks.

→ Ubiquotous or pervasive computing

→ Peer to peer computing.

→ Publish - subscribe content distribution and multimedia

→ Distributive data mining.

→ Grid computing

→ Security in distributed systems.

A Model of Distributed Computation

→ A distributed system consists of a set of processors connected by a communication network.

→ Communication network provides facility of information exchange among processors.

→ communication delay is finite, but unpredictable.

→ Communication medium may deliver messages out of order, and messages may be lost, garbled or duplicated due to timeout and re-transmission.

→ Processors may fail & communication links may go down

→ System can be modeled as a directed graph where vertices correspond to the processors (i.e. vertices represent the processes), and the edges represent the uni-directional communication channel.

→ A distributed application runs as a collection of processes on a distributed system.

→ A distributed program is a collection of finite set of processes.

A distributed program is composed of a set of n asynchronous process, $P_1, P_2, \ldots P_n$, and these processes communicate by message passing or a communication media.

C_{ij} : channel from process P_i to process P_j.

m_{ij} : message sent by process P_i to process P_j.

→ process execution and message transfer are asynchronous.

→ Global state of a distributed computation is composed of the states of processes and communication channels.

$\underline{\$}$ State of communication channel is dependent on the no. of messages in transit at that given point of time.

A Model of Distributed Execution

→ The execution of a process consists of a sequential execution of its actions.

→ The actions are atomic.

There are 3 types of events leading to these actions:

i) Internal event : only changes state of process.

ii) message send event : change in state of channel & sending process

iii) Message receive event : change in state of channel & receiving process.

e_i^x : xth event at process P_i

→ Occurrence of events change the state of the processes, and channels. And this leads to a transition in the global state.

→ Internal event will change the state of the process.

Send event will change the state of the sending process & also the channel.

Receive event will change the state of the receiving process & also the channel.

→ The events at a process are linearly ordered by their order of occurrence.

The execution of process P_i produces a sequence of events, $e_i^1, e_i^2 \dots e_i^x, e_i^{x+1} : H_i$

This sequence of events is denoted by H_i.

H_i : set of events with ordering relation

$H_i = (h_i, \rightarrow_i)$ h_i is the set of events

$\rightarrow_i$ defines a linear order on the events

→ The send and the receive events signify the flow of information between processes.

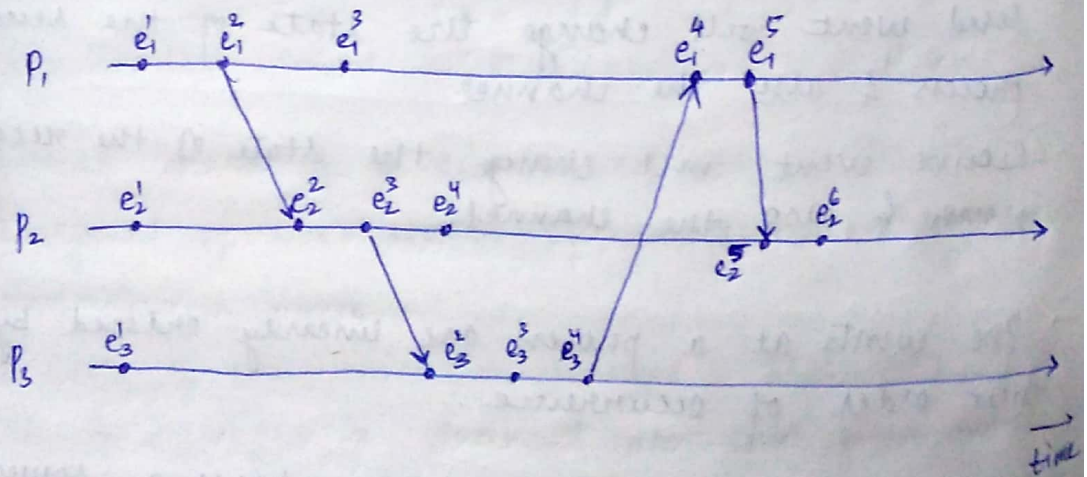
→ There is a causal dependency from the sender process to the receiver process.

$\longrightarrow_{msg}$: This relation reflects/captures the causal dependency due to message exchange.

$send(m) \longrightarrow_{msg} rec(m)$

So $\longrightarrow_{msg}$ relation defines the causal dependencies between the pairs of corresponding send and receive events.

→ The evolution of distributed execution is depicted by the space-time diagram.



Space-time diagram of distributed execution.

Horizontal line represents the progress of the process. A dot on the line indicates an event, and a slant arrow indicates a message construct/transfer.

→ Causal Precedence Relation :

↳ The execution of a distributed application results in a set of distributed events

$$H = \bigcup_i h_i$$

↳ A binary relation on the set H, denoted as '→', expresses the causal dependencies between events.

$\forall e_i^x \quad \forall e_j^y \in H$, then we will say,

$$e_i^x \rightarrow e_j^y \iff \begin{cases} e_i^x \rightarrow_i e_j^y \text{ , i.e., } (i=j) \wedge (x<y) \\ e_i^x \rightarrow_{msg} e_j^y \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{cases}$$

↳ Causal precedence relation induces an irreflexive partial order on the events of the distributed computation.

↳ $e_i \rightarrow e_j$: Implies that all the information available at e_i is potentially accessible at e_j.

↳ For events e_i & e_j, if it is given as,

$e_i \not\rightarrow e_j$: shows that e_j is not dependent on e_i.

This relation is called as: Lamport's 'happen before'

Eg: In the adjacent space-time diagram,

$e_1^1 \rightarrow e_3^3$; $e_3^3 \rightarrow e_2^6$

↳ For any two events, e_i & e_j, i) If $e_i \not\to e_j \not\Rightarrow e_j \not\to e_i$

ii) If $e_i \to e_j \Rightarrow e_j \not\to e_i$

iii) If $e_i \not\to e_j$ & $e_j \not\to e_i$: These 2 events e_i & e_j are said to be concurrent, and they are denoted by the relation : $e_i \| e_j$.

- 2 types of concurrencies:
  - physical concurrency : 2 events take place at the same time
  - logical concurrency : 2 events may take place at the same time.

Eg: $e_1^3 \| e_3^3$; $e_2^4 \| e_3^1$

- The concurrency relation is not transitive, i.e.,

$(e_i \| e_j) \wedge (e_j \| e_k) \not\Rightarrow e_i \| e_k$

Eg: $e_3^3 \| e_2^4$ and $e_2^4 \| e_1^5$, but, $e_3^3 \not\| e_1^5$

* ± logical concurrency is the most general type of concurrency present.

Eg: $\{e_1^3, e_2^4, e_3^3\}$: These 3 events are concurrent

~~→ 3 types of communication links.~~

Models of Communication Networks

i) FIFO
ii) non - FIFO
iii) Causal order (CO)

→ In FIFO, the first message entering the network, it is the first one received by the receiver.

→ In non - FIFO, messages are put in a set, and the receiver may get/select any message from the set, so it is not for sure that the message that enters the communication network will be received first by the receiver.

→ For causal order,
m_{ij} and m_{kj}
If send (m_{ij}) → send (m_{kj}), then rec (m_{ij}) → rec (m_{kj})

* CO ⊂ FIFO ⊂ non - FIFO

Global State of a Distributed System

→ collection of local states of the processes and communication channels.

↳ State of a process is given by the contents of processor registers, stacks, local memory and also depends on the local context of the distributed application.

↳ State of a channel is given by the set of messages in transit in the channel.

→ Internal event changes the state of the process at which it occurs.

A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received).

LS_i^x : local state of process P_i after the occurrence of event e_i^x, and before the next event e_i^{x+1}.

LS_i^0 : Initial state of the process P_i.

* LS_i^x is the effect/result/consequence of execution of all the events executed by process P_i till the event e_i^x.

→ Let $send(m) \leq LS_i^x$: This denotes a fact that,

$\Rightarrow \exists\, y : 1 \leq y \leq x :: e_i^y = send(m)$

Let $rec(m) \not\leq LS_i^x$

$\Rightarrow \forall\, y : 1 \leq y \leq x :: e_i^y \neq rec(m)$

→ Channel state, $SC_{ij}^{x,y} = \{ m_{ij} \mid send(m_{ij}) \leq LS_i^x$ AND $rec(m_{ij}) \not\leq LS_j^y \}$

Channel state of the channel between processes P_i & P_j.

x : till event e_i^x, P_i has sent messages

y : till event e_j^y, P_j has not received any messages.

Global State = $\{ \bigcup_i LS_i^{x_i} , \bigcup_{j,k} SC_{j,k}^{y_j z_k} \}$

→ Global Snapshot

* It is not possible to have the global state of a distributed system at a particular point of time because of no global clock / asynchronized events.

→

A message cannot be received if it was not sent.

↳ A system is consistent if ~~[illegible]~~ message received was actually sent earlier.

↳ This above property is the 'Causality Property'.

↳ A distributed state which satisfies the causality property is a consistent global state.

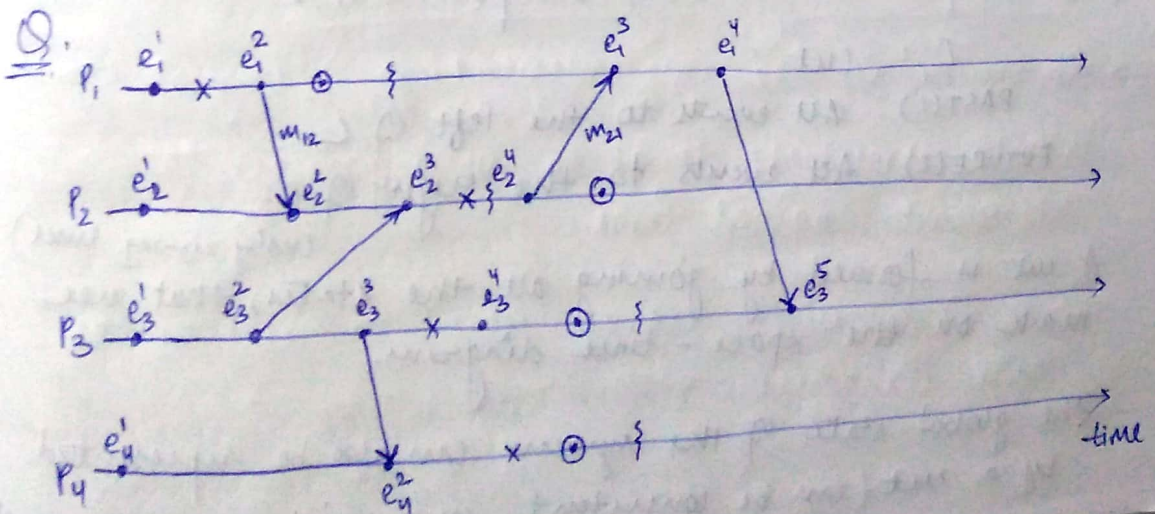
↳ A global state, $GS = \{ \bigcup_i LS_i^{x_i} , \bigcup_{j,k} SC_{j,k}^{y_j z_k} \}$ is a consistent state iff it satisfies the following condition:

$$\forall m_{ij} : send(m_{ij}) \notin LS_i^{x_i} \Rightarrow m_{ij} \notin SC_{ij}^{x_i y_j} \wedge rec(m_{ij}) \notin LS_j^{y_j}$$

* A global state cannot be inconsistent.

* Strongly consistent : NO messages in transit (all channels are empty)

Q.



i) $GS_1 = \{LS_1^1, LS_2^2, LS_3^3, LS_4^2\}$: States marked with '.'

↳ Inconsistent state due to violation of Causality

ii) $GS_2 = \{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$: States marked with '⊙'.

↳ System is consistent; causality is not violated by any of the states.

iii) $GS_3 = \{LS_1^2, LS_2^3, LS_3^4, LS_4^2\}$: States marked with '}'

↳ System is strongly consistent as there is no message in transit.

→ A global state $GS = \{\cup_i LS_i^{x_i}, \cup_{jk} SC_{jk}^{y_j z_k}\}$ is transit less iff,

$$1 \le i,j \le n :: SC_{ij}^{y_i z_j} = \phi$$

*. Strongly consistent global state is a state which is both consistent as well as transit less.

⇒ Cuts of a Distributed Computation:

C : Cut

PAST(C) : All events to the left of C

FUTURE(C) : All events to the right of C.

A cut is formed by joining all the states, that we mark on the space-time diagram. (using zig-zag lines)

The global state of the system can also be represented by a cut (can be consistent, inconsistent or strongly consistent)

Logical Time

→ Concept of causality between events: Design and analysis of parallel and distributed computing and OSs.

→ causality is tracked using physical time, whereas an asynchronous distributed computation makes progress in spurts.

So, it turns out that logical time, which advances in jumps is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.

→ Three ways to implement logical time:

- ↳ Scalar Time (or Lamport's Time)
- ↳ Vector Time
- ↳ Matrix Time

→ Causality is a very important concept in reasoning, analyzing and drawing inferences about a computation.

↓ Effects of causality.

→ 1. Distributed Algorithm Design

↳ Helps us to find the liveness and fairness in mutual exclusion algorithms.

↳ It helps maintain consistency in replicated databases.

↳ Deadlock detection algorithms

→ 2. Tracking of Dependent Events

→ 3. Knowledge about the progress as the events occur.

→ 4. Concurrency measure

A Framework for a System of Logical Clocks

→ consists of a Time domain T, and a logical clock C

↳ Elements of T form a partially ordered set over a relation $<$. (Because T is not continuous, it is broken into discrete jumps).

↳ $<$: Happened before or Causal precedence relation.

$$C : H \longrightarrow T$$

For 2 events, e_i & e_j,

$$e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j) \quad : \text{consistent}$$

e_i precedes e_j

↳ Time stamp of e_i is less than that of e_j

For two events, e_i and e_j,

if $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$: strongly consistent

Implementing Logical Clocks

→ 2 issues in the implementation of logical clocks:

i) Data structure local to every process to represent logical time.

ii) A protocol (a set of rules) to update the data structure.

→ Each process P_i maintains a local logical clock, lc_i, that helps process P_i measure its own progress.

↳ a logical global clock, denoted by gc_i. It is a representation of P_i's local view of the logical global time.

→ The protocol:

↳ Rule 1: governs how the local logical clock is updated by a process when it executes an event.

↳ Rule 2: governs how a process updates its global logical clock.
: it dictates what information about the logical time is piggybacked in a message and how this ~~mess~~ information is used by the receiving process to update its view of the global time.

Scalar Time

→ Proposed by Leslie Lamport

→ Time domain ~~Rese~~ in this representation is the set of non-negative integers.

↳ lc_i and gc_i of process P_i are squashed into one integer variable.

→ R1: Before executing a send or internal event, process P_i executes the following:

$$c_i = c_i + d \quad , \quad (d > 0)$$

*. Although value of d is not fixed, but generally we take that value as 1. (may change based on the application)

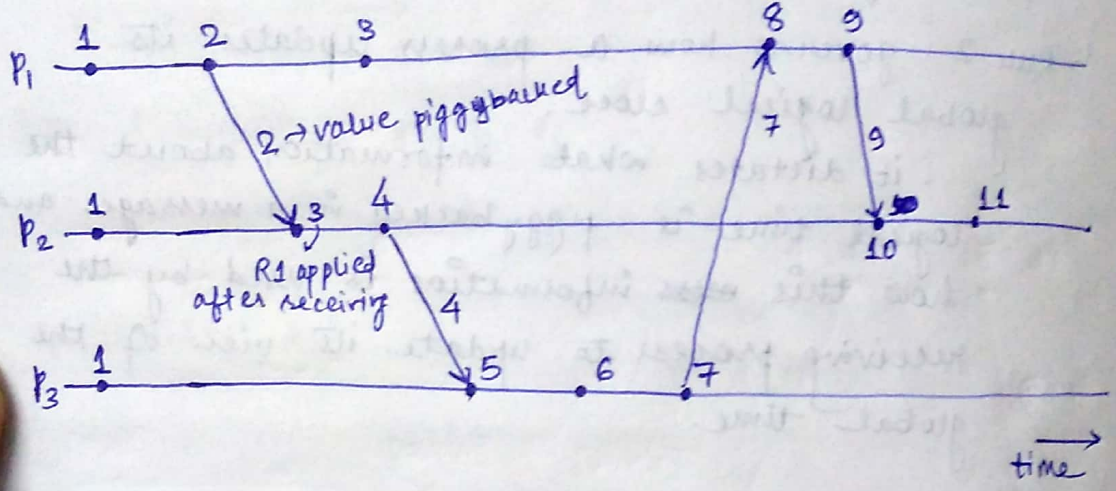
→ R2: Each message piggybacks the clock value of its sender at sending time.

When process P_i receives a message with time stamp C_{msg}, it executes the following actions:

i) $C_i := \max(C_i, C_{msg})$

ii) Execute R1.

iii) deliver the message.



Evolution of scalar Time

→ Basic Properties of Scalar Time:

↳ Consistency property is satisfied.

- For 2 events e_i & e_j,

  if $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$

↳ Total ordering.

- In case there are multiple events, let say 2 events x & y with time stamps (h, i) & (k, j), then $x < y \Rightarrow (h < k$ or $(h = k$ and $i < j))$

(h, i): $C = h$ & $P = P_i$

- smaller process no. will be given preference if the time instant of occuring is same for both the process.

↳ Event counting.

- event e has time stamp h, which shows that (h-1) events have occured before e.
- h : height of event e.

↳ NO strong consistency

- e_i & e_j : $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$.

Vector Time

→ The Time domain is represented by a set of n-dimensional non-negative integer vectors.

↳ P_i $vt_i[1...n]$: Each process maintains a vector.

↳ $vt_i[i]$: indicates the ith component of the vector time of P_i (ith component denotes the local time of process P_i).

↳ $vt_i[j]$: indicates the jth component of the vector time of process P_i.

→ R1 : before a send event, or an internal event,

$vt_i[i] = vt_i[i] + d$; $(d > 0)$.

→ R2: Each message is piggybacked with the vector clock of the sender process at sending time.

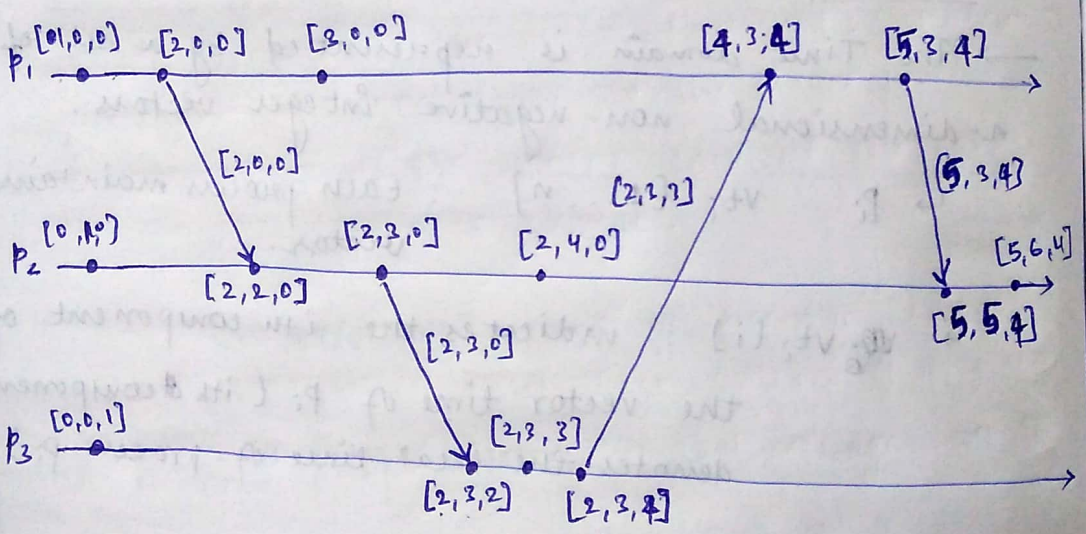
On the receipt of such a message (m, vt), p_i executes the following:

i) Update its global time (all components)

$\forall\ k \in [1, n]$, $vt_i[k] = \max(vt_i[k], vt[k])$

ii) Execute R1.

iii) Deliver the message.



→ if $vh = vk \iff \forall x\ ;\ vh[x] = vk[x]$

$vh \leq vk \iff \forall x : vh[x] \leq vk[x]$

$vh < vk \iff vh \leq vk$ and $\exists x\ ;\ vh[x] < vk[x]$

concurrent with ← $vh \parallel vk \iff \neg(vh < vk) \wedge \neg(vk < vh)$

↳ i.e., they are not related.

→ Basic Properties of Vector Time:

↳ Isomorphism:

- The relation "→" induces a partial order.
- If 2 events, x and y have timestamps vh and vk respectively, then.

$x \rightarrow y \iff vh < vk$

$x \parallel y \iff vh \parallel vk$ (the 2 vectors cannot be compared)

- If events x & y respectively occured at process P_i and P_j are assigned timestamps vh and vk respectively, then,

$x \rightarrow y \iff vh[i] \leq vk[i]$

$x \parallel y \iff vh[i] > vk[i] \wedge vh[j] < vk[j]$

↳ Strong consistency.

↳ Event counting.

- If an event e has a timestamp vh, then, vh[j] indicates/denotes the number of events executed by process P_j that causally precede e.

$\sum vh[j] - 1$: total number of events that causally precede e in the distributed computation.

→ Applications

- Distributed debugging.
- Implementation of causal order communicat^n
- Causal distributed shared memory (address space is same, but different modules of memory for different process)

- Establishment of global checkpoints.
- Consistency of checkpoints in optimistic recovery.

** When a process sends message to another process, usually only a few components change from the previous message sent.

Efficient Implementation of Vector Clocks.

→ The message overhead grows linearly with the number of processors in the system.

⟹ Singhal - Kshemkalyani Differential Technique

↳ It is based on the observation that: Between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change.

↳ When a process P_i sends a message to process P_j, it piggybacks only those entries of its vector clock that differ since the last message sent to P_j.

If entries $i_1, i_2 \ldots i_n$ of the vector clock at P_i have changed to $v_1, v_2 \ldots v_n$ respectively since the last message sent to P_j, then process P_i piggybacks a compressed timestamp of the form:

$$\{(i_1, v_1), (i_2, v_2) \ldots (i_n, v_n)\}$$

to the next message to P_j.

When P_j receives this message, it updates its

vector as follows:

$$vt_j[i_k] = \max(vt_j[k], vk) \quad \text{for } k = 1, 2 \ldots n$$

↳ This technique will reduce the message size, communication bandwidth and buffer requirements.

↳ Process P_i maintains the following 2 additional vectors:

- $LS_i[1 \ldots n]$ (last sent)

  $LS_i[j]$: Indicates the value of $vt[i]$ when process P_i last sent a message to process P_j.

- $LU_i[j]$ (last update)

  $LU_i[j]$: Indicates the value of $vt[i]$ when process P_i last updated the entry $vt_i[j]$.

* Both the above timestamps are the local time of the process P_i.

** $LU_i[i] = vt_i[i]$ (Always true)

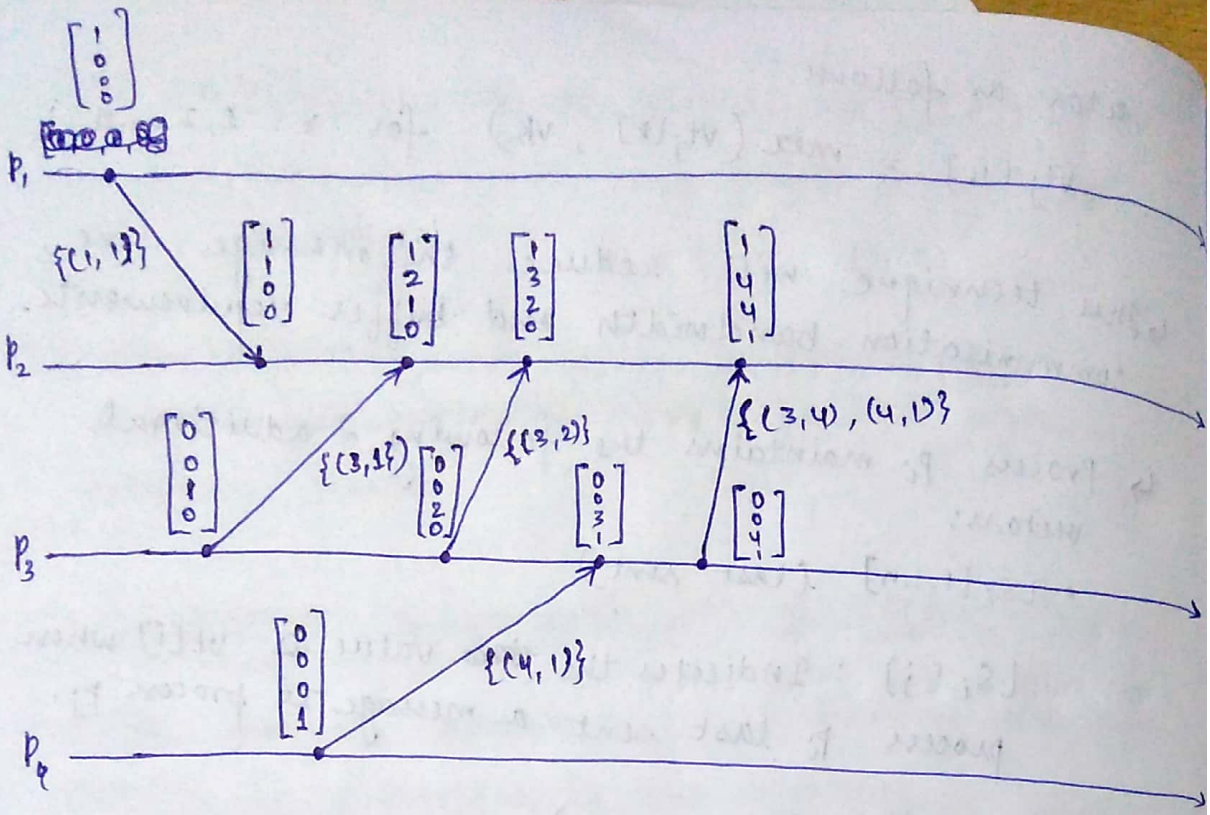
- $LU_i[j]$ needs to be updated only when the receipt of the message causes P_i to update entry $vt_i[j]$.

- $LS_i[j]$ needs to be updated only when P_i sends a message to P_j.

When P_i sends a message to P_j, it sends only a set of tuples which got modified after the last message has been sent.

$$\{(x, vt_i[x]) \mid LS_i[j] < LU_i[x]\}$$

as a vector timestamp to P_j instead of sending a vector of n entries in the message.



vector clock progress in singhal-kshemkalyani technique.

Synchronizing Physical Clocks

(open book in book list, book by surepal on synchronization)

→ 2 problems:

↳ clock skew

↳ clock drift

→ Coordinated Universal Time (UTC)

→ Land-based stations. (0.1 – 10 milisec accuracy)

→ Satellite (0.1 μsec accuracy)

→ External synchronization : Internal synchronization.

↳ External synchronization: For a synchronization $D > 0$ and for a source S of UTC time,

$$|S(t) - C_i(t)| < D \quad \text{for } i = 1, 2 \ldots N \text{ and}$$

for all times t in I (interval of real time)

clocks C_i are accurate to within the bound D.

↳ Internal synchronization: For a synchronization bound $D > 0$,

$$|C_i(t) - C_j(t)| < D \quad \text{for } i, j = 1, 2 \ldots N,$$

and for all real time t in I (interval of real time)

→ Cristian's Method

↳ Time Server

↳ Source : UTC

→ The Berkley Algorithm (for Internal)

↳ A coordinated computer is chosen to act as a master.

The Network Time Protocol (NTP).

→ Purpose of this is to distribute time information over the Internet.

→ Objective : i) To provide a service enabling clients across the Internet to be synchronized accurately to the UTC.

ii) To provide a reliable service that can survive lengthy losses of connectivity.

iii) To enable clients to re-synchronize sufficiently frequently to offset the rates of drift found in most computers.

iv) To provide protection against interference with time service, whether malicious or accidental.

→ NTP service is provided by a network of servers.

↳ Primary servers

↳ Secondary servers.

This network of servers is known as the synchronization subnet.

↳ Strata levels.

→ NTP servers synchronize with one another in one of three modes:

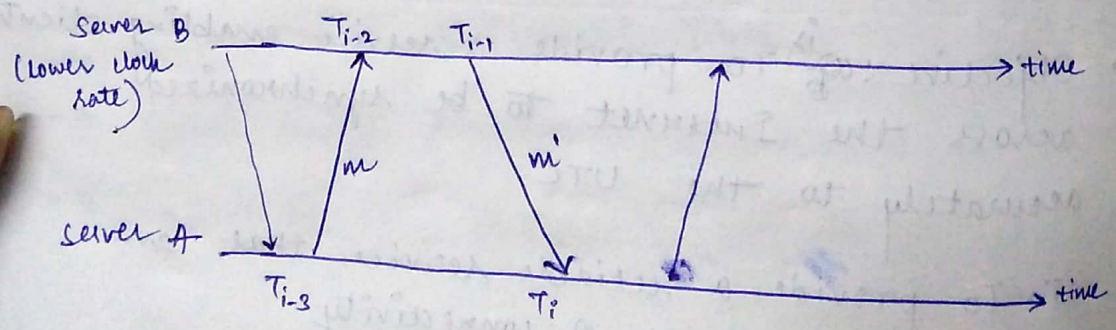
i) Multicast (one source & a subset of receivers)

ii) Procedure calls (between 2 processes) → At different levels. (High to low)

iii) Symmetric mode.

In all modes, messages are delivered unreliably using standard UDP. Internet transport protocol.

→ Message exchange between a pair of NTP peers:



$$T_{i-2} = T_{i-3} + t + o$$

(o: offset & variable; it not zero here)

$$T_i = T_{i-1} + t' - o$$

(This topic in book by the 4 authors, IV & V ed.)

or NTP by MD Mills

Total delay, $d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$

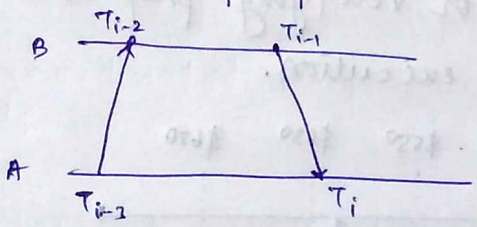
Offset, $O = O_i + \frac{(t'-t)}{2}$, where, $O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$

Using the fact that $t, t' \geq 0$, it can be shown that,

$$O_i - \frac{d_i}{2} \leq O \leq O_i + \frac{d_i}{2}$$

O_i : an estimate of the offset
d_i : a measure of accuracy of this estimate.

**. David L. Mills : "Internet Time Synchronization"
↓
Subject : Messaging delay and offset of paper.



$a = T_{i-2} - T_{i-3}$

$b = T_{i-1} - T_i$

Round trip delay, δ_i ; clock offset θ_i of B relative to A.

$\delta_i = a - b$

$\theta_i = \frac{(a+b)}{2}$

$T_{i-2} = T_{i-3} + t + \theta$
$T_i = T_{i-1} + t - \theta$
→ Assuming same message is sent

Actual Time $x + \theta = T_{i-2} - T_{i-3} = a$

$x = a - \theta \geq 0$

hence, $a \geq \theta$

$\theta \geq b$ → Basis is that if B sends a message, then actual time will be

$x - \theta = -b \quad (\because b = T_{i-1} - T_i)$

$\Rightarrow \theta = x + b$

$b \le \theta \le a$.

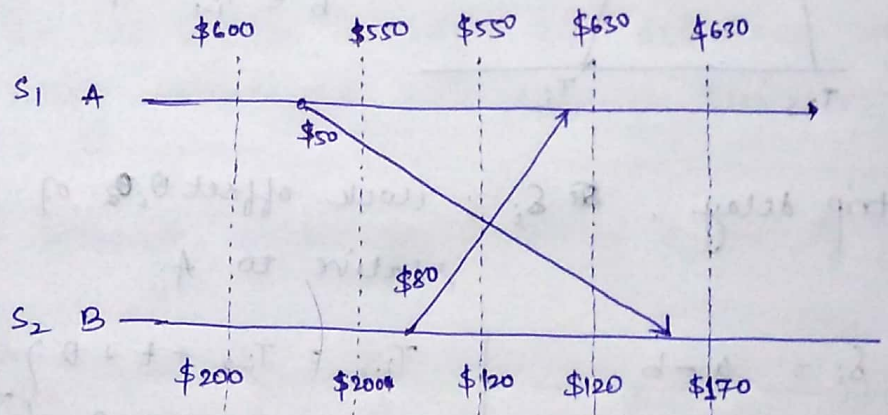
$$b = \frac{a+b}{2} - \frac{a-b}{2} \le \theta \le \frac{a+b}{2} + \frac{a-b}{2} = a$$

$$\theta_i - \frac{d_i}{2} \le \theta \le \theta_i + \frac{d_i}{2}$$

↳ Offset can be $-\frac{d_i}{2}$ or $\frac{d_i}{2}$.

Global State and Snapshot Recording Algorithm.

→ Recording global state is necessary for analyzing, testing or verifying properties associated with distributed execution.



C_{12}	0	$50	$50	$50	0
C_{21}	0	0	$80	0	0