

CS3551- DISTRIBUTED COMPUTING

UNIT I INTRODUCTION

Introduction: Definition-Relation to Computer System Components – Motivation – Message -Passing Systems versus Shared Memory Systems – Primitives for Distributed Communication – Synchronous versus Asynchronous Executions – Design Issues and Challenges; A Model of Distributed Computations: A Distributed Program – A Model of Distributed Executions – Models of Communication Networks – Global State of a Distributed System.

UNIT II LOGICAL TIME AND GLOBAL STATE

Logical Time: Physical Clock Synchronization: NTP – A Framework for a System of Logical Clocks – Scalar Time – Vector Time; Message Ordering and Group Communication: Message Ordering Paradigms – Asynchronous Execution with Synchronous Communication – Synchronous Program Order on Asynchronous System – Group Communication – Causal Order – Total Order; Global State and Snapshot Recording Algorithms: Introduction – System Model and Definitions – Snapshot Algorithms for FIFO Channels

UNIT III DISTRIBUTED MUTEX AND DEADLOCK

Distributed Mutual exclusion Algorithms: Introduction – Preliminaries – Lamport's algorithm – RicartAgrawala's Algorithm — Token-Based Algorithms – Suzuki-Kasami's Broadcast Algorithm; Deadlock Detection in Distributed Systems: Introduction – System Model – Preliminaries – Models of Deadlocks – Chandy-Misra-Haas Algorithm for the AND model and OR Model.

UNIT IV CONSENSUS AND RECOVERY

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure-Free System(Synchronous and Asynchronous) – Agreement in Synchronous Systems with Failures; Checkpointing and Rollback Recovery: Introduction – Background and Definitions – Issues in Failure Recovery – Checkpoint-based Recovery – Coordinated Checkpointing Algorithm – Algorithm for Asynchronous Checkpointing and Recovery

UNIT V CLOUD COMPUTING

Definition of Cloud Computing – Characteristics of Cloud – Cloud Deployment Models – Cloud Service Models – Driving Factors and Challenges of Cloud – Virtualization – Load Balancing – Scalability and Elasticity – Replication – Monitoring – Cloud Services and Platforms: Compute Services – Storage Services – Application Services

UNIT I INTRODUCTION

The process of computation was started from working on a single processor. This uni-processor computing can be termed as **centralized computing**.

A distributed system is a collection of independent computers, interconnected via a network, capable of collaborating on a task. Distributed computing is computing performed in a distributed system.

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. Distributed computing is widely used due to advancements in machines; faster and cheaper networks. In distributed systems, the entire network will be viewed as a computer. The multiple systems connected to the network will appear as a single system to the user.

Features of Distributed Systems:

No common physical clock - It introduces the element of “distribution” in the system and gives rise to the inherent asynchrony amongst the processors.

No shared memory - A key feature that requires message-passing for communication. This feature implies the absence of the common physical clock.

Geographical separation – The geographically wider apart that the processors are, the more representative is the system of a distributed system.

Autonomy and heterogeneity – Here the processors are “loosely coupled” in that they have different speeds and each can be running a different operating system.

Issues in distributed systems

- Heterogeneity
- Openness
- Security
- Scalability
- Failure handling
- Concurrency
- Transparency
- Quality of service

1.2 Relation to Computer System Components

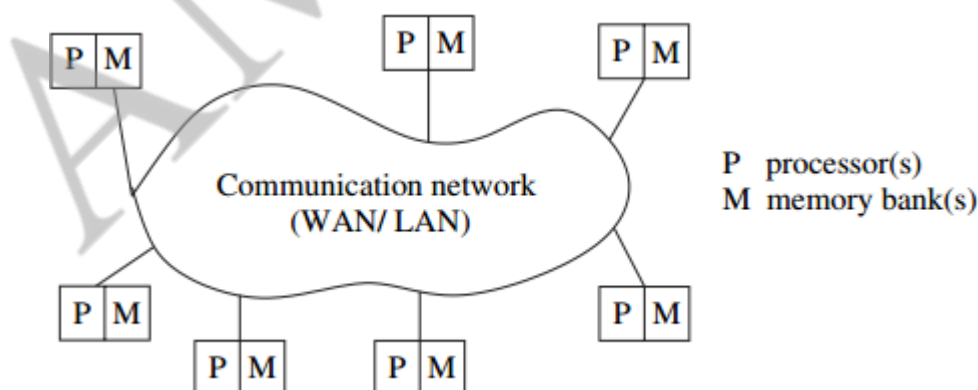


Fig 1.1: Example of a Distributed System

As shown in Fig 1.1, Each computer has a memory-processing unit and the computers are connected by a communication network. Each system connected to the distributed networks hosts distributed software which is a middleware technology. This drives the Distributed System (DS) at the same time preserves the heterogeneity of the DS. The term **computation or run** in a distributed system is the execution of processes to achieve a common goal.

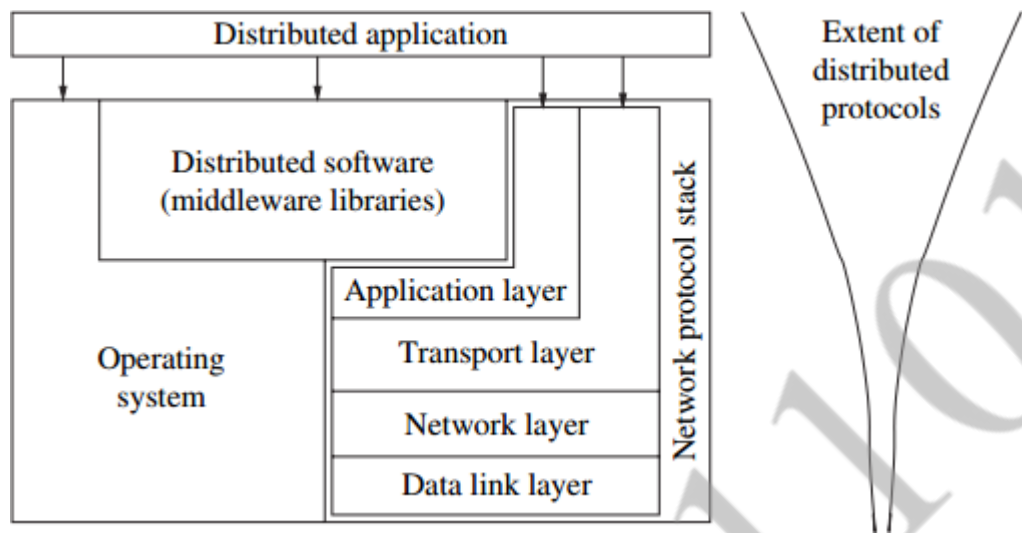


Fig 1.2: Interaction of layers of network

The interaction of the layers of the network with the operating system and middleware is shown in Fig 1.2. The middleware contains important library functions for facilitating the operations of DS.

The distributed system uses a layered architecture to break down the complexity of system design. The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level

Examples of middleware: Object Management Group's (OMG), Common Object Request Broker Architecture (CORBA) [36], Remote Procedure Call (RPC), Message Passing Interface (MPI)

1.3 Motivation

The following are the key points that acts as a driving force behind DS:

Inherently distributed computations: DS can process the computations at geographically remote locations.

Resource sharing: The hardware, databases, special libraries can be shared between systems without owning a dedicated copy or a replica. This is cost effective and reliable.

Access to geographically remote data and resources: Resources such as centralized servers can also be accessed from distant locations.

Enhanced reliability: DS provides enhanced reliability, since they run on multiple copies of resources.

The term reliability comprises of:

1. **Availability:** The resource/ service provided by the resource should be accessible at all times
2. **Integrity:** the value/state of the resource should be correct and consistent.

3. **Fault-Tolerance:** Ability to recover from system failures

Increased performance/cost ratio: The resource sharing and remote access features of DS naturally increase the performance / cost ratio.

Scalable: The number of systems operating in a distributed environment can be increased as the demand increases.

1.4 MESSAGE-PASSING SYSTEMS VERSUS SHARED MEMORY SYSTEMS

Communication among processors takes place via shared data variables, and control variables for synchronization among the processors. The communications between the tasks in multiprocessor systems take place through two main modes:

Message passing systems:

- This allows multiple processes to read and write data to the message queue without being connected to each other.
- Messages are stored on the queue until their recipient retrieves them.

Shared memory systems:

- The shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other.
- Communication among processors takes place through shared data variables, and control variables for synchronization among the processors.
- Semaphores and monitors are common synchronization mechanisms on shared memory systems.
- When shared memory model is implemented in a distributed environment, it is termed as **distributed shared memory**.

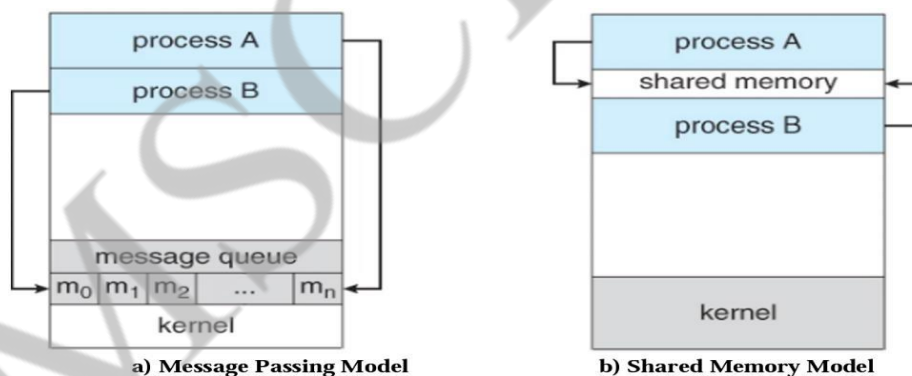


Fig 1.11: Inter-process communication models

Emulating message-passing on a shared memory system (MP \rightarrow SM)

- The shared memory system can be made to act as message passing system. The shared address space can be partitioned into disjoint parts, one part being assigned to each processor.
- Send and receive operations are implemented by writing to and reading from the destination/sender processor's address space. The read and write operations are synchronized.
- Specifically, a separate location can be reserved as the mailbox for each ordered pair of processes.

Emulating shared memory on a message-passing system (SM \rightarrow MP)

- This is also implemented through read and write operations. Each shared location can be modeled as a separate process. Write to a shared location is

emulated by sending an update message to the corresponding owner process and read operation to a shared location is emulated by sending a query message to the owner process.

- This emulation is expensive as the processes has to gain access to other process memory location. The latencies involved in read and write operations may be high even when using shared memory emulation because the read and write operations are implemented by using network-wide communication.

1.5 PRIMITIVES FOR DISTRIBUTED COMMUNICATION

Blocking / Non blocking / Synchronous / Asynchronous

- Message send and message receive communication primitives are done through Send() and Receive(), respectively.
- A Send primitive has two parameters: *the destination*, and *the buffer* in the user space that holds the data to be sent.
- The Receive primitive also has two parameters: *the source* from which the data is to be received and the *user buffer* into which the data is to be received.

There are two ways of sending data when the Send primitive is called:

- **Buffered:** The standard option copies the data from the user buffer to the kernel buffer. The data later gets copied from the kernel buffer onto the network. For the Receive primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.
- **Unbuffered:** The data gets copied directly from the user buffer onto the network.

Blocking primitives

- The primitive commands wait for the message to be delivered. The execution of the processes is blocked.
- The sending process must wait after a send until an acknowledgement is made by the receiver.
- The receiving process must wait for the expected message from the sending process
- A primitive is blocking if control returns to the invoking process after the processing for the primitive completes.

Non Blocking primitives

- If send is nonblocking, it returns control to the caller immediately, before the message is sent.
- The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle.
- This is a form of asynchronous communication.
- A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed.
- For a non-blocking Send, control returns to the process even before the data is copied out of the user buffer.

For a non-blocking Receive, control returns to the process even before the data may have arrived from the sender.

Synchronous

- A Send or a Receive primitive is synchronous if both the Send() and Receive() handshake with each other.
- The processing for the Send primitive completes only after the invoking processor learns

- The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.

Asynchronous

- A Send primitive is said to be asynchronous, if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer.
- For non-blocking primitives, a return parameter on the primitive call returns a system-generated **handle** which can be later used to check the status of completion of the call.
- The process can check for the completion:
 - checking if the handle has been flagged or posted
 - issue a Wait with a list of handles as parameters: usually blocks until one of the parameter handles is posted.

The send and receive primitives can be implemented in four modes:

- Blocking synchronous
- Non- blocking synchronous
- Blocking asynchronous
- Non- blocking asynchronous

Four modes of send operation

Blocking synchronous Send:

- The data gets copied from the user buffer to the kernel buffer and is then sent over the network.
- After the data is copied to the receiver's system buffer and a Receive call has been issued, an acknowledgement back to the sender causes control to return to the process that invoked the Send operation and completes the Send.

Non-blocking synchronous Send:

- Control returns back to the invoking process as soon as the copy of data from the user buffer to the kernel buffer is initiated.
- A parameter in the non-blocking call also gets set with the handle of a location that the user process can later check for the completion of the synchronous send operation.
- The location gets posted after an acknowledgement returns from the receiver.
- The user process can keep checking for the completion of the non-blocking synchronous Send by testing the returned handle, or it can invoke the blocking Wait operation on the returned handle

Blocking asynchronous Send:

- The user process that invokes the Send is blocked until the data is copied from the user's buffer to the kernel buffer.

Non-blocking asynchronous Send:

- The user process that invokes the Send is blocked until the transfer of the data from the user's buffer to the kernel buffer is initiated.
- Control returns to the user process as soon as this transfer is initiated, and a parameter in the non-blocking call also gets set with the handle of a location that the user process can check later using the Wait operation for the completion of the asynchronous Send.

The asynchronous Send completes when the data has been copied out of the user's buffer. The checking for the completion may be necessary if the user wants to reuse the buffer from which the data was sent.

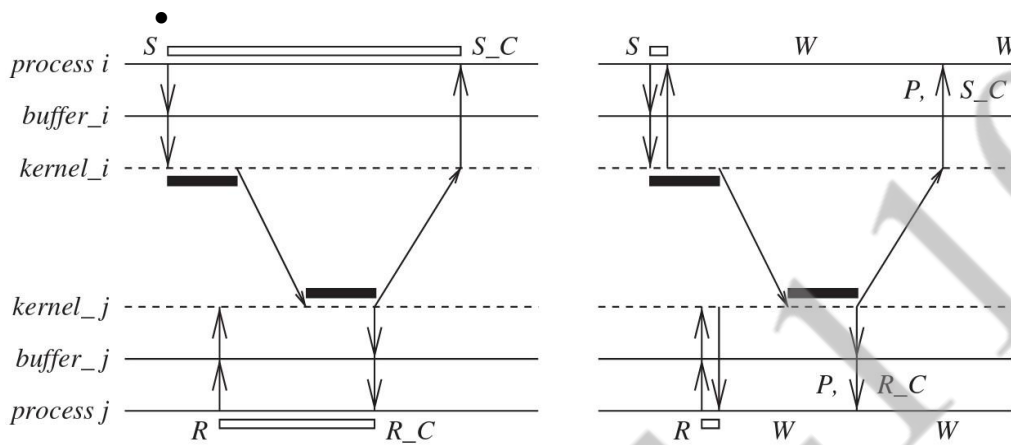
Modes of receive operation

Blocking Receive:

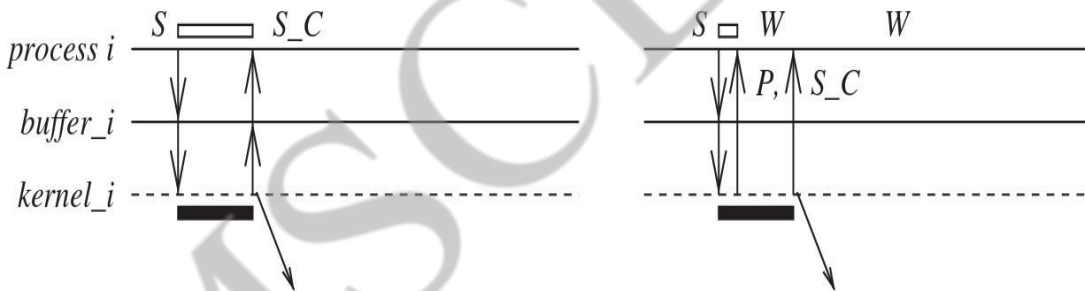
The Receive call blocks until the data expected arrives and is written in the specified user buffer. Then control is returned to the user process.

Non-blocking Receive:

- The Receive call will cause the kernel to register the call and return the handle of a location that the user process can later check for the completion of the non-blocking Receive operation.
- This location gets posted by the kernel after the expected data arrives and is copied to the user-specified buffer. The user process can check for then completion of the non-blocking Receive by invoking the Wait operation on the returned handle.



(a) Blocking sync. *Send*, blocking *Receive* (b) Nonblocking sync. *Send*, nonblocking *Receive*



(c) Blocking async. *Send* (d) Non-blocking async. *Send*

█ Duration to copy data from or to user buffer

▭ Duration in which the process issuing send or receive primitive is blocked

S *Send* primitive issued S_C processing for *Send* completes

R *Receive* primitive issued R_C processing for *Receive* completes

P The completion of the previously initiated nonblocking operation

W Process may issue *Wait* to check completion of nonblocking operation

Processor Synchrony

Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.

To ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

Libraries and standards

There exists a wide range of primitives for message-passing. The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community

- **Message Passing Interface (MPI):** This is a standardized and portable message-passing system to function on a wide variety of parallel computers. MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- **Parallel Virtual Machine (PVM):** It is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor.
- **Remote Procedure Call (RPC):** The Remote Procedure Call (RPC) is a common model of request reply protocol. In RPC, the procedure need not exist in the same address space as the calling procedure.
- **Remote Method Invocation (RMI):** RMI (Remote Method Invocation) is a way that a programmer can write object-oriented programming in which objects on different computers can interact in a distributed network.
- **Remote Procedure Call (RPC):** RPC is a powerful technique for constructing distributed, client-server based applications. In RPC, the procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.
- **Common Object Request Broker Architecture (CORBA):** CORBA describes a messaging mechanism by which objects distributed over a network can communicate with each other irrespective of the platform and language used to develop those objects.

1.6 SYNCHRONOUS VS ASYNCHRONOUS EXECUTIONS

The execution of process in distributed systems may be synchronous or asynchronous.

Asynchronous Execution:

A communication among processes is considered asynchronous, when every communicating process can have a different observation of the order of the messages being exchanged. In an asynchronous execution:

- there is no processor synchrony and there is no bound on the drift rate of processor clocks
- message delays are finite but unbounded
- no upper bound on the time taken by a process

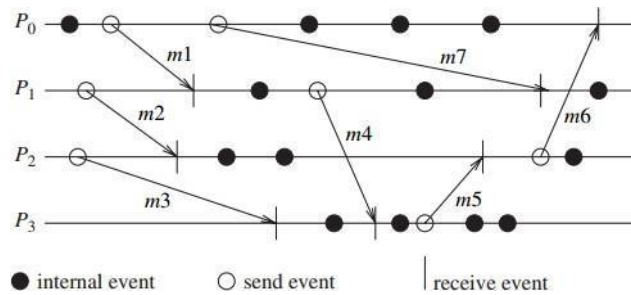
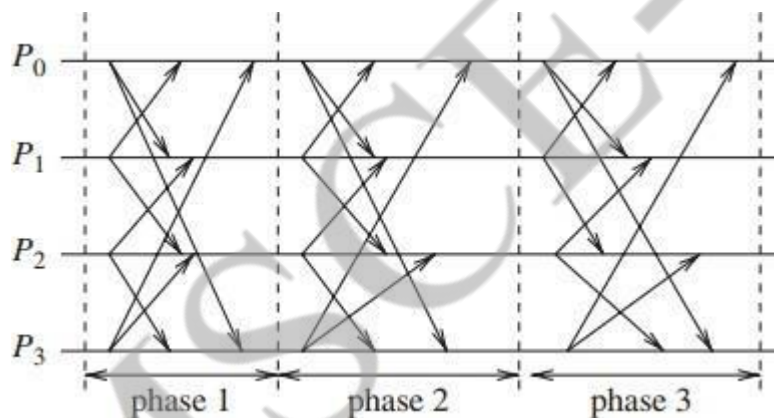


Fig: Asynchronous execution in message passing system

Synchronous Execution:

A communication among processes is considered synchronous when every process observes the same order of messages within the system. In an synchronous execution:

- processors are synchronized and the clock drift rate between any two processors is bounded
- message delivery times are such that they occur in one logical step or round
- upper bound on the time taken by a process to execute a step.



Emulating an asynchronous system by a synchronous system (A → S)

An asynchronous program can be emulated on a synchronous system fairly trivially as the synchronous system is a special case of an asynchronous system – all communication finishes within the same round in which it is initiated.

Emulating a synchronous system by an asynchronous system (S → A)

A synchronous program can be emulated on an asynchronous system using a tool called synchronizer.

Emulation for a fault free system

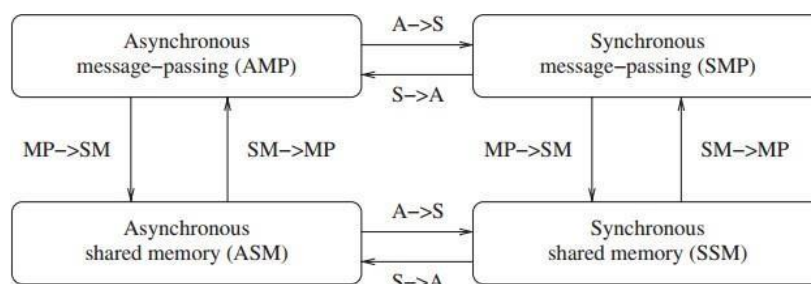


Fig 1.15: Emulations in a failure free message passing system

If system A can be emulated by system B, denoted A/B, and if a problem is not solvable in B, then it is also not solvable in A. If a problem is solvable in A, it is also solvable in B. Hence, in a sense, all four classes are equivalent in terms of computability in failure-free systems.

1.7 DESIGN ISSUES AND CHALLENGES IN DISTRIBUTED SYSTEMS

The design of distributed systems has numerous challenges. They can be categorized into:

- Issues related to system and operating systems design
- Issues related to algorithm design
- Issues arising due to emerging technologies

The above three classes are not mutually exclusive.

1.7.1 Issues related to system and operating systems design

The following are some of the common challenges to be addressed in designing a distributed system from system perspective:

➤ **Communication:** This task involves designing suitable communication mechanisms among the various processes in the networks.

Examples: RPC, RMI

➤ **Processes:** The main challenges involved are: process and thread management at both client and server environments, migration of code between systems, design of software and mobile agents.

➤ **Naming:** Devising easy to use and robust schemes for names, identifiers, and addresses is essential for locating resources and processes in a transparent and scalable manner. The remote and highly varied geographical locations make this task difficult.

➤ **Synchronization:** Mutual exclusion, leader election, deploying physical clocks, global state recording are some synchronization mechanisms.

➤ **Data storage and access Schemes:** Designing file systems for easy and efficient data storage with implicit accessing mechanism is very much essential for distributed operation

➤ **Consistency and replication:** The notion of Distributed systems goes hand in hand with replication of data, to provide high degree of scalability. The replicas should be handled with care since data consistency is prime issue.

➤ **Fault tolerance:** This requires maintenance of fail proof links, nodes, and processes. Some of the common fault tolerant techniques are resilience, reliable communication, distributed commit, checkpointing and recovery, agreement and consensus, failure detection, and self-stabilization.

➤ **Security:** Cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management are some of the security measure that is imposed on distributed systems.

➤ **Applications Programming Interface (API) and transparency:** The user friendliness and ease of use is very important to make the distributed services to be used by wide community. Transparency, which is hiding inner implementation policy from users, is of the following types:

- **Access transparency:** hides differences in data representation
- **Location transparency:** hides differences in locations y providing uniform access to data located at remote locations.

- **Migration transparency:** allows relocating resources without changing names.
- **Replication transparency:** Makes the user unaware whether he is working on original or replicated data.
- **Concurrency transparency:** Masks the concurrent use of shared resources for the user.
- **Failure transparency:** system being reliable and fault-tolerant.
- **Scalability and modularity:** The algorithms, data and services must be as distributed as possible. Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.

1.7.2 Algorithmic challenges in distributed computing

➤ **Designing useful execution models and frameworks**

The interleaving model, partial order model, input/output automata model and the Temporal Logic of Actions (TLA) are some examples of models that provide different degrees of infrastructure.

➤ **Dynamic distributed graph algorithms and distributed routing algorithms**

- The distributed system is generally modeled as a distributed graph.
- Hence graph algorithms are the base for large number of higher level communication, data dissemination, object location, and object search functions.
- These algorithms must have the capacity to deal with highly dynamic graph characteristics. They are expected to function like routing algorithms.
- The performance of these algorithms has direct impact on user-perceived latency, data traffic and load in the network.

➤ **Time and global state in a distributed system**

- The geographically remote resources demands the synchronization based on logical time.
 - Logical time is relative and eliminates the overheads of providing physical time for applications. Logical time can
 - (i) Capture the logic and inter-process dependencies
 - (ii) track the relative progress at each process
 - Maintaining the global state of the system across space involves the role of time dimension for consistency. This can be done with extra effort in a coordinated manner.
 - Deriving appropriate measures of concurrency also involves the time dimension, as the execution and communication speed of threads may vary a lot.
- #### ➤ **Synchronization/coordination mechanisms**
- Synchronization is essential for the distributed processes to facilitate concurrent execution without affecting other processes.

- The synchronization mechanisms also involve resource management and concurrency management mechanisms.

- Some techniques for providing synchronization are:

- ✓ **Physical clock synchronization:** Physical clocks usually diverge in their values due to hardware limitations. Keeping them synchronized is a fundamental challenge to maintain common time.

- ✓ **Leader election:** All the processes need to agree on which process will play the role of a distinguished process or a leader process. A leader is necessary even for many distributed algorithms because there is often some asymmetry.

- ✓ **Mutual exclusion:** Access to the critical resource(s) has to be coordinated.

- ✓ **Deadlock detection and resolution:** This is done to avoid duplicate work, and deadlock resolution should be coordinated to avoid unnecessary aborts of processes.

- ✓ **Termination detection:** cooperation among the processes to detect the specific global state of quiescence.
- ✓ **Garbage collection:** Detecting garbage requires coordination among the processes.
- **Group communication, multicast, and ordered message delivery**
 - A group is a collection of processes that share a common context and collaborate on a common task within an application domain. Group management protocols are needed for group communication wherein processes can join and leave groups dynamically, or fail.
- **Monitoring distributed events and predicates**
 - Predicates defined on program variables that are local to different processes are used for specifying conditions on the global system state.
 - On-line algorithms for monitoring such predicates are hence important.
 - The specification of such predicates uses physical or logical time relationships.
- **Distributed program design and verification tools**

Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering. Designing these is a big challenge.
- **Debugging distributed programs**

Debugging distributed programs is much harder because of the concurrency and replications. Adequate debugging mechanisms and tools are need of the hour.
- **Data replication, consistency models, and caching**
 - Fast access to data and other resources is important in distributed systems.
 - Managing replicas and their updates faces concurrency problems.
 - Placement of the replicas in the systems is also a challenge because resources usually cannot be freely replicated.
- **World Wide Web design – caching, searching, scheduling**
 - WWW is a commonly known distributed system.
 - The issues of object replication and caching, prefetching of objects have to be done on WWW also.
 - Object search and navigation on the web are important functions in the operation of the web.
- **Distributed shared memory abstraction**
 - A shared memory is easier to implement since it does not involve managing the communication tasks.
 - The communication is done by the middleware by message passing.
 - The overhead of shared memory is to be dealt by the middleware technology.
 - Some of the methodologies that does the task of communication in shared memory distributed systems are:
 - ✓ **Wait-free algorithms:** The ability of a process to complete its execution irrespective of the actions of other processes is wait free algorithm. They control the access to shared resources in the shared memory abstraction. They are expensive.
 - ✓ **Mutual exclusion:** Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner. Only one process is allowed to execute the critical section at any given time. In a distributed system, shared variables or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion.
- ✓ **Register constructions:** Architectures must be designed in such a way that, registers allows concurrent access without any restrictions on the concurrency permitted.
- **Reliable and fault-tolerant distributed systems**

The following are some of the fault tolerant strategies:

✓ **Consensus algorithms:** Consensus algorithms allow correctly functioning processes to reach agreement among themselves in spite of the existence of malicious processes. The goal of the malicious processes is to prevent the correctly functioning processes from reaching agreement. The malicious processes operate by sending messages with misleading information, to confuse the correctly functioning processes.

✓ **Replication and replica management:** The Triple Modular Redundancy (TMR) technique is used in software and hardware implementation. TMR is a fault-tolerant form of N-modular redundancy, in which three systems perform a process and that result is processed by a majority-voting system to produce a single output.

✓ **Voting and quorum systems:** Providing redundancy in the active or passive components in the system and then performing voting based on some quorum criterion is a classical way of dealing with fault-tolerance. Designing efficient algorithms for this purpose is the challenge.

✓ **Distributed databases and distributed commit:** The distributed databases should also follow atomicity, consistency, isolation and durability (ACID) properties.

✓ **Self-stabilizing systems:** A self-stabilizing algorithm guarantee to take the system to a good state even if a bad state were to arise due to some error. Self-stabilizing algorithms require some in-built redundancy to track additional variables of the state and do extra work.

✓ **Checkpointing and recovery algorithms:** Checkpointing is periodically recording the current state on secondary storage so that, in case of a failure. The entire computation is not lost but can be recovered from one of the recently taken checkpoints. Checkpointing in

distributed environment is difficult because if the checkpoints at the different processes are not coordinated, the local checkpoints may become useless because they are inconsistent with the checkpoints at other processes.

✓ **Failure detectors:** The asynchronous distributed do not have a bound on the message transmission time. This makes the message passing very difficult, since the receiver do not know the waiting time. Failure detectors probabilistically suspect another process as having failed and then converge on a determination of the up/down status of the suspected process.

➤ **Load balancing**

The objective of load balancing is to gain higher throughput, and reduce the user perceived latency. Load balancing may be necessary because of a variety of factors such as high network traffic or high request rate causing the network connection to be a bottleneck, or high computational load. The following are some forms of load balancing:

✓ **Data migration:** The ability to move data around in the system, based on the access pattern of the users

✓ **Computation migration:** The ability to relocate processes in order to perform an even distribution of the workload.

✓ **Distributed scheduling:** This achieves a better turnaround time for the users by using idle processing power in the system more efficiently.

➤ **Real-time scheduling**

Real-time scheduling becomes more challenging when a global view of the system state is absent with more frequent on-line or dynamic changes. The message propagation delays which are network-dependent are hard to control or predict. This is an hindrance to meet the QoS requirements of the network.

➤ **Performance**

User perceived latency in distributed systems must be reduced. The common issues in performance:

✓ **Metrics:** Appropriate metrics must be defined for measuring the performance of theoretical distributed algorithms and its implementation.

✓ **Measurement methods/tools:** The distributed system is a complex entity appropriate methodology and tools must be developed for measuring the performance metrics.

1.7.3 Applications of distributed computing and newer challenges

The deployment environment of distributed systems ranges from mobile systems to cloud storage. All the environments have their own challenges:

➤ **Mobile systems**

- Mobile systems which use wireless communication in shared broadcast medium have issues related to physical layer such as transmission range, power, battery power consumption, interfacing with wired internet, signal processing and interference.
- The issues pertaining to other higher layers include routing, location management, channel allocation, localization and position estimation, and mobility management.
- Apart from the above mentioned common challenges, the architectural differences of the mobile network demands varied treatment. The two architectures are:

✓ **Base-station approach (cellular approach):** The geographical region is divided into hexagonal physical locations called cells. The powerful base station transmits signals to all other nodes in its range

✓ **Ad-hoc network approach:** This is an infrastructure-less approach which do not have any base station to transmit signals. Instead all the responsibility is distributed among the mobile nodes.

✓ It is evident that both the approaches work in different environment with different principles of communication. Designing a distributed system to cater the varied need is a great challenge.

➤ **Sensor networks**

- A sensor is a processor with an electro-mechanical interface that is capable of sensing physical parameters.
- They are low cost equipment with limited computational power and battery life. They are designed to handle streaming data and route it to external computer network and processes.
- They are susceptible to faults and have to reconfigure themselves.
- These features introduces a whole new set of challenges, such as position estimation and time estimation when designing a distributed system .

➤ **Ubiquitous or pervasive computing**

- In Ubiquitous systems the processors are embedded in the environment to perform application functions in the background.
- **Examples:** Intelligent devices, smart homes etc.
- They are distributed systems with recent advancements operating in wireless environments through actuator mechanisms.
- They can be self-organizing and network-centric with limited resources.

➤ **Peer-to-peer computing**

- Peer-to-peer (P2P) computing is computing over an application layer network where all interactions among the processors are at a same level.
- This is a form of symmetric computation against the client sever paradigm.
- They are self-organizing with or without regular structure to the network.

Some of the key challenges include: object storage mechanisms, efficient object lookup, and retrieval in a scalable manner; dynamic reconfiguration with nodes as well as objects joining and leaving the network randomly; replication strategies to expedite object search; tradeoffs between object size latency and table sizes; anonymity, privacy, and security

- **Publish-subscribe, content distribution, and multimedia**
 - The users in present day require only the information of interest.
 - In a dynamic environment where the information constantly fluctuates there is great demand for
 - **Publish:** an efficient mechanism for distributing this information
 - **Subscribe:** an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information
 - An efficient mechanism for aggregating large volumes of published information and filtering it as per the user's subscription filter.
 - Content distribution refers to a mechanism that categorizes the information based on parameters.
 - The publish subscribe and content distribution overlap each other.
 - Multimedia data introduces special issue because of its large size.
- **Distributed agents**
 - Agents are software processes or sometimes robots that move around the system to do specific tasks for which they are programmed.
 - Agents collect and process information and can exchange such information with other agents.
 - Challenges in distributed agent systems include coordination mechanisms among the agents, controlling the mobility of the agents, their software design and interfaces.
- **Distributed data mining**
 - Data mining algorithms process large amount of data to detect patterns and trends in the data, to mine or extract useful information.
 - The mining can be done by applying database and artificial intelligence techniques to a data repository.
- **Grid computing**
 - Grid computing is deployed to manage resources. For instance, idle CPU cycles of machines connected to the network will be available to others.
 - The challenges includes: scheduling jobs, framework for implementing quality of service, real-time guarantees, security.
- **Security in distributed systems**

The challenges of security in a distributed setting include: confidentiality, authentication and availability. This can be addressed using efficient and scalable solutions.

1.8 A MODEL OF DISTRIBUTED COMPUTATIONS: DISTRIBUTED PROGRAM

- A distributed program is composed of a set of asynchronous processes that communicate by message passing over the communication network. Each process may run on different processor.
- The processes do not share a global memory and communicate solely by passing messages. These processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous – a process may execute an action spontaneously and a process sending a message does not wait for the delivery of the message to be complete.
- The global state of a distributed computation is composed of the states of the processes and the communication channels. The state of a process is characterized by the state of its local memory and depends upon the context.
- The state of a channel is characterized by the set of messages in transit in the channel.

A MODEL OF DISTRIBUTED EXECUTIONS

- The execution of a process consists of a sequential execution of its actions.
- The actions are atomic and the actions of a process are modeled as three types of events: internal events, message send events, and message receive events.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- The execution of process p_i produces a sequence of events e_1, e_2, e_3, \dots , and it is denoted by H_i : $H_i = (h_i \rightarrow i)$. Here h_i are states produced by p_i and \rightarrow are the casual dependencies among events p_i .
- \rightarrow_{msg} indicates the dependency that exists due to message passing between two events.

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

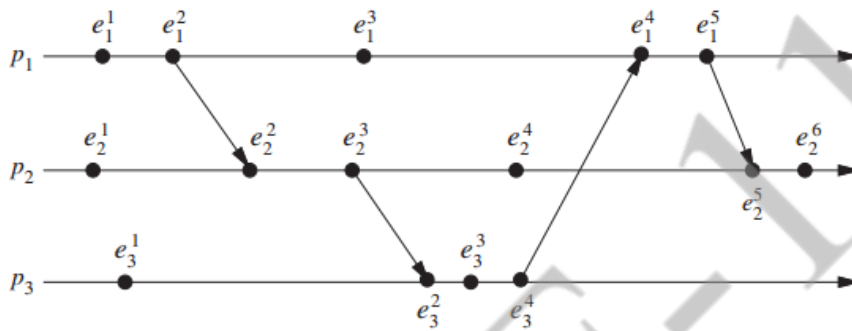


Fig Space time distribution of distributed systems

- An internal event changes the state of the process at which it occurs. A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that receives the message and the state of the channel on which the message is received.

Casual Precedence Relations

Casual message ordering is a partial ordering of messages in a distributed computing environment. It is the delivery of messages to a process in the order in which they were transmitted to that process.

It places a restriction on communication between processes by requiring that if the transmission of message m_i to process p_k necessarily preceded the transmission of message m_j to the same process, then the delivery of these messages to that process must be ordered such that m_i is delivered before m_j .

Happen Before Relation

The partial ordering obtained by generalizing the relationship between two process is called as ***happened-before relation or causal ordering or potential causal ordering***. This term was coined by Lamport. Happens-before defines a partial order of events in a distributed system. Some events can't be placed in the order. If say $A \rightarrow B$ if A happens before B. $A \square B$ is defined using the following rules:

- ✓ **Local ordering:** A and B occur on same process and A occurs before B.
- ✓ **Messages:** send(m) → receive(m) for any message m
- ✓ **Transitivity:** $e \rightarrow e''$ if $e \rightarrow e'$ and $e' \rightarrow e''$
- Ordering can be based on two situations:
 1. If two events occur in same process then they occurred in the order observed.
 2. During message passing, the event of sending message occurred before the event of receiving it.

Lamports ordering is happen before relation denoted by \rightarrow

- $a \rightarrow b$, if a and b are events in the same process and a occurred before b.
- $a \rightarrow b$, if a is the vent of sending a message m in a process and b is the event of the same message m being received by another process.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. Lamports law follow transitivity property.

When all the above conditions are satisfied, then it can be concluded that $a \rightarrow b$ is casually related. Consider two events c and d; $c \rightarrow d$ and $d \rightarrow c$ is false (i.e) they are not casually related, then c and d are said to be concurrent events denoted as $c \parallel d$.

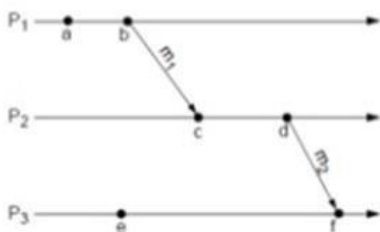


Fig Communication between processes

Fig 1.22 shows the communication of messages m1 and m2 between three processes p1, p2 and p3. a, b, c, d, e and f are events. It can be inferred from the diagram that, $a \rightarrow b$; $c \rightarrow d$; $e \rightarrow f$; $b \rightarrow c$; $d \rightarrow f$; $a \rightarrow d$; $a \rightarrow f$; $b \rightarrow d$; $b \rightarrow f$. Also $a \parallel e$ and $c \parallel e$.

Logical vs physical concurrency

Physical as well as logical concurrency is two events that creates confusion in distributed systems.

Physical concurrency: Several program units from the same program that execute simultaneously.

Logical concurrency: Multiple processors providing actual concurrency. The actual execution of programs is taking place in interleaved fashion on a single processor.

Differences between logical and physical concurrency

Logical concurrency	Physical concurrency
Several units of the same program execute simultaneously on same processor, giving an illusion to the programmer that they are executing on multiple processors.	Several program units of the same program execute at the same time on different processors.
They are implemented through interleaving.	They are implemented as uni-processor with I/O channels, multiple CPUs, network of uni or multi CPU machines.

MODELS OF COMMUNICATION NETWORK

The three main types of communication models in distributed systems are:

FIFO (first-in, first-out): each channel acts as a FIFO message queue.

Non-FIFO (N-FIFO): a channel acts like a set in which a sender process adds messages and receiver removes messages in random order.

Causal Ordering (CO): It follows Lamport's law.

- o The relation between the three models is given by $CO \sqsupseteq FIFO \sqsupseteq N-FIFO$.

A system that supports the causal ordering model satisfies the following property:

CO: For any two messages m_{ij} and m_{kj} , if $send(m_{ij}) \rightarrow send(m_{kj})$,
then $rec(m_{ij}) \rightarrow rec(m_{kj})$.

GLOBAL STATE

Distributed Snapshot represents a state in which the distributed system might have been in. A snapshot of the system is a single configuration of the system.

- The global state of a distributed system is a collection of the local states of its components, namely, the processes and the communication channels.
- The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of a channel is given by the set of messages in transit in the channel.

The state of a channel is difficult to state formally because a channel is a distributed entity and its state depends upon the states of the processes it connects. Let $SC_{ij}^{x,y}$ denote the state of a channel C_{ij} defined as follows:

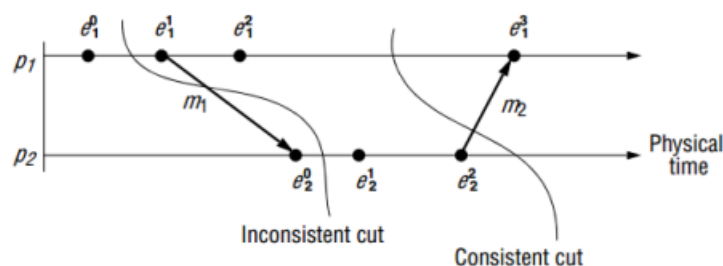
$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq LS_i^x \wedge rec(m_{ij}) \not\leq LS_j^y\}.$$

For a successful Global State, all states must be consistent:

- If we have recorded that a process P has received a message from a process Q, then we should have also recorded that process Q had actually send that message.
- Otherwise, a snapshot will contain the recording of messages that have been received but never sent.
- The reverse condition (Q has sent a message that P has not received) is allowed.

Consistent states: The states should not violate causality. Such states are called consistent global states and are meaningful global states.

Inconsistent global states: They are not meaningful in the sense that a distributed system



can never be in an inconsistent state.

UNIT II

LOGICAL TIME & GLOBAL STATE

Logical clocks are based on capturing chronological and causal relationships of processes and ordering events based on these relationships.

Three types of logical clock are maintained in distributed systems:

- Scalar clock
- Vector clock
- Matrix clock

In a system of logical clocks, every process has a logical clock that is advanced using a set of rules. Every event is assigned a timestamp and the causality relation between events can be generally inferred from their timestamps.

The timestamps assigned to events obey the fundamental monotonicity property; that is, if an event a causally affects an event b , then the timestamp of a is smaller than the timestamp of b .

A Framework for a system of logical clocks

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 $<$. This relation is usually called the happened before or causal precedence.

The logical clock C is a function that maps an event e in a distributed system to an element in the time domain T denoted as $C(e)$.

$C : H \mapsto T$ such that

for any two events e_i and e_j , $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$

This monotonicity property is called the **clock consistency condition**. When T and C satisfy the following condition,

$$e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$$

Then the system of clocks is **strongly consistent**.

Implementing logical clocks

The two major issues in implanting logical clocks are:

Data structures: representation of each process

Protocols: rules for updating the data structures to ensure consistent conditions.

Data structures:

Each process p_i maintains data structures with the given capabilities:

- A local logical clock (lc_i), that helps process p_i measure its own progress.
- A logical global clock (gc_i), that is a representation of process p_i 's local view of the logical global time. It allows this process to assign consistent timestamps to its local events.

Protocol:

The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently with the following rules:

Rule 1: Decides the updates of the logical clock by a process. It controls send, receive and other operations.

Rule 2: Decides how a process updates its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

2.1.1 SCALAR TIME

Scalar time is designed by Lamport to synchronize all the events in distributed systems. A Lamport logical clock is an incrementing counter maintained in each process. When a process receives a message, it resynchronizes its logical clock with that sender maintaining causal relationship.

The Lamport's algorithm is governed using the following rules:

- The algorithm of Lamport Timestamps can be captured in a few rules:
- All the process counters start with value 0.
- A process increments its counter for each event (internal event, message sending, message receiving) in that process.
- When a process sends a message, it includes its (incremented) counter value with the message.
- On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one.
- If C_i is the local clock for process P_i then,
- if a and b are two successive events in P_i , then $C_i(b) = C_i(a) + d1$, where $d1 > 0$
- if a is the sending of message m by P_i , then m is assigned timestamp $t_m = C_i(a)$
- if b is the receipt of m by P_j , then $C_j(b) = \max\{C_j(b), t_m + d2\}$, where $d2 > 0$

Rules of Lamport's clock

Rule 1: $C_i(b) = C_i(a) + d1$, where $d1 > 0$

Rule 2: The following actions are implemented when p_i receives a message m with timestamp C_m :

a) $C_i = \max(C_i, C_m)$

b) Execute Rule 1

c) deliver the message

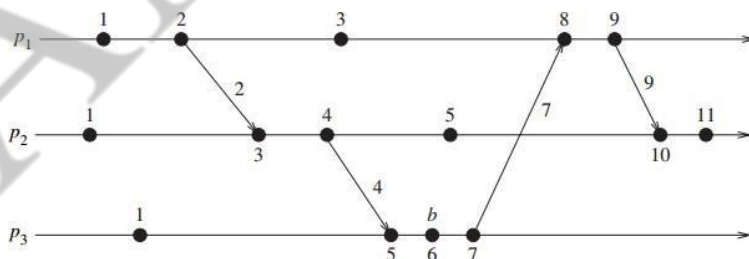


Fig 1.20: Evolution of scalar time

Basic properties of scalar time:

1. Consistency property: Scalar clock always satisfies monotonicity. A monotonic clock only increments its timestamp and never jump. Hence it is consistent.

$$C(e_i) < C(e_j).$$

2. Total Reordering: Scalar clocks order the events in distributed systems. But all the events do not follow a common identical timestamp. Hence a tie breaking mechanism is essential to order the events. The tie breaking is done through:

- Linearly order process identifiers.
- Process with low identifier value will be given higher priority.

The term (t, i) indicates timestamp of an event, where t is its time of occurrence and i is the identity of the process where it occurred.

The total order relation (\prec) over two events x and y with timestamp (h, i) and (k, j) is given by:

$$x \prec y \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$$

A total order is generally used to ensure liveness properties in distributed algorithms.

3. Event Counting

If event e has a timestamp h , then $h-1$ represents the minimum logical duration, counted in units of events, required before producing the event e . This is called **height** of the event e . $h-1$ events have been produced sequentially before the event e regardless of the processes that produced these events.

4. No strong consistency

The scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.

2.1.2 VECTOR TIME

The ordering from Lamport's clocks is not enough to guarantee that if two events precede one another in the ordering relation they are also causally related. Vector Clocks use a vector counter instead of an integer counter. The vector clock of a system with N processes is a vector of N counters, one counter per process. Vector counters have to follow the following update rules:

- Initially, all counters are zero.
- Each time a process experiences an event, it increments its own counter in the vector by one.

- Each time a process sends a message, it includes a copy of its own (incremented) vector in the message.
- Each time a process receives a message, it increments its own counter in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector counter and the value in the vector in the received message.

The time domain is represented by a set of n-dimensional non-negative integer vectors in vector time.

Rules of Vector Time

Rule 1: Before executing an event, process p_i updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \quad (d > 0)$$

Rule 2: Each message m is piggybacked with the vector clock vt of the sender process at sending time. On the receipt of such a message (m, vt) , process p_i executes the following sequence of actions:

1. update its global logical time

$$1 \leq k \leq n : vt_i[k] := \max(vt_i[k], vt[k])$$

2. execute $R1$

3. deliver the message m

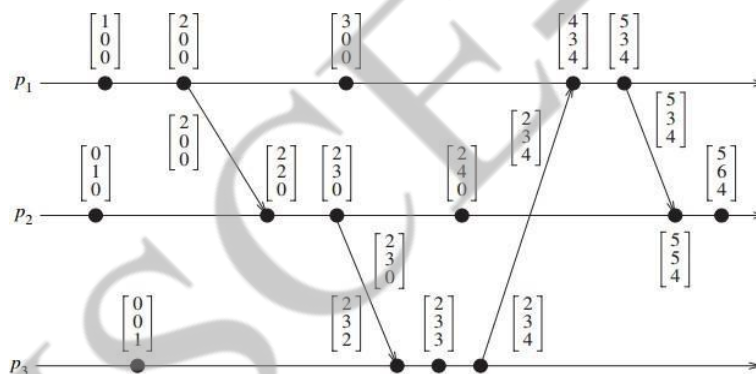


Fig 1.21: Evolution of vector scale

Basic properties of vector time

1. Isomorphism:

- “ \rightarrow ” induces a partial order on the set of events that are produced by a distributed execution.
- If events x and y are timestamped as vh and vk then,

$$x \rightarrow y \Leftrightarrow vh < vk$$

$$\square \quad x \parallel y \Leftrightarrow vh \parallel vk.$$

- If the process at which an event occurred is known, the test to compare two timestamps can be simplified as:

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

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

2. Strong consistency

The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.

3. Event counting

If an event e has timestamp $vh[i]$, $vh[j]$ denotes the number of events executed by process p_j that causally precede e .

2.2 PHYSICAL CLOCK SYNCHRONIZATION: NETWORK TIME PROTOCOL (NTP)

Centralized systems do not need clock synchronization, as they work under a common clock. But the distributed systems do not follow common clock: each system functions based on its own internal clock and its own notion of time. The time in distributed systems is measured in the following contexts:

- The time of the day at which an event happened on a specific machine in the network.
- The time interval between two events that happened on different machines in the network.
- The relative ordering of events that happened on different machines in the network.

Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time.

Due to different clocks rates, the clocks at various sites may diverge with time, and periodically a clock synchronization must be performed to correct this clock skew in distributed systems. Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time). Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed **physical clocks**. This degree of synchronization additionally enables to coordinate and schedule actions between multiple computers connected to a common network.

Basic terminologies:

If C_a and C_b are two different clocks, then:

- **Time:** The time of a clock in a machine p is given by the function $C_p(t)$, where $C_p(t) = t$ for a perfect clock.
- **Frequency:** Frequency is the rate at which a clock progresses. The frequency at time t of clock C_a is $C_a'(t)$.
- **Offset:** Clock offset is the difference between the time reported by a clock and the real time. The offset of the clock C_a is given by $C_a(t) - t$. The offset of clock C_a relative to C_b at time $t \geq 0$ is given by $C_a(t) - C_b(t)$.
- **Skew:** The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock C_a relative to clock C_b at time t is $C_a'(t) - C_b'(t)$.
- **Drift (rate):** The drift of clock C_a is the second derivative of the clock value with respect to time. The drift is calculated as:

$$C_a''(t) - C_b''(t).$$

Clocking Inaccuracies

Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time). Due to the clock inaccuracy discussed above, a timer (clock) is said to be working within its specification if:

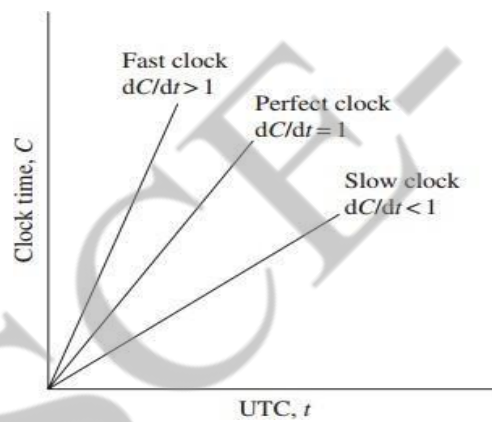
$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho.$$

1. Offset delay estimation

A time service for the Internet - synchronizes clients to UTC Reliability from redundant paths, scalable, authenticates time sources Architecture. The design of NTP involves a hierarchical tree of time servers with primary server at the root synchronizes with the UTC. The next level contains secondary servers, which act as a backup to the primary server. At the lowest level is the synchronization subnet which has the clients.

2. Clock offset and delay estimation

A source node cannot accurately estimate the local time on the target node due to varying message or network delays between the nodes. This protocol employs a very common practice of performing several trials and chooses the trial with the minimum



delay.

Fig : Behavior of clocks

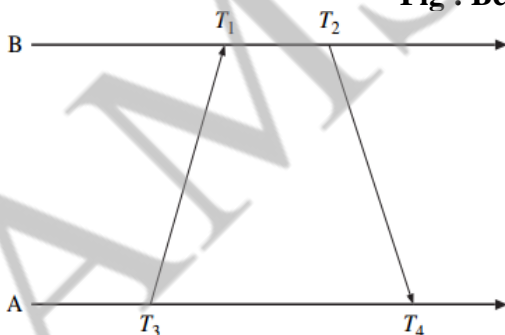


Fig a) Offset and delay estimation between processes from same server

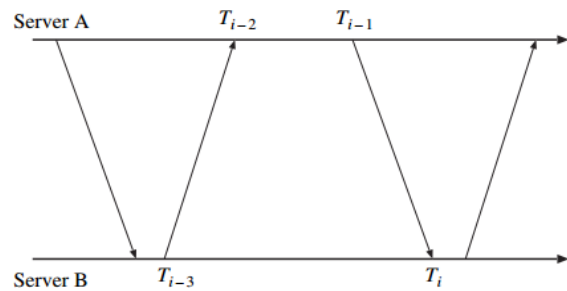


Fig b) Offset and delay estimation between processes from different servers

Let T_1, T_2, T_3, T_4 be the values of the four most recent timestamps. The clocks A and B are stable and running at the same speed. Let $a = T_1 - T_3$ and $b = T_2 - T_4$. If the network delay difference from A to B and from B to A, called **differential delay**, is

small, the clock offset and roundtrip delay of B relative to A at time T4 are approximately given by the following:

$$\theta = \frac{a+b}{2}, \quad \delta = a-b$$

Each NTP message includes the latest three timestamps T1, T2, and T3, while T4 is determined upon arrival.

2.3 MESSAGE ORDERING AND GROUP COMMUNICATION

As the distributed systems are a network of systems at various physical locations, the coordination between them should always be preserved. The message ordering means the order of delivering the messages to the intended recipients. The common message order schemes are First in First out (FIFO), non FIFO, causal order and synchronous order. In case of group communication with multicasting, the causal and total ordering scheme is followed. It is also essential to define the behaviour of the system in case of failures. The following are the notations that are widely used in this chapter:

- Distributed systems are denoted by a graph (N, L).
- The set of events are represented by event set {E, < }
- Message is denoted as mⁱ: send and receive events as sⁱ and rⁱ respectively.
- Send (M) and receive (M) indicates the message M send and received.
- a □ b denotes a and b occurs at the same process
- The send receive pairs □ = {(s, r) □ E_i x E_j} corresponds to r }

2.3.1 MESSAGE ORDERING PARADIGMS

The message orderings are

- (i) non-FIFO
- (ii) FIFO
- (iii) causal order
- (iv) synchronous order

There is always a trade-off between concurrency and ease of use and implementation.

Asynchronous Executions

An asynchronous execution (or A-execution) is an execution (E, <) for which the causality relation is a partial order.

- There cannot be any causal relationship between events in asynchronous execution.
- The messages can be delivered in any order even in non FIFO.
- Though there is a physical link that delivers the messages sent on it in FIFO order due to the physical properties of the medium, a may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link.

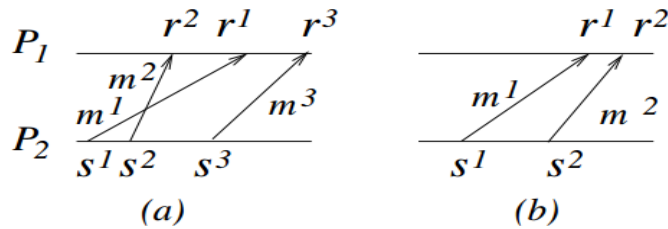


Fig 2.1: a) FIFO executions b) non FIFO executions

FIFO executions

A FIFO execution is an A-execution in which, for all (s, r) and $(s', r') \in \mathcal{T}$, $(s \sim s' \text{ and } r \sim r' \text{ and } s < s') \implies r < r'$.

- The logical link is non-FIFO.
- A FIFO logical channel can be created over a non-FIFO channel by using a separate numbering scheme to sequence the messages on each logical channel.
- The sender assigns and appends a $\langle \text{sequence_num}, \text{connection_id} \rangle$ tuple to each message.
- The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence.

Causally Ordered (CO) executions

CO execution is an A-execution in which, for all, (s, r) and $(s', r') \in \mathcal{T}$, $(r \sim r' \text{ and } s < s') \implies r < r'$

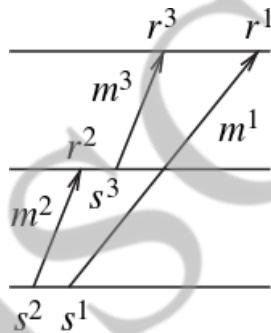


Fig: CO Execution

- Two send events s and s' are related by causality ordering (not physical time ordering), then a causally ordered execution requires that their corresponding receive events r and r' occur in the same order at all common destinations.

Applications of causal order:

Applications that requires update to shared data to implement distributed shared memory, and fair resource allocation in distributed mutual exclusion.

Causal Order(CO) for Implementations:

If $send(m^1) < send(m^2)$ then for each common destination d of messages m^1 and m^2 , $deliverd(m^1) < deliverd(m^2)$ must be satisfied.

Other properties of causal ordering

1. Message Order (MO): A MO execution is an A-execution in which, for all

(s, r) and $(s', r') \in \mathcal{T}$, $s < s' \implies \neg(r' < r)$.

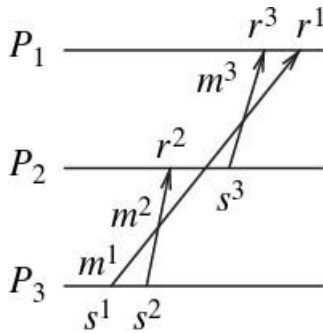


Fig: Not a CO execution

Empty Interval Execution: An execution $(E, <)$ is an empty-interval (EI) execution if for each pair of events $(s, r) \in \mathcal{T}$, the open interval set

$$\{x \in E \mid s < x < r\}$$

in the partial order is empty.

An execution $(E, <)$ is CO if and only if for each pair of events $(s, r) \in \mathcal{T}$ and each event $e \in E$,

- weak common past:

$$e < r \implies \neg(s < e)$$

- weak common future:

$$s < e \implies \neg(e < r).$$

Synchronous Execution

- When all the communication between pairs of processes uses synchronous send and receives primitives, the resulting order is the synchronous order.
- The synchronous communication always involves a handshake between the receiver and the sender, the handshake events may appear to be occurring instantaneously and atomically.

Causality in a synchronous execution

The synchronous causality relation \ll on E is the smallest transitive relation that satisfies the following: S1: If x occurs before y at the same process, then $x \ll y$.

S2: If $sr \in \mathcal{T}$, then for all $x \in E$, $[(x \ll s \iff x \ll r) \text{ and } (s \ll x \iff r \ll x)]$.

S3: If $x \ll y$ and $y \ll z$, then $x \ll z$

Synchronous Execution:

A synchronous execution (or S-execution) is an execution (E, \ll) for which the causality relation \ll is a partial order

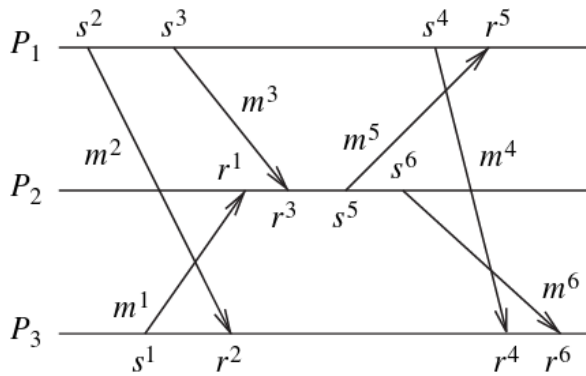


Fig) Execution in an asynchronous system

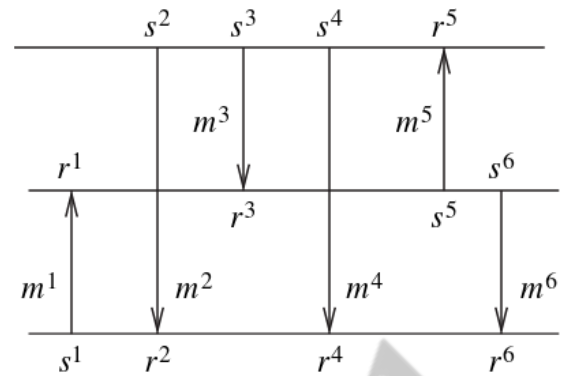


Fig) Execution in synchronous

Timestamping a synchronous execution

An execution(E <)is synchronous if and only if there exists a mapping from E to T (scalar timestamps) such that

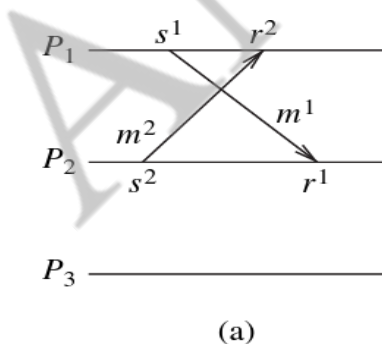
- for any message M, $T(s(M))=T(r(M))$;
- for each process P_i , if $e_i < e_i^1$ then $T(e_i) < T(e_i^1)$.

2.4 ASYNCHRONOUS EXECUTION WITH SYNCHRONOUS COMMUNICATION

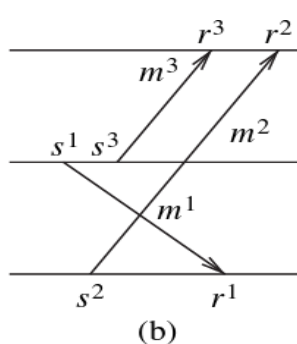
When all the communication between pairs of processes is by using synchronous send and receive primitives, the resulting order is synchronous order. If a program is written for an asynchronous system, say a FIFO system, will it still execute correctly if the communication is done by synchronous primitives. There is a possibility that the program may deadlock,



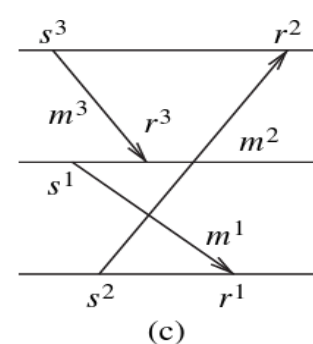
Fig) A communication program for an asynchronous system deadlock when using synchronous primitives



(a)



(b)



(c)

Fig) Illustrations of asynchronous execution and of crowns
 crown of size 2
 crown of size 3
 Crown of size 2

Realizable Synchronous Communication (RSC)

A-execution can be realized under synchronous communication is called a realizable with synchronous communication (RSC).

An execution can be modeled to give a total order that extends the partial order $(E, <)$.

In an A-execution, the messages can be made to appear instantaneous if there exist a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event in this linear extension.

Non-separated linear extension is an extension of $(E, <)$ is a linear extension of $(E, <)$ such that for each pair $(s, r) \in T$, the interval $\{x \in E \mid s < x < r\}$ is empty.

A-execution $(E, <)$ is an RSC execution if and only if there exists a non-separated linear extension of the partial order $(E, <)$.

In the non-separated linear extension, if the adjacent send event and its corresponding receive event are viewed atomically, then that pair of events shares a common past and a common future with each other.

Crown

Let E be an execution. A crown of size k in E is a sequence $\langle (s_i, r_i), i \in \{0, \dots, k-1\} \rangle$ of pairs of corresponding send and receive events such that: $s_0 < r_1, s_1 < r_2, s_{k-2} < r_{k-1}, s_{k-1} < r_0$.

The crown is $\langle (s_1, r_1) (s_2, r_2) \rangle$ as we have $s_1 < r_2$ and $s_2 < r_1$. Cyclic dependencies may exist in a crown. The crown criterion states that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, if and only if it contains no crown.

Timestamp criterion for RSC execution

An execution $(E, <)$ is RSC if and only if there exists a mapping from E to T (scalar timestamps) such that

for any message M , $T(s(M)) = T(r(M))$;
for each (a, b) in $(E \times E) \setminus T$, $a < b \implies T(a) < T(b)$

Synchronous programs on asynchronous systems

- A (valid) S-execution can be trivially realized on an asynchronous system by scheduling the messages in the order in which they appear in the S-execution.
- The partial order of the S-execution remains unchanged but the communication occurs on an asynchronous system that uses asynchronous communication primitives.
- Once a message send event is scheduled, the middleware layer waits for acknowledgment; after the ack is received, the synchronous send primitive completes.

2.5 SYNCHRONOUS PROGRAM ORDER ON AN ASYNCHRONOUS SYSTEM

Non deterministic programs

The partial ordering of messages in the distributed systems makes the repeated runs of the same program will produce the same partial order, thus preserving deterministic nature.

But sometimes the distributed systems exhibit non determinism:

- A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.
- Multiple send and receive calls which are enabled at a process can be executed in an interchangeable order.
- If i sends to j , and j sends to i concurrently using blocking synchronous calls, there

results a deadlock.

- There is no semantic dependency between the send and the immediately following receive at each of the processes. If the receive call at one of the processes can be scheduled before the send call, then there is no deadlock.

Rendezvous

Rendezvous systems are a form of synchronous communication among an arbitrary number of asynchronous processes. All the processes involved meet with each other, i.e., communicate synchronously with each other at one time. Two types of rendezvous systems are possible:

- Binary rendezvous: When two processes agree to synchronize.
- Multi-way rendezvous: When more than two processes agree to synchronize.

Features of binary rendezvous:

- For the receive command, the sender must be specified. However, multiple receive commands can exist. A type check on the data is implicitly performed.
- Send and received commands may be individually disabled or enabled. A command is disabled if it is guarded and the guard evaluates to false. The guard would likely contain an expression on some local variables.
- Synchronous communication is implemented by scheduling messages under the covers using asynchronous communication.
- Scheduling involves pairing of matching send and receives commands that are both enabled. The communication events for the control messages under the covers do not alter the partial order of the execution.

2.3.2 Binary rendezvous algorithm

If multiple interactions are enabled, a process chooses one of them and tries to synchronize with the partner process. The problem reduces to one of scheduling messages satisfying the following constraints:

- Schedule on-line, atomically, and in a distributed manner.
- Schedule in a deadlock-free manner (i.e., crown-free).
- Schedule to satisfy the progress property in addition to the safety property.

Steps in Bagrodia algorithm

1. Receive commands are forever enabled from all processes.
2. A send command, once enabled, remains enabled until it completes, i.e., it is not possible that a send command gets before the send is executed.
3. To prevent deadlock, process identifiers are used to introduce asymmetry to break potential crowns that arise.
4. Each process attempts to schedule only one send event at any time.

The message (M) types used are: M, ack(M), request(M), and permission(M). Execution events in the synchronous execution are only the send of the message M and receive of the message M. The send and receive events for the other message types – ack(M), request(M), and permission(M) which are control messages. The messages request(M), ack(M), and permission(M) use M's unique tag; the message M is not included in these messages.

(message types)

M, ack(M), request(M), permission(M)

(1) P_i wants to execute SEND(M) to a lower priority process P_j :

P_i executes *send*(M) and blocks until it receives *ack*(M) from P_j . The send event SEND(M) now completes.

Any M' message (from a higher priority processes) and *request*(M') request for synchronization (from a lower priority processes) received during the blocking period are queued.

(2) P_i wants to execute SEND(M) to a higher priority

process P_j : (2a) P_i seeks permission from P_j by executing *send*(*request*(M)).

(2b) While P_i is waiting for permission, it remains unblocked.

(i) If a message M' arrives from a higher priority process P_k , P_i accepts M' by scheduling a RECEIVE(M') event and then executes *send*(*ack*(M')) to P_k .

(ii) If a *request*(M') arrives from a lower priority process P_k , P_i executes *send*(*permission*(M')) to P_k and blocks waiting for the message M' . When M' arrives, the RECEIVE(M') event is executed.

(2c) When the *permission*(M) arrives, P_i knows partner P_j is synchronized and P_i executes *send*(M). The SEND(M) now completes.

(3) *request*(M) arrival at P_i from a lower priority process P_j :

At the time a *request*(M) is processed by P_i , process P_i executes *send*(*permission*(M)) to P_j and blocks waiting for the message M. When M arrives, the RECEIVE(M) event is executed and the process unblocks.

(4) Message M arrival at P_i from a higher priority process P_j :

At the time a message M is processed by P_i , process P_i executes $RECEIVE(M)$ (which is assumed to be always enabled) and then $send(ack(M))$ to P_j .

(5) Processing when P_i is unblocked:

When P_i is unblocked, it dequeues the next (if any) message from the queue and processes it as a message arrival (as per rules 3 or 4).

Fig 2.5: Bagrodia Algorithm

2.6 GROUP COMMUNICATION

Group communication is done by broadcasting of messages. A message broadcast is the sending of a message to all members in the distributed system. The communication may be

- **Multicast:** A message is sent to a certain subset or a group.
- **Unicasting:** A point-to-point message communication.

The network layer protocol cannot provide the following functionalities:

- Application-specific ordering semantics on the order of delivery of messages.
- Adapting groups to dynamically changing membership.
- Sending multicasts to an arbitrary set of processes at each send event.
- Providing various fault-tolerance semantics.
- The multicast algorithms can be open or closed group.

2.7 CAUSAL ORDER (CO)

In the context of group communication, there are two modes of communication: *causal order* and *total order*. Given a system with FIFO channels, causal order needs to be explicitly enforced by a protocol. The following two criteria must be met by a causal ordering protocol:

- **Safety:** In order to prevent causal order from being violated, a message M that arrives at a process may need to be buffered until all system wide messages sent in the causal past of the send (M) event to that same destination have already arrived. The arrival of a message is transparent to the application process. The delivery event corresponds to the receive event in the execution model.
- **Liveness:** A message that arrives at a process must eventually be delivered to the process.

The Raynal–Schiper–Toueg algorithm

- Each message M should carry a log of all other messages sent causally before M 's send event, and sent to the same destination $dest(M)$.
- The Raynal–Schiper–Toueg algorithm canonical algorithm is a representative of several algorithms that reduces the size of the local space and message space overhead by various techniques.
- This log can then be examined to ensure whether it is safe to deliver a message.

- All algorithms aim to reduce this log overhead, and the space and time overhead of maintaining the log information at the processes.
- To distribute this log information, broadcast and multicast communication is used.
- The hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features:
 - Application-specific ordering semantics on the order of delivery of messages.
 - Adapting groups to dynamically changing membership.
 - Sending multicasts to an arbitrary set of processes at each send event.
 - Providing various fault-tolerance semantics

The Kshem Kalyani – Singhal Optimal Algorithm

An optimal CO algorithm stores in local message logs and propagates on messages, information of the form d is a destination of M about a message M sent in the causal past, as long as and only as long as:

Propagation Constraint I: it is not known that the message M is delivered to d .

Propagation Constraint II: it is not known that a message has been sent to d in the causal future of $\text{Send}(M)$, and hence it is not guaranteed using a reasoning based on transitivity that the message M will be delivered to d in CO.

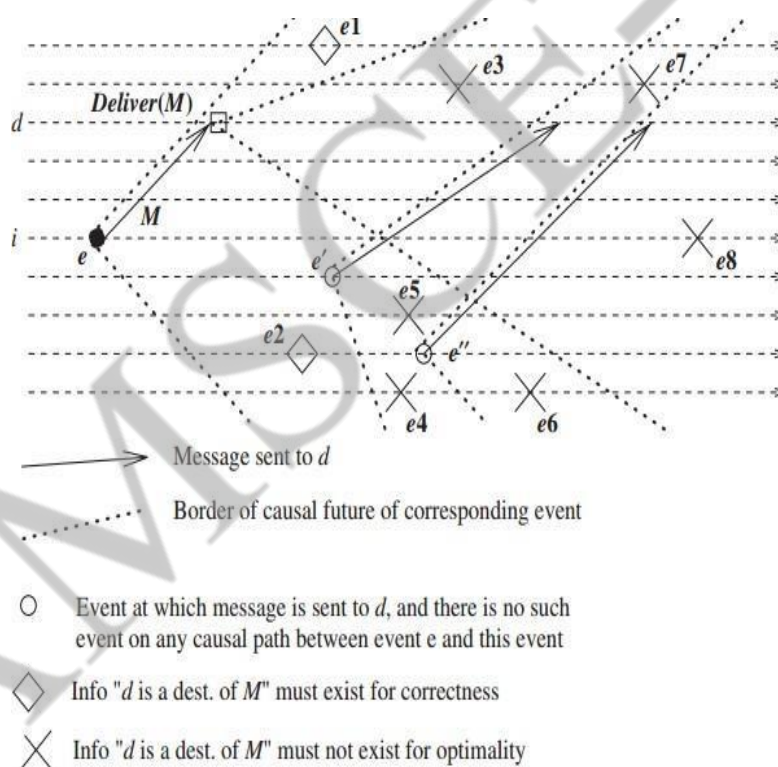


Fig : Conditions for causal ordering

The Propagation Constraints also imply that if either (I) or (II) is false, the information “ $d \in M.Dests$ ” must not be stored or propagated, even to remember that (I) or (II) has been falsified:

- not in the causal future of $Deliverd(M_1, a)$
- not in the causal future of $e_{k, c}$ where $d \in M_{k, c}.Dests$ and there is no other message sent causally between $M_{i, a}$ and $M_{k, c}$ to the same destination d .

Information about messages:

(i) not known to be delivered

(ii) not guaranteed to be delivered in CO, is explicitly tracked by the algorithm using (source, timestamp, destination) information.

Information about messages already delivered and messages guaranteed to be delivered in CO is implicitly tracked without storing or propagating it, and is derived from the explicit information. The algorithm for the send and receive operations is given in Fig. 2.7 a) and b). Procedure SND is executed atomically. Procedure RCV is executed atomically except for a possible interruption in line 2a where a non-blocking wait is required to meet the Delivery Condition.

(1) **SND: j sends a message M to $Dests$:**

```

(1a)  $clock_j \leftarrow clock_j + 1$ ;
(1b) for all  $d \in M.Dests$  do:
     $O_M \leftarrow LOG_j$ ; //  $O_M$  denotes  $O_{M_j, clock_j}$ 
    for all  $o \in O_M$ , modify  $o.Dests$  as follows:
        if  $d \notin o.Dests$  then  $o.Dests \leftarrow (o.Dests \setminus M.Dests)$ ;
        if  $d \in o.Dests$  then  $o.Dests \leftarrow (o.Dests \setminus M.Dests) \cup \{d\}$ ;
        // Do not propagate information about indirect dependencies that are
        // guaranteed to be transitively satisfied when dependencies of  $M$  are satisfied.
        for all  $o_{s, t} \in O_M$  do
            if  $o_{s, t}.Dests = \emptyset \wedge (\exists o'_{s, t'} \in O_M \mid t < t')$  then  $O_M \leftarrow O_M \setminus \{o_{s, t}\}$ ;
            // do not propagate older entries for which  $Dests$  field is  $\emptyset$ 
        send ( $j, clock_j, M, Dests, O_M$ ) to  $d$ ;
(1c) for all  $l \in LOG_j$  do  $l.Dests \leftarrow l.Dests \setminus Dests$ ;
    // Do not store information about indirect dependencies that are guaranteed
    // to be transitively satisfied when dependencies of  $M$  are satisfied.
    Execute  $PURGE\_NULL\_ENTRIES(LOG_j)$ ; // purge  $l \in LOG_j$  if  $l.Dests = \emptyset$ 
(1d)  $LOG_j \leftarrow LOG_j \cup \{(j, clock_j, Dests)\}$ .

```

Fig 2.7 a) Send algorithm by Kshemkalyani–Singhal to optimally implement causal ordering

(2) **RCV:** *j* receives a message $(k, t_k, M, Dest_s, O_M)$ from *k*:

```

(2a) // Delivery Condition: ensure that messages sent causally before M are delivered.
    for all  $o_{m,t_m} \in O_M$  do
        if  $j \in o_{m,t_m}.Dest_s$  wait until  $t_m \leq SR_j[m]$ ;
(2b) Deliver M;  $SR_j[k] \leftarrow t_k$ ;
(2c)  $O_M \leftarrow \{(k, t_k, Dest_s)\} \cup O_M$ ;
    for all  $o_{m,t_m} \in O_M$  do  $o_{m,t_m}.Dest_s \leftarrow o_{m,t_m}.Dest_s \setminus \{j\}$ ;
        // delete the now redundant dependency of message represented by  $o_{m,t_m}$  sent to j
(2d) // Merge  $O_M$  and  $LOG_j$  by eliminating all redundant entries.
    // Implicitly track “already delivered” & “guaranteed to be delivered in CO”
    // messages.
    for all  $o_{m,t} \in O_M$  and  $l_{s,t'} \in LOG_j$  such that  $s = m$  do
        if  $t < t' \wedge l_{s,t} \notin LOG_j$  then mark  $o_{m,t}$ ;
            //  $l_{s,t}$  had been deleted or never inserted, as  $l_{s,t}.Dest_s = \emptyset$  in the causal past
        if  $t' < t \wedge o_{m,t'} \notin O_M$  then mark  $l_{s,t'}$ ;
            //  $o_{m,t'} \notin O_M$  because  $l_{s,t'}$  had become  $\emptyset$  at another process in the causal past
    Delete all marked elements in  $O_M$  and  $LOG_j$ ;
        // delete entries about redundant information
    for all  $l_{s,t'} \in LOG_j$  and  $o_{m,t} \in O_M$ , such that  $s = m \wedge t' = t$  do
         $l_{s,t'}.Dest_s \leftarrow l_{s,t'}.Dest_s \cap o_{m,t}.Dest_s$ ;
            // delete destinations for which Delivery
            // Condition is satisfied or guaranteed to be satisfied as per  $o_{m,t}$ 
        Delete  $o_{m,t}$  from  $O_M$ ; // information has been incorporated in  $l_{s,t'}$ 
     $LOG_j \leftarrow LOG_j \cup O_M$ ; // merge non-redundant information of  $O_M$  into  $LOG_j$ 
(2e) PURGE_NULL_ENTRIES( $LOG_j$ ). // Purge older entries l for which  $l.Dest_s = \emptyset$ 

```

PURGE_NULL_ENTRIES(Log_j): // Purge older entries *l* for which $l.Dest_s = \emptyset$ is
// implicitly inferred

Fig b) Receive algorithm by Kshemkalyani–Singhal to optimally implement causal ordering

The data structures maintained are sorted row–major and then column–major:

1. Explicit tracking:

- Tracking of (source, timestamp, destination) information for messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, is done explicitly using the $l.Dest_s$ field of entries in local logs at nodes and $o.Dest_s$ field of entries in messages.
- Sets $l_{i,a}.Dest_s$ and $o_{i,a}.Dest_s$ contain explicit information of destinations to which $M_{i,a}$ is not guaranteed to be delivered in CO and is not known to be delivered.
- The information about $d \in M_{i,a}.Dest_s$ is propagated up to the earliest events on all causal paths from (i, a) at which it is known that $M_{i,a}$ is delivered to *d* or is guaranteed to be delivered to *d* in CO.

2. Implicit tracking:

- Tracking of messages that are either (i) already delivered, or (ii) guaranteed to be delivered in CO, is performed implicitly.

- The information about messages (i) already delivered or (ii) guaranteed to be delivered in CO is deleted and not propagated because it is redundant as far as enforcing CO is concerned.
- It is useful in determining what information that is being carried in other messages and is being stored in logs at other nodes has become redundant and thus can be purged.
- The semantics are implicitly stored and propagated. This information about messages that are (i) already delivered or (ii) guaranteed to be delivered in CO is tracked without explicitly storing it.
- The algorithm derives it from the existing explicit information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, by examining only $oi,aDests$ or $li,aDests$, which is a part of the explicit information.

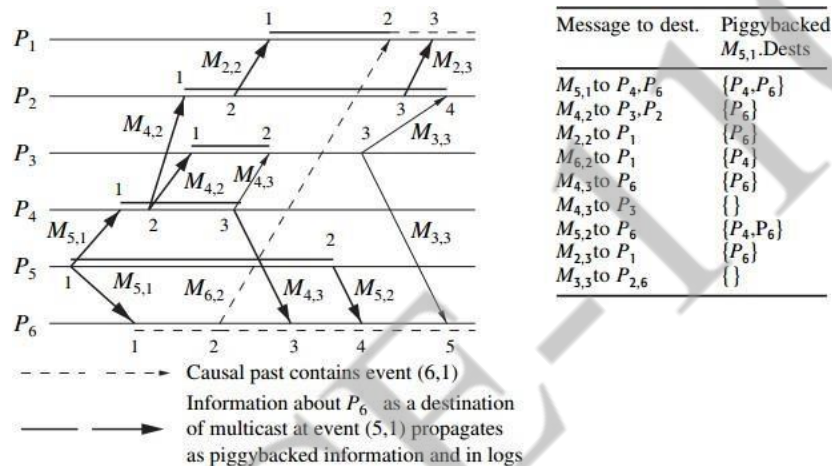


Fig 2.8: Illustration of propagation

constraints Multicasts $M_{5,1}$ and $M_{4,1}$

Message $M_{5,1}$ sent to processes P_4 and P_6 contains the piggybacked information $M_{5,1}$. $Dest = \{P_4, P_6\}$. Additionally, at the send event $(5, 1)$, the information $M_{5,1}.Dests = \{P_4, P_6\}$ is also inserted in the local log Log_5 . When $M_{5,1}$ is delivered to P_6 , the (new) piggybacked information $P_4 \in M_{5,1}.Dests$ is stored in Log_6 as $M_{5,1}.Dests = \{P_4\}$ information about $P_6 \in M_{5,1}.Dests$ which was needed for routing, must not be stored in Log_6 because of constraint I.

In the same way when $M_{5,1}$ is delivered to process P_4

at event $(4, 1)$, only the new piggybacked information $P_6 \in M_{5,1}.Dests$ is inserted in Log_4 as $M_{5,1}.Dests = P_6$ which is later propagated during multicast $M_{4,2}$.

Multicast $M_{4,3}$

At event $(4, 3)$, the information $P_6 \in M_{5,1}.Dests$ in Log_4 is propagated on multicast $M_{4,3}$ only to process P_6 to ensure causal delivery using the Delivery Condition. The piggybacked information on message $M_{4,3}$ sent to process P_3 must not contain this information because of constraint II. As long as any future message sent to P_6 is delivered in causal order w.r.t. $M_{4,3}$ sent to P_6 , it will also be delivered in causal order w.r.t. $M_{5,1}$. And as $M_{5,1}$ is already delivered to P_4 , the information $M_{5,1}.Dests = \emptyset$ is piggybacked on $M_{4,3}$ sent to P_3 . Similarly, the information $P_6 \in M_{5,1}.Dests$ must be deleted from Log_4 as it will no longer be needed, because of constraint II. $M_{5,1}.Dests = \emptyset$ is stored in Log_4 to remember that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to all its destinations.

Learning implicit information at P2 and P3

When message $M_{4,2}$ is received by processes P2 and P3, they insert the (new) piggybacked information in their local logs, as information $M_{5,1}.Dests = P6$. They both continue to store this in Log2 and Log3 and propagate this information on multicasts until they learn at events (2, 4) and (3, 2) on receipt of messages $M_{3,3}$ and $M_{4,3}$, respectively, that any future message is expected to be delivered in causal order to process P6, w.r.t. $M_{5,1}$ sent to P6. Hence by constraint II, this information must be deleted from Log2 and Log3. The flow of events is given by;

- When $M_{4,3}$ with piggybacked information $M_{5,1}.Dests = \emptyset$ is received by P3 at (3, 2), this is inferred to be valid current implicit information about multicast $M_{5,1}$ because the log Log3 already contains explicit information $P6 \in M_{5,1}.Dests$ about that multicast. Therefore, the explicit information in Log3 is inferred to be old and must be deleted to achieve optimality. $M_{5,1}.Dests$ is set to \emptyset in Log3.
- The logic by which P2 learns this implicit knowledge on the arrival of $M_{3,3}$ is identical.

Processing at P6

When message $M_{5,1}$ is delivered to P6, only $M_{5,1}.Dests = P4$ is added to Log6. Further, P6 propagates only $M_{5,1}.Dests = P4$ on message $M_{6,2}$, and this conveys the current implicit information $M_{5,1}$ has been delivered to P6 by its very absence in the explicit information.

- When the information $P6 \in M_{5,1}.Dests$ arrives on $M_{4,3}$, piggybacked as $M_{5,1}.Dests = P6$ it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in Log6 (constraint I) – further, the presence of $M_{5,1}.Dests = P4$ in Log6 implies the implicit information that $M_{5,1}$ has already been delivered to P6. Also, the absence of $P4$ in $M_{5,1}.Dests$ in the explicit piggybacked information implies the implicit information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to P4, and, therefore, $M_{5,1}.Dests$ is set to \emptyset in Log6.
- When the information $P6 \in M_{5,1}.Dests$ arrives on $M_{5,2}$ piggybacked as $M_{5,1}.Dests = \{P4, P6\}$ it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in Log6 because Log6 contains $M_{5,1}.Dests = \emptyset$, which gives the implicit information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to both P4 and P6.

Processing at P1

- When $M_{2,2}$ arrives carrying piggybacked information $M_{5,1}.Dests = P6$ this (new) information is inserted in Log1.
- When $M_{6,2}$ arrives with piggybacked information $M_{5,1}.Dests = \{P4\}$, P1 learns implicit information $M_{5,1}$ has been delivered to P6 by the very absence of explicit information $P6 \in M_{5,1}.Dests$ in the piggybacked information, and hence marks information $P6 \in M_{5,1}.Dests$ for deletion from Log1
- The information “ $P6 \in M_{5,1}.Dests$ piggybacked on $M_{2,3}$, which arrives at P1, is inferred to be outdated using the implicit knowledge derived from $M_{5,1}.Dests = \emptyset$ ” in Log1.

2.8 TOTAL ORDER

For each pair of processes P_i and P_j and for each pair of messages M_x and M_y that are delivered to both the processes, P_i is delivered M_x before M_y if and only if P_j is delivered M_x before M_y .

Centralized Algorithm for total ordering

Each process sends the message it wants to broadcast to a centralized process, which relays all the messages it receives to every other process over FIFO channels.

- (1) When process P_i wants to multicast a message M to group G :
 - (1a) **send** $M(i, G)$ to central coordinator.
- (2) When $M(i, G)$ arrives from P_i at the central coordinator:
 - (2a) **send** $M(i, G)$ to all members of the group G .
- (3) When $M(i, G)$ arrives at P_j from the central coordinator:
 - (3a) **deliver** $M(i, G)$ to the application.

Complexity: Each message transmission takes two message hops and exactly n messages in a system of n processes.

Drawbacks: A centralized algorithm has a single point of failure and congestion, and is not an elegant solution.

Three phase distributed algorithm

Three phases can be seen in both sender and receiver side.

Sender

Phase 1

- In the first phase, a process multicasts the message M with a locally unique tag and the local timestamp to the group members.

Phase 2

- The sender process awaits a reply from all the group members who respond with a tentative proposal for a revised timestamp for that message M .
- The await call is non-blocking.

Phase 3

- The process multicasts the final timestamp to the group.

record Q_entry

M : **int**; // the application message
 tag : **int**; // unique message identifier
 $sender_id$: **int**; // sender of the message
 $timestep$: **int**; // tentative timestamp assigned to message
 $deliverable$: **boolean**; // whether message is ready for delivery

(local variables)

queue of Q_entry : $temp_Q, delivery_Q$

int: $clock$ // Used as a variant of Lamport's scalar clock

int: $priority$ // Used to track the highest proposed timestamp

(message types)

$REVISE_TS(M, i, tag, ts)$

// Phase 1 message sent by P_i , with initial timestamp ts

$PROPOSED_TS(j, i, tag, ts)$

// Phase 2 message sent by P_j , with revised timestamp, to P_i

$FINAL_TS(i, tag, ts)$ // Phase 3 message sent by P_i , with final timestamp

(1) When process P_i wants to multicast a message M with a tag tag :

(1a) $clock \leftarrow clock + 1$;

(1b) **send** $REVISE_TS(M, i, tag, clock)$ to all processes;

(1c) $temp_ts \leftarrow 0$;

(1d) **await** $PROPOSED_TS(j, i, tag, ts_j)$ from each process P_j ;

(1e) $\forall j \in N$, **do** $temp_ts \leftarrow \max(temp_ts, ts_j)$;

(1f) **send** $FINAL_TS(i, tag, temp_ts)$ to all processes;

(1g) $clock \leftarrow \max(clock, temp_ts)$.

Fig) Sender side of three phase distributed algorithm

Receiver Side

Phase 1

- The receiver receives the message with a tentative timestamp. It updates the variable priority that tracks the highest proposed timestamp, then revises the proposed timestamp to the priority, and places the message with its tag and the revised timestamp at the tail of the queue $temp_Q$. In the queue, the entry is marked as undeliverable.

Phase 2

- The receiver sends the revised timestamp back to the sender. The receiver then waits in a non-blocking manner for the final timestamp.

Phase 3

- The final timestamp is received from the multi-caster. The corresponding message entry in $temp_Q$ is identified using the tag, and is marked as deliverable after the revised timestamp is overwritten by the final timestamp.

- The queue is then resorted using the timestamp field of the entries as the key. As the queue is already sorted except for the modified entry for the message under consideration, that message entry has to be placed in its sorted position in the queue.
- If the message entry is at the head of the temp_Q, that entry, and all consecutive subsequent entries that are also marked as deliverable, are dequeued from temp_Q, and enqueued in deliver_Q.

Complexity

This algorithm uses three phases, and, to send a message to $n - 1$ processes, it uses $3(n - 1)$ messages and incurs a delay of three message hops

Example

An example execution to illustrate the algorithm is given in Figure 6.14. Here, A and B multicast to a set of destinations and C and D are the common destinations for both multicasts. •

Figure (a) The main sequence of steps is as follows:

1. A sends a REVISE_TS(7) message, having timestamp 7. B sends a REVISE_TS(9) message, having timestamp 9.
2. C receives A's REVISE_TS(7), enters the corresponding message in temp_Q, and marks it as undeliverable; priority = 7. C then sends PROPOSED_TS(7) message to A
3. D receives B's REVISE_TS(9), enters the corresponding message in temp_Q, and marks it as undeliverable; priority = 9. D then sends PROPOSED_TS(9) message to B.
4. C receives B's REVISE_TS(9), enters the corresponding message in temp_Q, and marks it as undeliverable; priority = 9. C then sends PROPOSED_TS(9) message to B.
5. D receives A's REVISE_TS(7), enters the corresponding message in temp_Q, and marks it as undeliverable; priority = 10. D assigns a tentative timestamp value of 10, which is greater than all of the times tamps on REVISE_TSs seen so far, and then sends PROPOSED_TS(10) message to A.

The state of the system is as shown in the figure

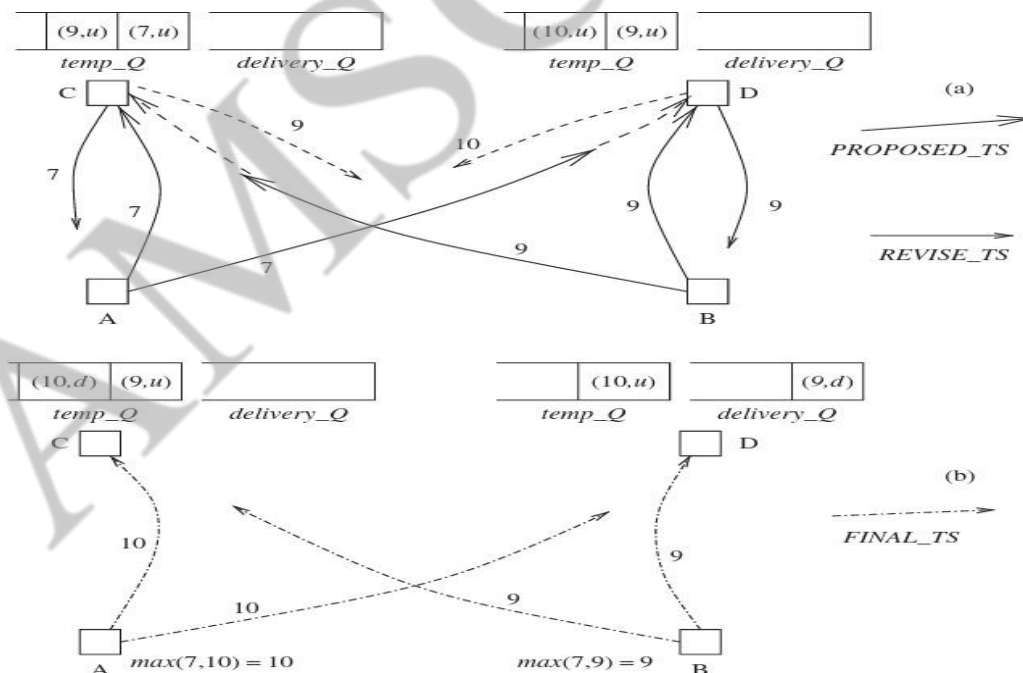


Fig) An example to illustrate the three-phase total ordering algorithm. (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages in the example.

Figure (b) The continuing sequence of main steps is as follows:

6. When A receives PROPOSED_TS(7) from C and PROPOSED_TS(10) from D, it computes the final timestamp as $\max(7, 10) = 10$, and sends FINAL_TS(10) to C and D.

7. When B receives PROPOSED_TS(9) from C and PROPOSED_TS(9) from D, it computes the final timestamp as $\max(9, 9) = 9$, and sends FINAL_TS(9) to C and D.

8. C receives FINAL_TS(10) from A, updates the corresponding entry in temp_Q with the timestamp, resorts the queue, and marks the message as deliverable. As the message is not at the head of the queue, and some entry ahead of it is still undeliverable, the message is not moved to delivery_Q.

9. D receives FINAL_TS(9) from B, updates the corresponding entry in temp_Q by marking the corresponding message as deliverable, and resorts the queue. As the message is at the head of the queue, it is moved to delivery_Q. This is the system snapshot shown in Figure (b).

The following further steps will occur:

10. When C receives FINAL_TS(9) from B, it will update the corresponding entry in temp_Q by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the delivery_Q, and the next message (of A), which is also deliverable, is also moved to the delivery_Q.

11. When D receives FINAL_TS(10) from A, it will update the corresponding entry in temp_Q by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the delivery_Q.

2.9 GLOBAL STATE AND SNAPSHOT RECORDING ALGORITHMS

- A distributed computing system consists of processes that do not share a common memory and communicate asynchronously with each other by message passing.
- Each component has a local state. The state of the process is the local memory and a history of its activity.
- The state of a channel is characterized by the set of messages sent along the channel less the messages received along the channel. The global state of a distributed system is a collection of the local states of its components.
- If shared memory were available, an up-to-date state of the entire system would be available to the processes sharing the memory.
- The absence of shared memory necessitates ways of getting a coherent and complete view of the system based on the local states of individual processes.
- A meaningful global snapshot can be obtained if the components of the distributed system record their local states at the same time.
- This would be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that could be instantaneously read by the processes.
- If processes read time from a single common clock, various indeterminate transmission delays during the read operation will cause the processes to identify various physical instants as the same time.

2.9.1 System Model

- The system consists of a collection of n processes, p_1, p_2, \dots, p_n that are connected by channels.
- Let C_{ij} denote the channel from process p_i to process p_j .
- Processes and channels have states associated with them.

- The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc., and may be highly dependent on the local context of the distributed application.
 - The state of channel C_{ij} , denoted by SC_{ij} , is given by the set of messages in transit in the channel.
 - The events that may happen are: internal event, send ($send(m_{ij})$) and receive ($rec(m_{ij})$) events.
 - The occurrences of events cause changes in the process state.
 - A **channel** is a distributed entity and its state depends on the local states of the processes on which it is incident.
- Transit:** $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$
- The transit function records the state of the channel C_{ij} .
 - In the FIFO model, each channel acts as a first-in first-out message queue and, thus, message ordering is preserved by a channel.
 - In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

2.9.2 A consistent global state

The global state of a distributed system is a collection of the local states of the processes and the channels. The global state is given by:

$$GS = \{\cup_i LS_i, \cup_{i,j} SC_{ij}\}.$$

The two conditions for global state are:

C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$

C2: $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j.$

Condition 1 preserves **law of conservation of messages**. Condition C2 states that in the collected global state, for every effect, its cause must be present.

Law of conservation of messages: Every message m_{ij} that is recorded as sent in the local state of a process p_i must be captured in the state of the channel C_{ij} or in the collected local state of the receiver process p_j .

- In a consistent global state, every message that is recorded as received is also recorded as sent. Such a global state captures the notion of causality that a message cannot be received if it was not sent.
- Consistent global states are meaningful global states and inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

2.9.3 Interpretation of cuts

- Cuts in a space–time diagram provide a powerful graphical aid in representing and reasoning about the global states of a computation. A cut is a line joining an arbitrary point on each process line that slices the space–time diagram into a PAST and a FUTURE.

- A consistent global state corresponds to a cut in which every message received in the PAST of the cut has been sent in the PAST of that cut. Such a cut is known as a consistent cut.
- In a consistent snapshot, all the recorded local states of processes are concurrent; that is, the recorded local state of no process casually affects the recorded local state of any other process.

Issues in recording global state

The non-availability of global clock in distributed system, raises the following issues:

Issue 1:

How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?

Answer:

- Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from C1).
- Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from C2).

Issue 2:

How to determine the instant when a process takes its snapshot?

The answer

Answer:

A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot

2.9.4 SNAPSHOT ALGORITHMS FOR FIFO CHANNELS

Each distributed application has number of processes running on different physical servers. These processes communicate with each other through messaging channels.

A snapshot captures the local states of each process along with the state of each communication channel.

Snapshots are required to:

- Checkpointing
- Collecting garbage
- Detecting deadlocks
- Debugging

Chandy-Lamport algorithm

- The algorithm will record a global snapshot for each process channel.
- The Chandy-Lamport algorithm uses a control message, called a marker.
- After a site has recorded its snapshot, it sends a marker along all of its outgoing channels before sending out any more messages.
- Since channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- This addresses issue I1. The role of markers in a FIFO system is to act as delimiters for the messages in the channels so that the channel state recorded by the process

at the receiving end of the channel satisfies the condition C2.

Marker sending rule for process p_i

- (1) Process p_i records its state.
- (2) For each outgoing channel C on which a marker has not been sent, p_i sends a marker along C before p_i sends further messages along C .

Marker receiving rule for process p_j

On receiving a marker along channel C :

- if** p_j has not recorded its state **then**
 Record the state of C as the empty set
 Execute the “marker sending rule”
else
 Record the state of C as the set of messages received along C after p_j 's state was recorded and before p_j received the marker along C

Fig 2.10: Chandy–Lamport algorithm

Initiating a snapshot

- Process P_i initiates the snapshot
- P_i records its own state and prepares a special marker message.
- Send the marker message to all other processes.
- Start recording all incoming messages from channels C_{ij} for j not equal to i .

Propagating a snapshot

- For all processes P_j consider a message on channel C_{kj} .
- If marker message is seen for the first time:
 - P_j records own state and marks C_{kj} as empty
 - Send the marker message to all other processes.
 - Record all incoming messages from channels C_{lj} for l not equal to j or k .
 - Else add all messages from inbound channels.

Terminating a snapshot

- All processes have received a marker.
- All process have received a marker on all the $N-1$ incoming channels.
- A central server can gather the partial state to build a global snapshot.

Correctness of the algorithm

- Since a process records its snapshot when it receives the first marker on any incoming channel, no messages that follow markers on the channels incoming to it are recorded in the process's snapshot.
- A process stops recording the state of an incoming channel when a marker is received on that channel.

- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition C2 is satisfied.
- When a process p_j receives message m_{ij} that precedes the marker on channel C_{ij} , it acts as follows: if process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij} . Thus, condition C1 is satisfied.

Complexity

The recording part of a single instance of the algorithm requires $O(e)$ messages and $O(d)$ time, where e is the number of edges in the network and d is the diameter of the network.

Properties of the recorded global state

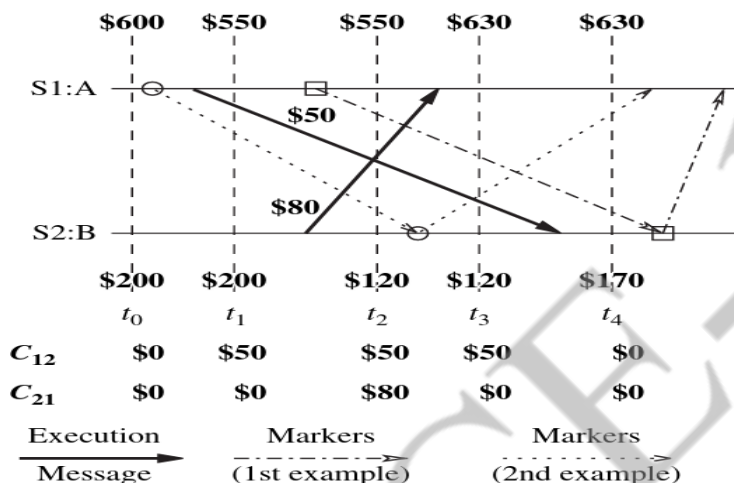


Fig) Timing diagram of two possible executions of the banking examples

1. (Markers shown using dashed-and-dotted arrows.) Let site S1 initiate the algorithm just after t_1 . Site S1 records its local state (account A=\$550) and sends a marker to site S2. The marker is received by site S2 after t_4 . When site S2 receives the marker, it records its local state (account B=\$170), the state of channel C_{12} as \$0, and sends a marker along channel C_{21} . When site S1 receives this marker, it records the state of channel C_{21} as \$80. The \$800 amount in the system is conserved in the recorded global state,
 $A=\$550$ $B=\$170$ $C_{12}=\$0$ $C_{21}=\$80$
2. (Markers shown using dotted arrows.) Let site S1 initiate the algorithm just after t_0 and before sending the \$50 for S2. Site S1 records its local state (account A = \$600) and sends a marker to S2. The marker is received by site S2 between t_2 and t_3 . When site S2 receives the marker, it records its local state (account B = \$120), the state of channel C_{12} as \$0, and sends a marker along channel C_{21} . When site S1 receives this marker, it records the state of channel C_{21} as \$80. The \$800 amount in the system is conserved in the recorded global state,
 $A=\$600$ $B=\$120$ $C_{12}=\$0$ $C_{21}=\$80$

The recorded global state may not correspond to any of the global states that occurred during the computation.

This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.

But the system could have passed through the recorded global states in some equivalent executions.

The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.

Therefore, a recorded global state is useful in detecting stable properties.

UNIT III

DISTRIBUTED MUTEX AND DEADLOCK

DISTRIBUTED MUTEX & DEADLOCK

Distributed mutual exclusion algorithms: Introduction – Preliminaries – Lamport's algorithm – Ricart-Agrawala algorithm – Token-Based algorithms – Suzuki Kasami's broadcast algorithm; Deadlock detection in distributed systems: Introduction – System model – Preliminaries – Models of deadlocks – Chandy-Misra-Haas Algorithms for the AND model and OR model.

3.1 DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

- Mutual exclusion is a concurrency control property which is introduced to prevent race conditions.
- It is the requirement that a process cannot access a shared resource while another concurrent process is currently present or executing the same resource.

Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time.

- Message passing is the sole means for implementing distributed mutual exclusion. There are three basic approaches for implementing distributed mutual exclusion:

1. Token-based approach:

- A unique token (also known as the privilege message) is shared among the sites.
- A site is allowed to enter its CS if it possesses the token.
- Mutual Exclusion is ensured because the token is unique.
- Eg: Suzuki-Kasami's Broadcast Algorithm, Raymond's Tree- Based Algorithm etc

2. Non-token-based approach:

- Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.
- Eg: Lamport's algorithm, Ricart-Agrawala algorithm

Quorum-based approach:

- Each site requests permission to execute the CS from a subset of sites (called a quorum)
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to make sure that only one request executes the

CS at any time.

- Eg: Maekawa's Algorithm

3.1.1 Preliminaries

- The system consists of N sites, $S_1, S_2, S_3, \dots, S_N$.
- Assume that a single process is running on each site.
- The process at site S_i is denoted by p_i .
- All these processes communicate asynchronously over an underlying communication network.
- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS.
- In the requesting the CS state, the site is blocked and cannot make further requests for the CS.
- In the idle state, the site is executing outside the CS.
- In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such state is referred to as the idle token state.
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.
- N denotes the number of processes or sites involved in invoking the critical section, T denotes the average message delay, and E denotes the average critical section execution time.

3.1.2 Requirements of mutual exclusion algorithms

- **Safety property:**

At any instant, only one process can execute the critical section. This is an essential property of a mutual exclusion algorithm.

- **Liveness property:**

This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages that will never arrive. This is an important property of a mutual exclusion algorithm

- **Fairness:**

Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS. In mutual exclusion algorithms, the fairness property generally means that the CS execution requests are executed in order of their arrival in the system.

3.1.3 Performance metrics

- **Message complexity:** This is the number of messages that are required per CS execution by a site.
- **Synchronization delay:** After a site leaves the CS, it is the time required and before the next site enters the CS.
- **Response time:** This is the time interval a request waits for its CS execution to be over after its request messages have been sent out. Thus, response time does not include the time a request waits at a site before its request messages have been sent out.

System throughput: This is the rate at which the system executes requests for the

$$\text{System throughput} = \frac{1}{(SD + E)}$$

CS. If SD is the synchronization delay and E is the average critical section execution time.

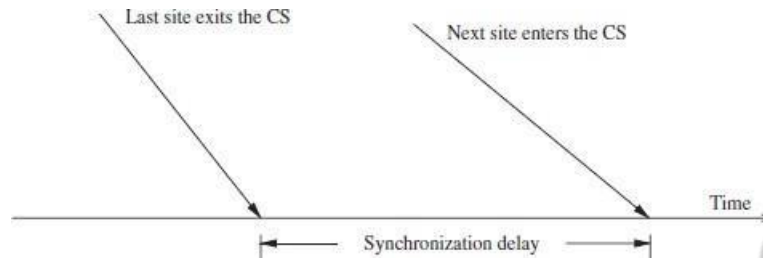


Figure: Synchronization delay

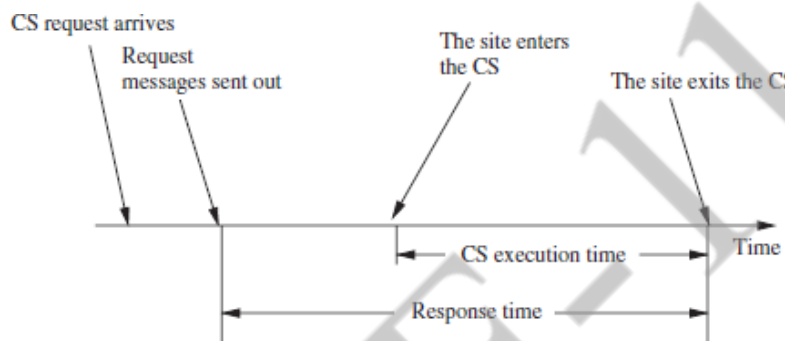


Figure: Response Time

Low and High Load Performance:

- The performance of mutual exclusion algorithms is classified as two special loading conditions, viz., “low load” and “high load”.
- The load is determined by the arrival rate of CS execution requests.
- Under *low load* conditions, there is seldom more than one request for the critical section present in the system simultaneously.
- Under *heavy load* conditions, there is always a pending request for critical section at a site.

Best and worst case performance

- In the best case, prevailing conditions are such that a performance metric attains the best possible value. For example, the best value of the response time is a roundtrip message delay plus the CS execution time, $2T + E$.
- For examples, the best and worst values of the response time are achieved when load is, respectively, low and high;
- The best and the worse message traffic is generated at low and heavy load conditions, respectively.

3.2 LAMPORT'S ALGORITHM

- Request for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site S_i keeps a queue, $request_queue_i$ which contains mutual exclusion requests ordered by their timestamps
- This algorithm requires communication channels to deliver messages the FIFO order. Three types of messages are used Request, Reply and Release. These messages with timestamps also updates logical clock

Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a $REQUEST(ts_i, i)$ message to all other sites and places the request on $request_queue_i$. ((ts_i, i) denotes the timestamp of the request.)
- When a site S_j receives the $REQUEST(ts_i, i)$ message from site S_i , it places site S_i 's request on $request_queue_j$ and returns a timestamped $REPLY$ message to S_i .

Executing the critical section

Site S_i enters the CS when the following two conditions hold:

L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

L2: S_i 's request is at the top of $request_queue_i$.

Releasing the critical section

- Site S_i , upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped $RELEASE$ message to all other sites.
- When a site S_j receives a $RELEASE$ message from site S_i , it removes S_i 's request from its request queue.

Fig: Lamport's distributed mutual exclusion algorithm

To enter Critical section:

- When a site S_i wants to enter the critical section, it sends a request message $Request(ts_i, i)$ to all other sites and places the request on $request_queue_i$. Here, T_{si} denotes the timestamp of Site S_i .
- When a site S_j receives the request message $REQUEST(ts_i, i)$ from site S_i , it returns a timestamped $REPLY$ message to site S_i and places the request of site S_i on $request_queue_j$

To execute the critical section:

- A site S_i can enter the critical section if it has received the message with timestamp larger than (ts_i, i) from all other sites and its own request is at the top of $request_queue_i$.

To release the critical section:

When a site S_i exits the critical section, it removes its own request from the top of its request queue and sends a timestamped $RELEASE$ message to all other sites. When a site S_j receives the timestamped $RELEASE$ message from site S_i , it removes the request of S_i from its request queue.

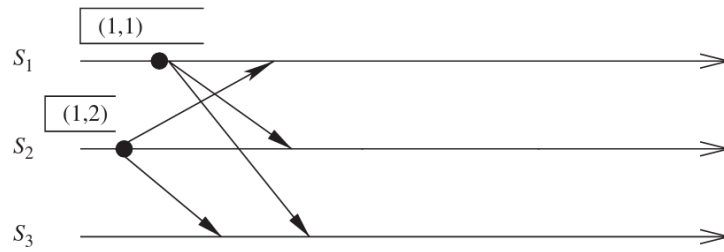


Fig) S1 and S2 are making requests for the CS

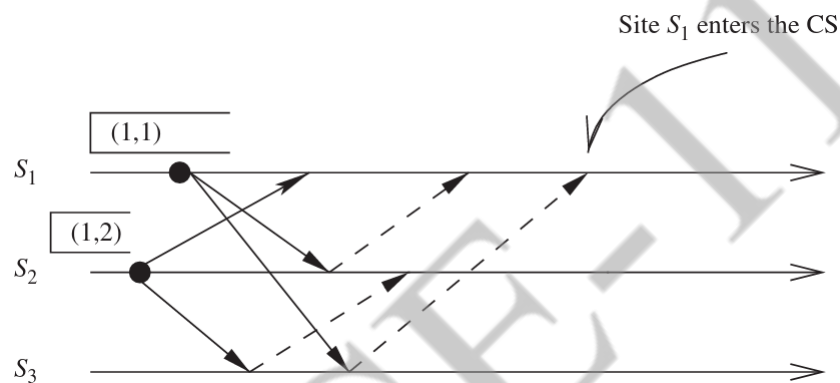


Fig) Site S1 enters the CS

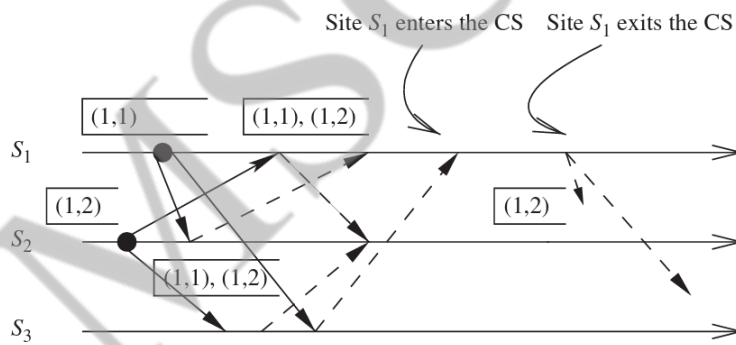


Fig) Site S1 exists the CS and sends RELEASE messages

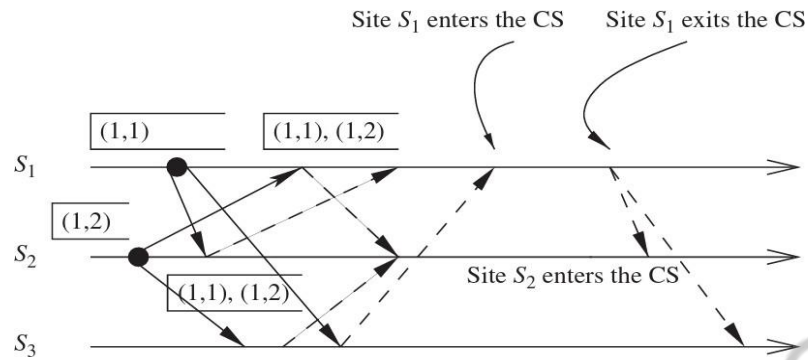


Fig) Site S_2 enters the CS

Correctness

Theorem: Lamport's algorithm achieves mutual exclusion.

Proof: Proof is by contradiction.

- Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites concurrently.
- This implies that at some instant in time, say t , both S_i and S_j have their own requests at the top of their request queues and condition L1 holds at them. Without loss of generality, assume that S_i 's request has smaller timestamp than the request of S_j .
- From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of S_i must be present in request queue j when S_j was executing its CS. This implies that S_j 's own request is at the top of its own request queue when a smaller timestamp request, S_i 's request, is present in the request queue j – a contradiction!

Theorem: Lamport's algorithm is fair.

Proof: The proof is by contradiction.

- Suppose a site S_i 's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i .
- For S_j to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t , S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But request queue at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So S_i 's request must be placed ahead of the S_j 's request in the request queue j . This is a contradiction!

Message Complexity:

Lamport's Algorithm requires invocation of $3(N - 1)$ messages per critical section execution.

These $3(N - 1)$ messages involves

- $(N - 1)$ request messages
- $(N - 1)$ reply messages
- $(N - 1)$ release messages

Drawbacks of Lamport's Algorithm:

- **Unreliable approach:** failure of any one of the processes will halt the progress of entire system.
- **High message complexity:** Algorithm requires $3(N-1)$ messages per critical section invocation.

To enter Critical section:

- When a site S_i wants to enter the critical section, it sends a timestamped REQUEST message to all other sites.
- When a site S_j receives a REQUEST message from site S_i , it sends a REPLY message to site S_i if and only if Site S_j is neither requesting nor currently executing the critical section.
- In case Site S_j is requesting, the timestamp of Site S_i 's request is smaller than its own request.
- Otherwise the request is deferred by site S_j .

To execute the critical section:

Site S_i enters the critical section if it has received the REPLY message from all other sites.

To release the critical section:

Upon exiting site S_i sends REPLY message to all the deferred requests.

Performance:

Synchronization delay is equal to maximum message transmission time. It requires $3(N - 1)$ messages per CS execution. Algorithm can be optimized to $2(N - 1)$ messages by omitting the REPLY message in some situations.

3.3 RICART-AGRAWALA ALGORITHM

- Ricart-Agrawala algorithm is an algorithm for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala.
- This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm.
- It follows permission based approach to ensure mutual exclusion.
- Two types of messages (REQUEST and REPLY) are used and communication channels are assumed to follow FIFO order.
- A site sends a REQUEST message to all other sites to get their permission to enter critical section.
- A site sends a REPLY message to other sites to give its permission to enter the

criticalsection.

- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests.
- Smaller timestamp gets high priority over larger timestamp.
- The execution of critical section request is always in the order of their timestamp.

Requesting the critical section

- (a) When a site S_i wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.
- (b) 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 the site S_j is requesting and S_i 's request's timestamp is smaller than site S_j 's own request's timestamp. Otherwise, the reply is deferred and S_j sets $RD_j[i] := 1$.

Executing the critical section

- (c) Site S_i enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

Releasing the critical section

- (d) When site S_i exits the CS, it sends all the deferred REPLY messages: $\forall j$ if $RD_i[j] = 1$, then sends a REPLY message to S_j and sets $RD_i[j] := 0$.

Fig: Ricart-Agrawala algorithm

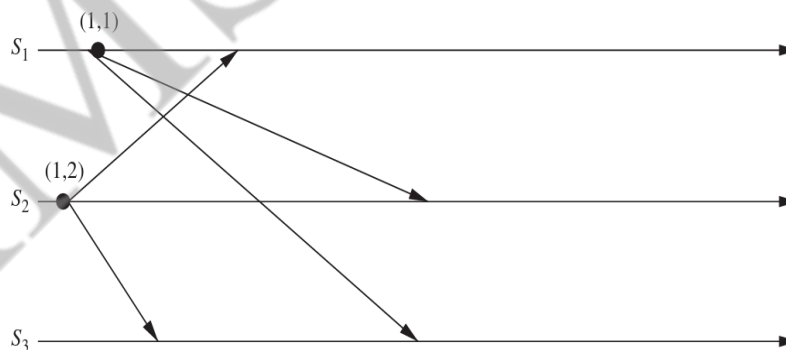


Fig) site S1 and S2 each make a request for the CS

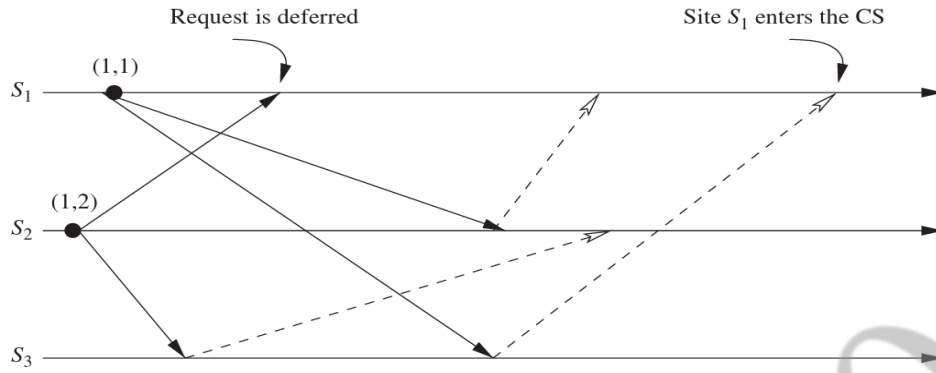


Fig) site S_1 enters the CS

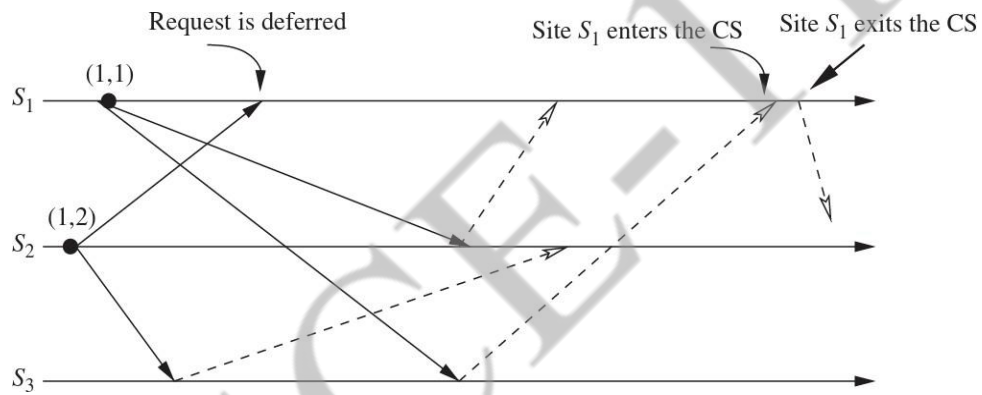


Fig) Site S_1 exits the CS and sends a reply message to S_2 's deferred request

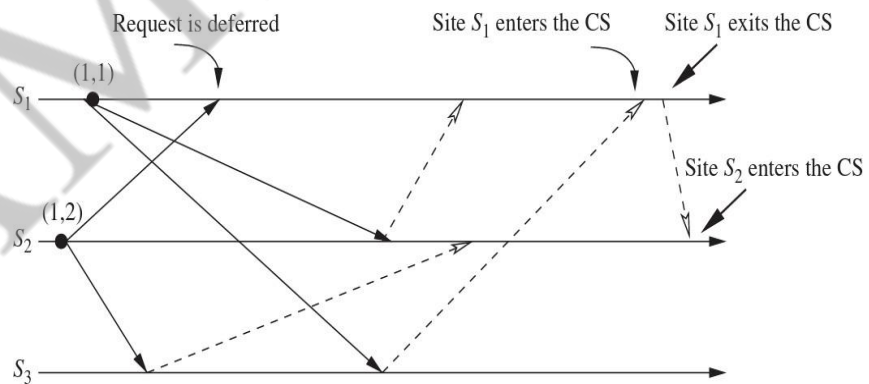


Fig) Site S_2 enters the CS

Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.

Proof: Proof is by contradiction.

- Suppose two sites S_i and S_j are executing the CS concurrently and S_i 's request has higher priority than the request of S_j . Clearly, S_i received S_j 's request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j 's request) before S_i exits the CS.
- However, this is impossible because S_j 's request has lower priority. Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

Message Complexity:

Ricart-Agrawala algorithm requires invocation of $2(N - 1)$ messages per critical section execution. These $2(N - 1)$ messages involve:

- $(N - 1)$ request messages
- $(N - 1)$ reply messages

Drawbacks of Ricart-Agrawala algorithm:

- **Unreliable approach:** failure of any one of node in the system can halt the progress of the system. In this situation, the process will starve forever. The problem of failure of node can be solved by detecting failure after some timeout.

Performance:

Synchronization delay is equal to maximum message transmission time. It requires $2(N - 1)$ messages per Critical section execution.

3.4 SUZUKI-KASAMI'S BROADCAST ALGORITHM

- Suzuki-Kasami algorithm is a token-based algorithm for achieving mutual exclusion in distributed systems.
- This is modification of Ricart-Agrawala algorithm, a permission based (Non-token based) algorithm which uses REQUEST and REPLY messages to ensure mutual exclusion.
- In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token.
- Non-token based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.
- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests

To enter Critical section:

- When a site S_j wants to enter the critical section and it does not have the token then it increments its sequence number $RN_i[i]$ and sends a request message REQUEST(i, sn) to all other sites in order to request the token.
- Here sn is update value of $RN_i[i]$
- When a site S_j receives the request message REQUEST(i, sn) from site S_i , it

sets $RN_j[i]$ to maximum of $RN_j[i]$ and sn . $eRN_j[i] = \max(RN_j[i], sn)$.

After updating $RN_j[i]$, Site S_j sends the token to site S_i if it has token and $RN_j[i] = LN[i] + 1$

Requesting the critical section:

- (a) If requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(i, sn) message to all other sites. ("sn" is the updated value of $RN_i[i]$.)
- (b) 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 $RN_j[i] = LN[i] + 1$.

Executing the critical section:

- (c) Site S_i executes the CS after it has received the token.

Releasing the critical section: Having finished the execution of the CS, site S_i takes the following actions:

- (d) It sets $LN[i]$ element of the token array equal to $RN_i[i]$.
- (e) For every site S_j whose i.d. is not in the token queue, it appends its i.d. to the token queue if $RN_j[j] = LN[j] + 1$.
- (f) If the token queue is nonempty after the above update, S_i deletes the top site i.d. from the token queue and sends the token to the site indicated by the i.d.

Fig: Suzuki-Kasami's broadcast

algorithm To execute the critical section:

- Site S_i executes the critical section if it has acquired the token.

To release the critical section:

After finishing the execution Site S_i exits the critical section and does following:

- sets $LN[i] = RN_i[i]$ to indicate that its critical section request $RN_i[i]$ has been executed

- For every site S_j , whose ID is not present in the token queue Q , it appends its ID to Q if $RN_j[j] = LN[j] + 1$ to indicate that site S_j has an outstanding request.
- After above updation, if the Queue Q is non-empty, it pops a site ID from the Q and sends the token to site indicated by popped ID.
- If the queue Q is empty, it keeps the token

Correctness

Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

Theorem: A requesting site enters the CS in finite time.

Proof: Token request messages of a site S_i reach other sites in finite time.

Since one of these sites will have token in finite time, site S_i 's request will be placed in the token queue in finite time.

Since there can be at most $N - 1$ requests in front of this request in the token queue, site S_i will get the token and execute the CS in finite time.

Message Complexity:

The algorithm requires 0 message invocation if the site already holds the idle token at the time of critical section request or maximum of N message per critical section execution.

This N messages involves

- $(N - 1)$ request messages
- 1 reply message

Drawbacks of Suzuki–Kasami Algorithm:

- Non-symmetric Algorithm: A site retains the token even if it does not have requested for critical section.

Performance:

Synchronization delay is 0 and no message is needed if the site holds the idle token at the time of its request. In case site does not hold the idle token, the maximum synchronization delay is equal to maximum message transmission time and a maximum of N message is required per critical section invocation.

3.5 DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS

Deadlock can neither be prevented nor avoided in distributed system as the system is so vast that it is impossible to do so. Therefore, only deadlock detection can be implemented. The techniques of deadlock detection in the distributed system require the following:

- **Progress:** The method should be able to detect all the deadlocks in the system.
- **Safety:** The method should not detect false or phantom deadlocks.

There are three approaches to detect deadlocks in distributed systems.

Centralized approach:

- Here there is only one responsible resource to detect deadlock.
- The advantage of this approach is that it is simple and easy to implement, while the

drawbacks include excessive workload at one node, single point failure which in turn makes the system less reliable.

Distributed approach:

- In the distributed approach different nodes work together to detect deadlocks. Nosingle point failure as workload is equally divided among all nodes.
- The speed of deadlock detection also increases.

Hierarchical approach:

- This approach is the most advantageous approach.
- It is the combination of both centralized and distributed approaches of deadlock detection in a distributed system.
- In this approach, some selected nodes or cluster of nodes are responsible for deadlock detection and these selected nodes are controlled by a single node.

System Model

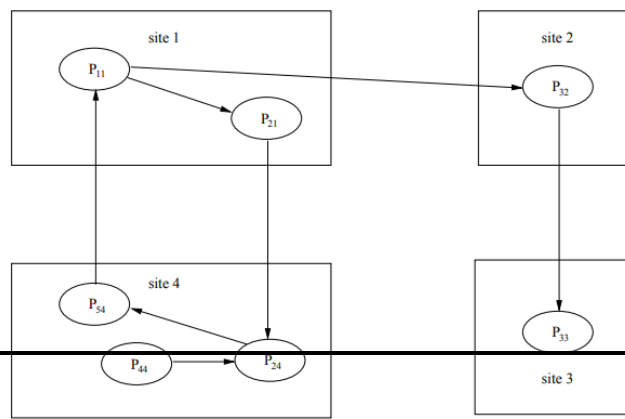
- A distributed program is composed of a set of n asynchronous processes $p_1, p_2, \dots, p_i, \dots, p_n$ that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.
- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.

We make the following assumptions:

- The systems have only reusable resources.
- Processes are allowed to make only exclusive access to resources.
- There is only one copy of each resource.
- A process can be in two states: running or blocked.
- In the running state (also called active state), a process has all the needed resources and is either executing or is ready for execution.
- In the blocked state, a process is waiting to acquire some resource.

Wait for graph

This is used for deadlock deduction. A graph is drawn based on the request and



acquisition of the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

Preliminaries

3.6.1 Deadlock Handling Strategies

Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay. There are three strategies for handling deadlocks:

- **Deadlock prevention:**
 - This is achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource.
 - This approach is highly inefficient and impractical in distributed systems.
- **Deadlock avoidance:**
 - A resource is granted to a process if the resulting global system state is safe. This is impractical in distributed systems.
- **Deadlock detection:**
 - This requires examination of the status of process-resource interactions for presence of cyclic wait.
 - Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.

3.6.2 Issues in deadlock Detection

Deadlock handling faces two major issues

1. Detection of existing deadlocks

Resolution of detected deadlocks

Deadlock Detection

- Detection of deadlocks involves addressing two issues namely maintenance of the WFG and searching of the WFG for the presence of cycles or **knots**.
- In distributed systems, a cycle or knot may involve several sites, the search for cycles greatly depends upon how the WFG of the system is represented across the system.
- Depending upon the way WFG information is maintained and the search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems.

Correctness criteria

A deadlock detection algorithm must satisfy the following two conditions:

1. Progress-No undetected deadlocks:

The algorithm must detect all existing deadlocks in finite time. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any

more events to occur to detect the deadlock.

2. Safety -No false deadlocks:

The algorithm should not report deadlocks which do not exist. This is also called as called **phantom or false deadlocks**

Resolution of a Detected Deadlock

- Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock.
- It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution.
- The deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph.
- When a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system.
- If this information is not cleaned in a timely manner, it may result in detection of phantom deadlocks.

3.7 MODELS OF DEADLOCKS

The models of deadlocks are explained based on their hierarchy. The diagrams illustrate the working of the deadlock models. P_a , P_b , P_c , P_d are passive processes that had already acquired the resources. P_e is active process that is requesting the resource.

3.7.1 Single Resource Model

- A process can have at most one outstanding request for only one unit of a resource.
- The maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.

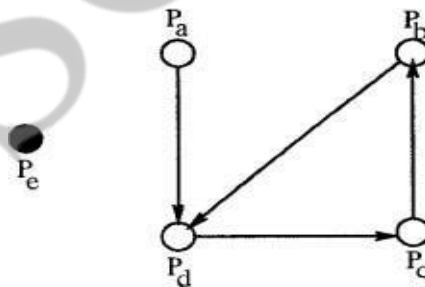


Fig: Deadlock in single resource model

3.7.2 AND Model

- In the AND model, a passive process becomes active (i.e., its activation condition is fulfilled) only after a message from each process in its dependent set has arrived.
- In the AND model, a process can request more than one resource simultaneously and therequest is satisfied only after all the requested resources are granted to the process.
- The requested resources may exist at different locations.

- The out degree of a node in the WFG for AND model can be more than 1.
- The presence of a cycle in the WFG indicates a deadlock in the AND model.
- Each node of the WFG in such a model is called an AND node.
- In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not viceversa. That is, a process may not be a part of a cycle, it can still be deadlocked.

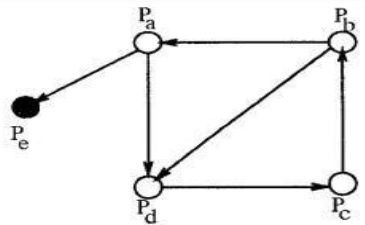


Fig: Deadlock in AND model

3.7.3 OR Model

- A process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
- Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
- In the OR model, the presence of a knot indicates a deadlock.

Deadlock in OR model: a process P_i is blocked if it has a pending OR request to be satisfied.

- With every blocked process, there is an associated set of processes called **dependent set**.
- A process shall move from an idle to an active state on receiving a grant message from any of the processes in its dependent set.
- A process is permanently blocked if it never receives a grant message from any of the processes in its dependent set.
- A set of processes S is deadlocked if all the processes in S are permanently blocked.
- In short, a process is deadlocked or permanently blocked, if the following conditions are met:
 1. Each of the process in the set S is blocked.
 2. The dependent set for each process in S is a subset of S .
 3. No grant message is in transit between any two processes in set S .
- A blocked process P in the set S becomes active only after receiving a grant message from a process in its dependent set, which is a subset of S .

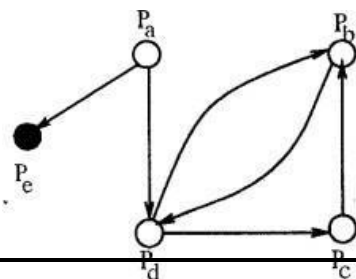


Fig: OR Model

3.7.4 $\binom{p}{q}$ Model (p out of q model)

- This is a variation of AND-OR model.
- This allows a request to obtain any k available resources from a pool of n resources. Both the models are the same in expressive power.
- This favours more compact formation of a request.
- Every request in this model can be expressed in the AND-OR model and vice-versa.
- Note that AND requests for p resources can be stated as $\binom{p}{q}$ and OR requests for p resources can be stated as $\binom{p}{1}$.

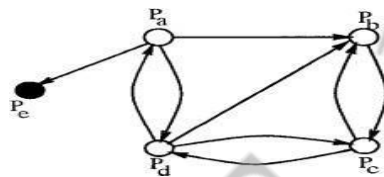


Fig: p out of q Model

3.7.5 Unrestricted model

- No assumptions are made regarding the underlying structure of resource requests.
- In this model, only one assumption that the deadlock is stable is made and hence it is the most general model.
- This model helps separate concerns: Concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication).

3.8 CHANDY–MISRA–HAAS ALGORITHM FOR THE AND MODEL

This is considered an edge-chasing, probe-based algorithm.

It is also considered one of the best deadlock detection algorithms for distributed systems.

If a process makes a request for a resource which fails or times out, the process generates a probe message and sends it to each of the processes holding one or more of its requested resources.

This algorithm uses a special message called probe, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process P_i and it is being sent by the home site of process P_j to the home site of process P_k.

Each probe message contains the following information:

- the id of the process that is blocked (the one that initiates the probe message);
- the id of the process is sending this particular version of the probe message;
- the id of the process that should receive this probe message.

A probe message travels along the edges of the global WFG graph, and a deadlock is

detected when a probe message returns to the process that initiated it.

A process P_j is said to be dependent on another process P_k if there exists a sequence of processes $P_j, P_{i1}, P_{i2}, \dots, P_{im}, P_k$ such that each process except P_k in the sequence is blocked and each process, except the P_j , holds a resource for which the previous process in the sequence is waiting.

Process P_j is said to be locally dependent upon process P_k if P_j is dependent upon P_k and both the processes are on the same site.

When a process receives a probe message, it checks to see if it is also waiting for resources

If not, it is currently using the needed resource and will eventually finish and release the resource.

If it is waiting for resources, it passes on the probe message to all processes it knows to be holding resources it has itself requested.

The process first modifies the probe message, changing the sender and receiver ids. If a process receives a probe message that it recognizes as having initiated, it knows there is a cycle in the system and thus, deadlock.

Data structures

Each process P_i maintains a boolean array, $dependent_i$, where $dependent_i(j)$ is true only if P_i knows that P_j is dependent on it. Initially, $dependent_i(j)$ is false for all i and j .

```
if  $P_i$  is locally dependent on itself
then declare a deadlock
else for all  $P_j$  and  $P_k$  such that
  (a)  $P_i$  is locally dependent upon  $P_j$ , and
  (b)  $P_j$  is waiting on  $P_k$ , and
  (c)  $P_j$  and  $P_k$  are on different sites,
  send a probe ( $i, j, k$ ) to the home site of  $P_k$ 

On the receipt of a probe ( $i, j, k$ ), the site takes
the following actions:
if
  (d)  $P_k$  is blocked, and
  (e)  $dependent_k(i)$  is false, and
  (f)  $P_k$  has not replied to all requests  $P_j$ ,
  then
  begin
     $dependent_k(i) = true$ ;
    if  $k = i$ 
      then declare that  $P_i$  is deadlocked
    else for all  $P_m$  and  $P_n$  such that
      (a')  $P_k$  is locally dependent upon  $P_m$ , and
      (b')  $P_m$  is waiting on  $P_n$ , and
      (c')  $P_m$  and  $P_n$  are on different sites,
      send a probe ( $i, m, n$ ) to the home site of  $P_n$ 
  end.
```

Fig : Chandy–Misra–Haas algorithm for the AND model

Performance analysis

In the algorithm, one probe message is sent on every edge of the WFG which connects processes on two sites.

The algorithm exchanges at most $m(n - 1)/2$ messages to detect a deadlock that involves m processes and spans over n sites.

The size of messages is fixed and is very small (only three integer words).

The delay in detecting a deadlock is $O(n)$.

Advantages:

It is easy to implement.

Each probe message is of fixed length. There is very little computation.

There is very little overhead.

There is no need to construct a graph, nor to pass graph information to other sites.

This algorithm does not find false (phantom) deadlock.

There is no need for special data structures.

3.9 CHANDY MISRA HAAS ALGORITHM FOR THE OR MODEL

A blocked process determines if it is deadlocked by initiating a diffusion computation. Two types of messages are used in a diffusion computation:

- query(i, j, k)
- reply(i, j, k)

denoting that they belong to a diffusion computation initiated by a process p_i and are being sent from process p_j to process p_k .

A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set.

If an active process receives a query or reply message, it discards it. When a blocked process P_k receives a query(i, j, k) message, it takes the following actions:

1. If this is the first query message received by P_k for the deadlock detection initiated by P_i , then it propagates the query to all the processes in its dependent set and sets a local variable $num_k(i)$ to the number of query messages sent.
2. If this is not the engaging query, then P_k returns a reply message to it immediately provided P_k has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.
 - Process P_k maintains a boolean variable $wait_k(i)$ that denotes the fact that it has been continuously blocked since it received the last engaging query from process P_i .
 - When a blocked process P_k receives a reply(i, j, k) message, it decrements $num_k(i)$ only if $wait_k(i)$ holds.
 - A process sends a reply message in response to an engaging query only after it has received a reply to every query message it has sent out for this engaging

query.

- The initiator process detects a deadlock when it has received reply messages to all the query messages it has sent out.

Initiate a diffusion computation for a blocked process P_i :

send $query(i, i, j)$ to all processes P_j in the dependent set DS_i of P_i ;
 $num_i(i) := |DS_i|$; $wait_i(i) := true$;

When a blocked process P_k receives a query(i, j, k):

if this is the engaging query for process P_i then
send $query(i, k, m)$ to all P_m in its dependent set DS_k ;
 $num_k(i) := |DS_k|$; $wait_k(i) := true$
else if $wait_k(i)$ then send a $reply(i, k, j)$ to P_j .

When a process P_k receives a reply(i, j, k):

if $wait_k(i)$ then
 $num_k(i) := num_k(i) - 1$;
if $num_k(i) = 0$ then
if $i = k$ then **declare a deadlock**
else send $reply(i, k, m)$ to the process P_m
which sent the engaging query.

Fig: Chandy–Misra–Haas algorithm for the OR model

Performance analysis

For every deadlock detection, the algorithm exchanges e query messages and e reply messages, where $e = n(n - 1)$ is the number of edges.

UNIT IV CONSENSUS AND RECOVERY

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure-Free System (Synchronous and Asynchronous) – Agreement in Synchronous Systems with Failures; Checkpointing and Rollback Recovery: Introduction – Background and Definitions – Issues in Failure Recovery – Checkpoint-based Recovery – Coordinated Checkpointing Algorithm – Algorithm for Asynchronous Checkpointing and Recovery

Problem definition

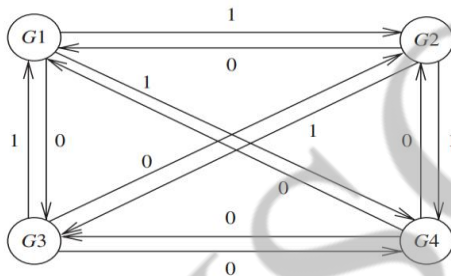
Agreement among the processes in a distributed system is a fundamental requirement for a wide range of applications. Many forms of coordination require the processes to exchange information to negotiate with one another and eventually reach a common understanding or agreement, before taking application-specific actions. A classical example is that of the commit decision in database systems, wherein the processes collectively decide whether to commit or abort a transaction that they participate in.

We first state some assumptions underlying our study of agreement algorithms:

- **Failure models** Among the n processes in the system, at most f processes can be faulty. A faulty process can behave in any manner allowed by the failure model assumed. The various failure models – fail-stop, send omission and receive omission, and Byzantine failures.
- **Synchronous/asynchronous communication** If a failure-prone process chooses to send a message to process P_i but fails, then P_i cannot detect the non-arrival of the message in an

asynchronous system. In a synchronous system, however, the scenario in which a message has not been sent can be recognized by the intended recipient, at the end of the round.

- **Network connectivity** The system has full logical connectivity, i.e., each process can communicate with any other by direct message passing.
- **Sender identification** A process that receives a message always knows the identity of the sender process.
- **Channel reliability** The channels are reliable, and only the processes may fail (under one of various failure models).
- **Authenticated vs. non-authenticated messages** With unauthenticated messages, when a faulty process relays a message to other processes, (i) it can forge the message and claim that it was received from another process, and (ii) it can also tamper with the contents of a received message before relaying it. When a process receives a message, it has no way to verify its authenticity. An unauthenticated message is also called an oral message or an unsigned message. Using authentication via techniques such as digital signatures, it is easier to solve the agreement problem because, if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the forgery or tampering. Thus, faulty processes can inflict less damage.
- **Agreement variable** The agreement variable may be boolean or multivalued, and need not be an integer.



The Byzantine agreement

The Byzantine agreement problem requires a designated process, called the source process, with an initial value

Problem definition agreement with the other processes about its initial value, subject to the following conditions:

- **Agreement** All non-faulty processes must agree on the same value.
- **Validity** If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.
- **Termination** Each non-faulty process must eventually decide on a value. The validity condition rules out trivial solutions, such as one in which the agreed upon value is a constant.

The consensus problem

The consensus problem differs from the Byzantine agreement problem in that each process has an initial value and all the correct processes must agree on a single value

- Agreement All non-faulty processes must agree on the same (single) value.
- Validity If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.
- Termination Each non-faulty process must eventually decide on a value.

The interactive consistency problem

The interactive consistency problem differs from the Byzantine agreement problem in that each process has an initial value, and all the correct processes must agree upon a set of values, with one value for each process

- Agreement All non-faulty processes must agree on the same array of values $A[v_1 \dots v_n]$
- Validity If process i is non-faulty and its initial value is v_i , then all nonfaulty processes agree on v_i as the i th element of the array A . If process j is faulty, then the non-faulty processes can agree on any value for $A[j]$.
- Termination Each non-faulty process must eventually decide on the array A

Overview of results:

Failure mode	Synchronous system (message-passing and sharedmemory)	Asynchronous system (message-passing and sharedmemory)
No Failure	agreement attainable; common knowledge attainable	agreement attainable; concurrent common knowledge
Crash Failure	agreement attainable $f < n$ processes	agreement not attainable
Byzantine Failure	agreement attainable $f \leq [(n - 1)/3]$ Byzantine processes	agreement not attainable

AGREEMENT IN A FAILURE-FREE SYSTEM (SYNCHRONOUS OR ASYNCHRONOUS)

In a failure-free system, consensus can be reached by collecting information from the different processes, arriving at a “decision,” and distributing this decision in the system.

A distributed mechanism would have each process broadcast its values to others, and each process computes the same function on the values received.

The decision can be reached by using an applicationspecific function – some simple examples being the majority, max, and min functions. Algorithms to collect the initial values and then distribute the decision may be based on the token circulation on a logical ring, or the three-phase

Consensus and agreement algorithms tree-based broadcast–converge cast–broadcast, or direct communication with all nodes.

AGREEMENT IN (MESSAGE-PASSING) SYNCHRONOUS SYSTEMS WITH FAILURES

CONSENSUS ALGORITHM FOR CRASH FAILURES (SYNCHRONOUS SYSTEM)

- The consensus algorithm for n processes where up to f processes where $f < n$ may fail in a fail stop failure model.
- Here the consensus variable x is integer value; each process has initial value x_i . If up to f failures are to be tolerated then algorithm has $f+1$ rounds, in each round a process i sends the value of its variable x_i to all other processes if that value has not been sent before.
- So, of all the values received within that round and its own value x_i at that start of the round the process takes minimum and updates x_i occur $f + 1$ rounds the local value x_i guaranteed to be the consensus value.
- If one process is faulty, among three processes then $f = 1$. So the agreement requires $f + 1$ that is equal to two rounds.
- If it is faulty let us say it will send 0 to 1 process and 1 to another process i, j and k . Now, on receiving one on receiving 0 it will broadcast 0 over here and this particular process on receiving 1 it will broadcast 1 over here.
- So, this will complete one round in this one round and this particular process on receiving 1 it will send 1 over here and this on the receiving 0 it will send 0 over here.

(global constants)

integer: f ; // maximum number of crash failures tolerated

(local variables)

integer: $x \leftarrow$ local value;

- (1) Process P_i ($1 \leq i \leq n$) executes the consensus algorithm for up to f crash failures:
 - (1a) **for round from 1 to $f + 1$ do**
 - (1b) **if** the current value of x has not been broadcast **then**
 - (1c) **broadcast**(x);
 - (1d) $y_j \leftarrow$ value (if any) received from process j in this round;
 - (1e) $x \leftarrow \min_{\forall j}(x, y_j)$;
 - (1f) **output** x as the consensus value.

Algorithm 14.1 Consensus with up to f fail-stop processes in a system of n processes, $n > f$ [8]. Code shown is for process P_i , $1 \leq i \leq n$.

- The agreement condition is satisfied because in the $f+1$ rounds, there must be at least one round in which no process failed.
- In this round, say round r , all the processes that have not failed so far succeed in broadcasting their values, and all these processes take the minimum of the values broadcast and received in that round.
- Thus, the local values at the end of the round are the same, say x_r for all non-failed processes.
- In further rounds, only this value may be sent by each process at most once, and no process i will update its value x_r .

- The validity condition is satisfied because processes do not send fictitious values in this failure model.
- For all i , if the initial value is identical, then the only value sent by any process is the value that has been agreed upon as per the agreement condition.
- The termination condition is seen to be satisfied.

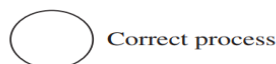
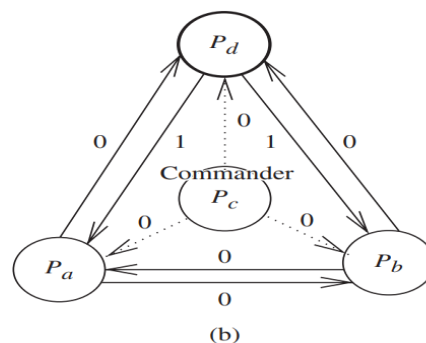
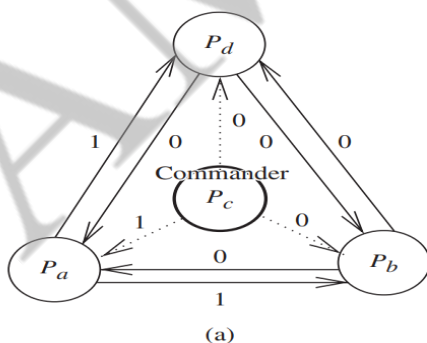
Complexity: The complexity of this particular algorithm is it requires $f + 1$ rounds where $f < n$ and the number of messages is $O(n^2)$ in each round and each message has one integer hence the total number of messages is $O((f+1) \cdot n^2)$ is the total number of rounds and in each round n^2 messages are required.

Consensus algorithms for Byzantine failures (synchronous system)

- Total of n processes, at most f of which can be faulty
- Reliable communication medium
- Fully connected
- Receiver always knows the identity of the sender of a message
- Byzantine faults
- In each round, a process receives messages, perform computation and sends messages

Solution to Byzantine Agreement problem

- The solution is first defined and solved by Lamport.
- Pease showed that in a fully connected network, it is impossible to reach an agreement if number of faulty processes ' f ' exceeds $(n-1)/3$ where n is the number of processes.
- ie $f \leq (n-1)/3$



.....> First round exchange

————> Second round exchange

STEPS FOR BYZANTINE GENERALS (ITERATIVE FORMULATION), SYNCHRONOUS, MESSAGE-PASSING:

(variables)

boolean: $v \leftarrow$ initial value;

integer: $f \leftarrow$ maximum number of malicious processes, $\leq \lfloor \frac{n-1}{3} \rfloor$;

tree of boolean:

- level 0 root is v_{init}^L , where $L = \langle \rangle$;
- level h ($f \geq h > 0$) nodes: for each v_j^L at level $h-1 = \text{sizeof}(L)$, its $n-2 - \text{sizeof}(L)$ descendants at level h are $v_k^{\text{concat}(\langle j \rangle, L)}$, $\forall k$ such that $k \neq j$, i and k is not a member of list L .

(message type)

$OM(v, Dests, List, faulty)$, where the parameters are as in the recursive formulation.

(1) Initiator (i.e., Commander) initiates Oral Byzantine agreement:

(1a) send $OM(v, N - \{i\}, \langle P_i \rangle, f)$ to $N - \{i\}$;

(1b) return(v).

(2) (Non-initiator, i.e., Lieutenant) receives Oral Message OM :

(2a) for $rnd = 0$ to f do

(2b) for each message OM that arrives in this round, do

(2c) receive $OM(v, Dests, L = \langle P_{k_1} \dots P_{k_{f+1-faulty}} \rangle, faulty)$ from P_{k_1} ;
 $// faulty + round = f, |Dests| + \text{sizeof}(L) = n$

(2d) $v_{\text{head}(L)}^{\text{tail}(L)} \leftarrow v$; $// \text{sizeof}(L) + faulty = f + 1$. fill in estimate.

(2e) send $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \dots P_{k_{f+1-faulty}} \rangle, faulty - 1)$ to $Dests - \{i\}$ if $rnd < f$;

(2f) for $level = f - 1$ down to 0 do

(2g) for each of the $1 \cdot (n-2) \cdot \dots \cdot (n - (level + 1))$ nodes v_x^L in level $level$, do

(2h) $v_x^L (x \neq i, x \notin L) = \text{majority}_{y \notin \text{concat}(\langle x \rangle, L)} (v_x^L, v_y^{\text{concat}(\langle x \rangle, L)})$;

STEPS FOR BYZANTINE GENERALS (RECURSIVE FORMULATION), SYNCHRONOUS, MESSAGE-PASSING:

(variables)

boolean: $v \leftarrow$ initial value;

integer: $f \leftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;

(message type)

$Oral_Msg(v, Dests, List, faulty)$, where

v is a boolean,

$Dests$ is a set of destination process ids to which the message is sent,

$List$ is a list of process ids traversed by this message, ordered from most recent to earliest,

$faulty$ is an integer indicating the number of malicious processes to be tolerated.

$Oral_Msg(f)$, where $f > 0$:

- 1 The algorithm is initiated by the Commander, who sends his source value v to all other processes using a $OM(v, N, \langle i \rangle, f)$ message. The commander returns his own value v and terminates.
- 2 [Recursion unfolding:] For each message of the form $OM(v_j, Dests, List, f')$ received in this round from some process j , the process i uses the value v_j it receives from the source, and using that value, acts as a new source. (If no value is received, a default value is assumed.)
To act as a new source, the process i initiates $Oral_Msg(f' - 1)$, wherein it sends $OM(v_j, Dests - \{i\}, \text{concat}(\langle i \rangle, L), (f' - 1))$ to destinations not in $\text{concat}(\langle i \rangle, L)$ in the next round.
- 3 [Recursion folding:] For each message of the form $OM(v_j, Dests, List, f')$ received in Step 2, each process i has computed the agreement value v_k , for each k not in $List$ and $k \neq i$, corresponding to the value received from P_k after traversing the nodes in $List$, at one level lower in the recursion. If it receives no value in this round, it uses a default value. Process i then uses the value $\text{majority}_{k \notin List, k \neq i}(v_j, v_k)$ as the agreement value and returns it to the next higher level in the recursive invocation.

$Oral_Msg(0)$:

- 1 [Recursion unfolding:] Process acts as a source and sends its value to each other process.
- 2 [Recursion folding:] Each process uses the value it receives from the other sources, and uses that value as the agreement value. If no value is received, a default value is assumed.

CODE FOR THE PHASE KING ALGORITHM:

Each phase has a unique "phase king" derived, say, from PID. Each phase has two rounds:

- 1 in 1st round, each process sends its estimate to all other processes.
- 2 in 2nd round, the "Phase king" process arrives at an estimate based on the values it received in 1st round, and broadcasts its new estimate to all others.

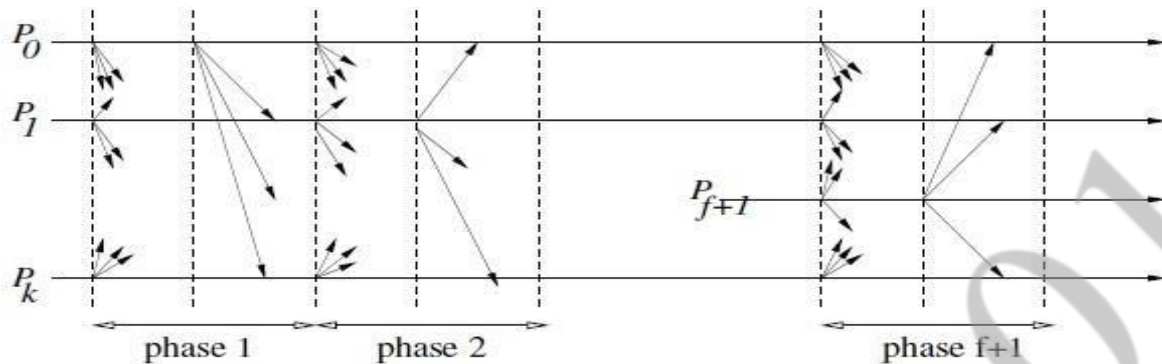


Fig. Message pattern for the phase-king algorithm.

PHASE KING ALGORITHM CODE:

```

(variables)
boolean:  $v \leftarrow$  initial value;
integer:  $f \leftarrow$  maximum number of malicious processes,  $f < \lceil n/4 \rceil$ ;

(1) Each process executes the following  $f + 1$  phases, where  $f < n/4$ :
(1a) for phase = 1 to  $f + 1$  do
(1b)   Execute the following Round 1 actions:           // actions in round one of each phase
(1c)     broadcast  $v$  to all processes;
(1d)     await value  $v_j$  from each process  $P_j$ ;
(1e)      $majority \leftarrow$  the value among the  $v_j$  that occurs  $> n/2$  times (default if no maj.);
(1f)      $mult \leftarrow$  number of times that  $majority$  occurs;
(1g)   Execute the following Round 2 actions:           // actions in round two of each phase
(1h)     if  $i = phase$  then // only the phase leader executes this send step
(1i)       broadcast  $majority$  to all processes;
(1j)       receive  $tiebreaker$  from  $P_{phase}$  (default value if nothing is received);
(1k)       if  $mult > n/2 + f$  then
(1l)          $v \leftarrow majority$ ;
(1m)       else  $v \leftarrow tiebreaker$ ;
(1n)       if  $phase = f + 1$  then
(1o)         output decision value  $v$ .

```

$(f + 1)$ phases, $(f + 1)[(n - 1)(n + 1)]$ messages, and can tolerate up to $f < dn=4e$ malicious processes

Correctness Argument

- 1 Among $f + 1$ phases, at least one phase k where phase-king is non-malicious.
- 2 In phase k , all non-malicious processes P_i and P_j will have same

estimate of consensus value as P_k does.

- P_i and P_j use their own majority values. (P_i 's mult $> n/2 + f$)
- P_i uses its majority value; P_j uses phase-king's tie-breaker value. (P_i 's mult $> n/2 + f$, P_j 's mult $> n/2$ for same value)
- P_i and P_j use the phase-king's tie-breaker value. (In the phase in which P_k is non-malicious, it sends same value to P_i and P_j)

In all 3 cases, argue that P_i and P_j end up with same value as estimate

- If all non-malicious processes have the value x at the start of a phase, they will continue to have x as the consensus value at the end of the phase.

Check pointing and rollback recovery: Introduction

- Rollback recovery protocols restore the system back to a consistent state after a failure,
- It achieves fault tolerance by periodically saving the state of a process during the failure-free execution
- It treats a distributed system application as a collection of processes that communicate over a network

Checkpoints

The saved state is called a checkpoint, and the procedure of restarting from a previously checkpointed state is called rollback recovery. A checkpoint can be saved on either the stable storage or the volatile storage

Why is rollback recovery of distributed systems complicated?

Messages induce inter-process dependencies during failure-free operation

Rollback propagation

The dependencies among messages may force some of the processes that did not fail to roll back. This phenomenon of cascaded rollback is called the domino effect.

Uncoordinated check pointing

If each process takes its checkpoints independently, then the system cannot avoid the domino effect – this scheme is called independent or uncoordinated check pointing

Techniques that avoid domino effect

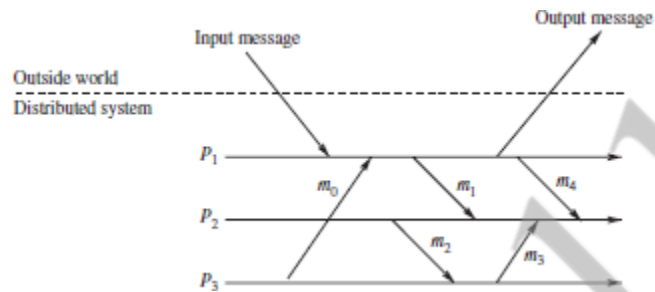
1. Coordinated check pointing rollback recovery - Processes coordinate their checkpoints to form a system-wide consistent state
2. Communication-induced check pointing rollback recovery - Forces each process to take checkpoints based on information piggybacked on the application.
3. Log-based rollback recovery - Combines check pointing with logging of non-deterministic events
 - relies on piecewise deterministic (PWD) assumption.

Background and definitions

System model

- A distributed system consists of a fixed number of processes, P_1, P_2, \dots, P_N , which communicate only through messages.

- Processes cooperate to execute a distributed application and interact with the outside world by receiving and sending input and output messages, respectively.
- Rollback-recovery protocols generally make assumptions about the reliability of the inter-process communication.
- Some protocols assume that the communication uses first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem can lose, duplicate, or reorder messages.
- Rollback-recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world.



An example of a distributed system with three processes.

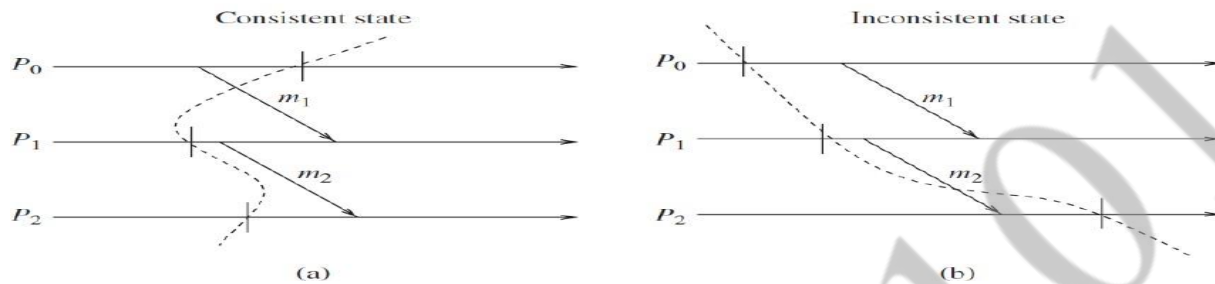
A local checkpoint

- All processes save their local states at certain instants of time
- A local check point is a snapshot of the state of the process at a given instance
- Assumption
 - A process stores all local checkpoints on the stable storage
 - A process is able to roll back to any of its existing local checkpoints
- $C_{i,k}$ – The k th local checkpoint at process P_i
- $C_{i,0}$ – A process P_i takes a checkpoint $C_{i,0}$ before it starts execution

Consistent states

- A global state of a distributed system is a collection of the individual states of all participating processes and the states of the communication channels
- Consistent global state
 - a global state that may occur during a failure-free execution of distribution of distributed computation
 - if a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message
- A global checkpoint is a set of local checkpoints, one from each process
- A consistent global checkpoint is a global checkpoint such that no message is sent by a process after taking its local point that is received by another process before taking its checkpoint.

Consistent states - examples



- For instance, Figure shows two examples of global states.
- The state in fig (a) is consistent and the state in Figure (b) is inconsistent.
- Note that the consistent state in Figure (a) shows message m_1 to have been sent but not yet received, but that is alright.
- The state in Figure (a) is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event.
- The state in Figure (b) is inconsistent because process P_2 is shown to have received m_2 but the state of process P_1 does not reflect having sent it.
- Such a state is impossible in any failure-free, correct computation. Inconsistent states occur because of failures.

Interactions with outside world

A distributed system often interacts with the outside world to receive input data or deliver the outcome of a computation. If a failure occurs, the outside world cannot be expected to roll back. For example, a printer cannot roll back the effects of printing a character

Outside World Process (OWP)

- It is a special process that interacts with the rest of the system through message passing.
- It is therefore necessary that the outside world see a consistent behavior of the system despite failures.
- Thus, before sending output to the OWP, the system must ensure that the state from which the output is sent will be recovered despite any future failure.

A common approach is to save each input message on the stable storage before allowing the application program to process it.

An interaction with the outside world to deliver the outcome of a computation is shown on the process-line by the symbol “||”.

Different types of Messages

1. In-transit message

- messages that have been sent but not yet received

2. Lost messages

- messages whose “send” is done but “receive” is undone due to rollback

3. Delayed messages

- messages whose “receive” is not recorded because the receiving process was either down or the message arrived after rollback

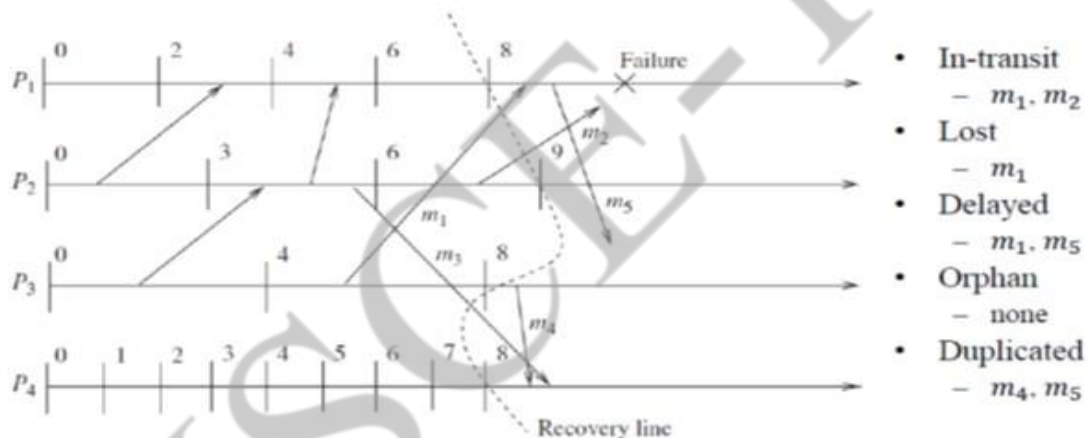
4. Orphan messages

- messages with “receive” recorded but message “send” not recorded
- do not arise if processes roll back to a consistent global state

5. Duplicate messages

- arise due to message logging and replaying during process recovery

Messages – example



In-transit messages

In Figure , the global state $\{C1,8, C2, 9, C3,8, C4,8\}$ shows that message m_1 has been sent but not yet received. We call such a message an *in-transit* message. Message m_2 is also an in-transit message.

Delayed messages

Messages whose receive is not recorded because the receiving process was either down or the message arrived after the rollback of the receiving process, are called *delayed* messages. For example, messages m_2 and m_5 in Figure are delayed messages.

Lost messages

Messages whose send is not undone but receive is undone due to rollback are called lost messages. This type of messages occurs when the process rolls back to a checkpoint prior to

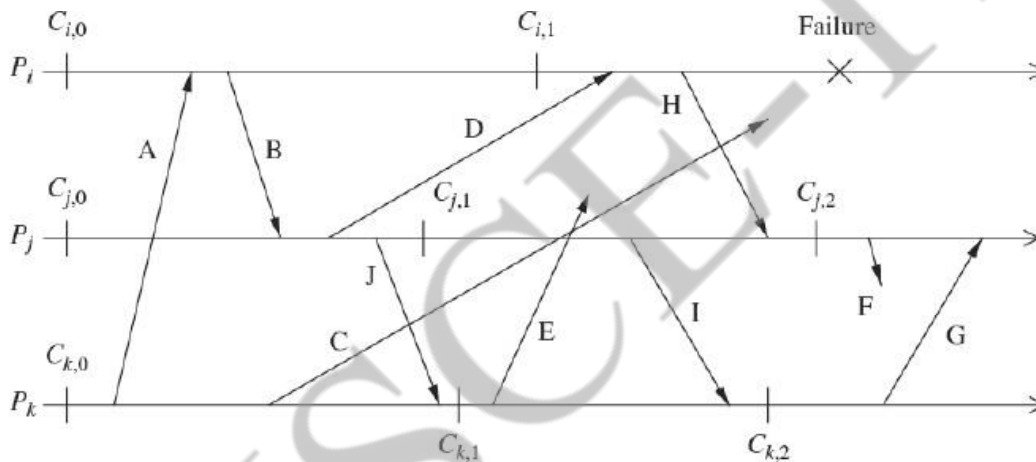
reception of the message while the sender does not rollback beyond the send operation of the message. In Figure , message m1 is a lost message.

Duplicate messages

- Duplicate messages arise due to message logging and replaying during process recovery. For example, in Figure, message m4 was sent and received before the rollback. However, due to the rollback of process P4 to C4,8 and process P3 to C3,8, both send and receipt of message m4 are undone.
- When process P3 restarts from C3,8, it will resend message m4.
- Therefore, P4 should not replay message m4 from its log.
- If P4 replays message m4, then message m4 is called a duplicate message.

Issues in failure recovery

In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure and recovery



- The computation comprises of three processes P_i , P_j , and P_k , connected through a communication network. The processes communicate solely by exchanging messages over fault free, FIFO communication channels.
- Processes P_i , P_j , and P_k , have taken checkpoints $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}\}$, respectively, and these processes have exchanged messages A to J
- Suppose process P_i fails at the instance indicated in the figure. All the contents of the volatile memory of P_i are lost and, after P_i has recovered from the failure, the system needs to be restored to a consistent global state from where the processes can resume their execution.
- Process P_i 's state is restored to a valid state by rolling it back to its most recent checkpoint $C_{i,1}$. To restore the system to a consistent state, the process P_j rolls back to checkpoint $C_{j,1}$ because the rollback of process P_i to checkpoint $C_{i,1}$ created an orphan message H (the receive event of H is recorded at process P_j while the send event of H has been undone at process P_i).
- P_j does not roll back to checkpoint $C_{j,2}$ but to checkpoint $C_{j,1}$. An orphan message I is created due to the roll back of process P_j to checkpoint $C_{j,1}$. To eliminate this orphan message, process P_k rolls back to checkpoint $C_{k,1}$.
- Messages C, D, E, and F are potentially problematic. Message C is in transit during the failure and it is a delayed message. The delayed message C has several possibilities: C might arrive at process P_i before it recovers, it might arrive while P_i is recovering, or it might arrive after P_i has

completed recovery. Each of these cases must be dealt with correctly.

- Message D is a lost message since the send event for D is recorded in the restored state for process P_j , but the receive event has been undone at process P_i . Process P_j will not resend D without an additional mechanism.
- Messages E and F are delayed orphan messages and pose perhaps the most serious problem of all the messages. When messages E and F arrive at their respective destinations, they must be discarded since their send events have been undone. Processes, after resuming execution from their checkpoints, will generate both of these messages.
- Lost messages like D can be handled by having processes keep a message log of all the sent messages. So when a process restores to a checkpoint, it replays the messages from its log to handle the lost message problem.
- Overlapping failures further complicate the recovery process. If overlapping failures are to be tolerated, a mechanism must be introduced to deal with amnesia and the resulting inconsistencies.

Checkpoint-based recovery

Checkpoint-based rollback-recovery techniques can be classified into three categories:

1. *Uncoordinated checkpointing*
2. *Coordinated checkpointing*
3. *Communication-induced checkpointing*

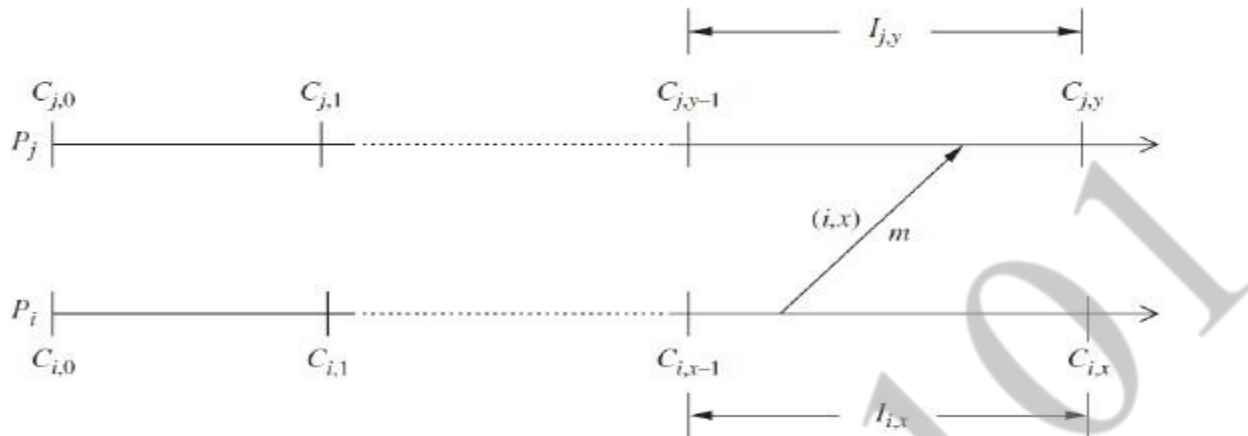
1. Uncoordinated Checkpointing

- Each process has autonomy in deciding when to take checkpoints
- Advantages
 - The lower runtime overhead during normal execution
- Disadvantages
 1. Domino effect during a recovery
 2. Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints
 3. Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm
 4. Not suitable for application with frequent output commits
- The processes record the dependencies among their checkpoints caused by message exchange during failure-free operation
- The following direct dependency tracking technique is commonly used in uncoordinated checkpointing.

Direct dependency tracking technique

- Assume each process P_i starts its execution with an initial checkpoint $C_{i,0}$
- $I_{i,x}$: checkpoint interval, interval between $C_{i,x-1}$ and $C_{i,x}$

- When P_j receives a message m during $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto stable storage when P_j takes $C_{j,y}$



- When a failure occurs, the recovering process initiates rollback by broadcasting a *dependency request* message to collect all the dependency information maintained by each process.
- When a process receives this message, it stops its execution and replies with the dependency information saved on the stable storage as well as with the dependency information, if any, which is associated with its current state.
- The initiator then calculates the recovery line based on the global dependency information and broadcasts a *rollback request* message containing the recovery line.
- Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution; otherwise, it rolls back to an earlier checkpoint as indicated by the recovery line.

2. Coordinated Checkpointing

In coordinated checkpointing, processes orchestrate their checkpointing activities so that all local checkpoints form a consistent global state

Types

1. Blocking Checkpointing: After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete
Disadvantages: The computation is blocked during the checkpointing
2. Non-blocking Checkpointing: The processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

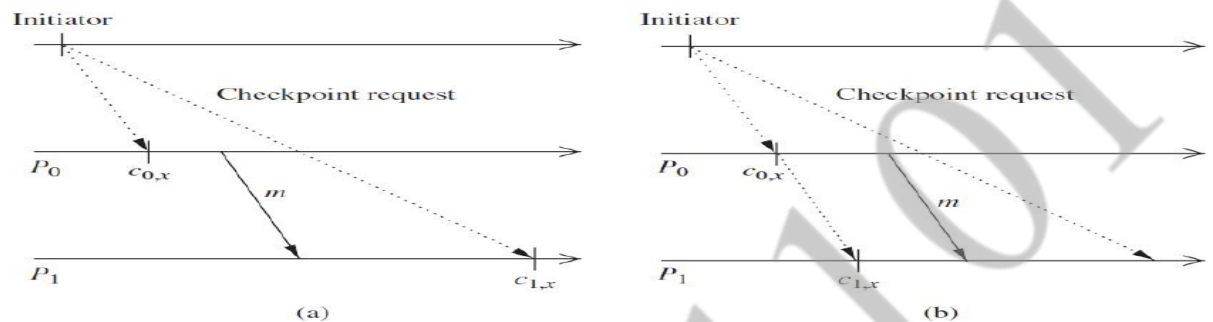
Example (a) : Checkpoint inconsistency

- Message m is sent by P_0 after receiving a checkpoint request from the checkpoint coordinator
- Assume m reaches P_1 before the checkpoint request
- This situation results in an inconsistent checkpoint since checkpoint $C_{1,x}$ shows the receipt of message m from P_0 , while checkpoint $C_{0,x}$ does not show m being sent from P_0

Example (b) : A solution with FIFO channels

- If channels are FIFO, this problem can be avoided by preceding the first post-checkpoint message on each channel by a checkpoint request, forcing each process to take a checkpoint before receiving the first post-checkpoint message

Coordinated Checkpointing



Impossibility of min-process non-blocking checkpointing

- A min-process, non-blocking checkpointing algorithm is one that forces only a minimum number of processes to take a new checkpoint, and at the same time it does not force any process to suspend its computation.

Algorithm

- The algorithm consists of two phases. During the first phase, the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request.
- Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified.
- During the second phase, all processes identified in the first phase take a checkpoint. The result is a consistent checkpoint that involves only the participating processes.
- In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.

3. Communication-induced Checkpointing

Communication-induced checkpointing is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently. Processes may be forced to take additional checkpoints

Two types of checkpoints

1. Autonomous checkpoints
2. Forced checkpoints

The checkpoints that a process takes independently are called *local* checkpoints, while those that a process is forced to take are called *forced* checkpoints.

- Communication-induced check pointing piggybacks protocol- related information on

each application message

- The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line
- The forced checkpoint must be taken before the application may process the contents of the message
- In contrast with coordinated check pointing, no special coordination messages are exchanged

Two types of communication-induced checkpointing

1. Model-based checkpointing
2. Index-based checkpointing.

Model-based checkpointing

- Model-based checkpointing prevents patterns of communications and checkpoints that could result in inconsistent states among the existing checkpoints.
- No control messages are exchanged among the processes during normal operation. All information necessary to execute the protocol is piggybacked on application messages
- There are several domino-effect-free checkpoint and communication model.
- The MRS (mark, send, and receive) model of Russell avoids the domino effect by ensuring that within every checkpoint interval all message receiving events precede all message-sending events.

Index-based checkpointing.

- Index-based communication-induced checkpointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.

KOO AND TOUEG COORDINATED CHECKPOINTING AND RECOVERY TECHNIQUE:

- Koo and Toueg coordinated check pointing and recovery technique takes a consistent set of checkpoints and avoids the domino effect and livelock problems during the recovery.
- Includes 2 parts: the check pointing algorithm and the recovery algorithm

A. The Checkpointing Algorithm

The checkpoint algorithm makes the following assumptions about the distributed system:

- Processes communicate by exchanging messages through communication channels.
- Communication channels are FIFO.
- Assume that end-to-end protocols (the sliding window protocol) exist to handle with message loss due to rollback recovery and communication failure.
- Communication failures do not divide the network.

The checkpoint algorithm takes two kinds of checkpoints on the stable storage: Permanent and Tentative.

A *permanent checkpoint* is a local checkpoint at a process and is a part of a consistent global checkpoint.

A *tentative checkpoint* is a temporary checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint algorithm.

The algorithm consists of two phases.

First Phase

1. An initiating process P_i takes a tentative checkpoint and requests all other processes to take tentative checkpoints. Each process informs P_i whether it succeeded in taking a tentative checkpoint.
2. A process says “no” to a request if it fails to take a tentative checkpoint
3. If P_i learns that all the processes have successfully taken tentative checkpoints, P_i decides that all tentative checkpoints should be made permanent; otherwise, P_i decides that all the tentative checkpoints should be thrown-away.

Second Phase

1. P_i informs all the processes of the decision it reached at the end of the first phase.
2. A process, on receiving the message from P_i will act accordingly.
3. Either all or none of the processes advance the checkpoint by taking permanent checkpoints.
4. The algorithm requires that after a process has taken a tentative checkpoint, it cannot send messages related to the basic computation until it is informed of P_i 's decision.

Correctness: for two reasons

- i. Either all or none of the processes take permanent checkpoint
- ii. No process sends message after taking permanent checkpoint

An Optimization

The above protocol may cause a process to take a checkpoint even when it is not necessary for consistency. Since taking a checkpoint is an expensive operation, we avoid taking checkpoints.

B. The Rollback Recovery Algorithm

The rollback recovery algorithm restores the system state to a consistent state after a failure. The rollback recovery algorithm assumes that a single process invokes the algorithm. It assumes that the checkpoint and the rollback recovery algorithms are not invoked concurrently. The rollback recovery algorithm has two phases.

First Phase

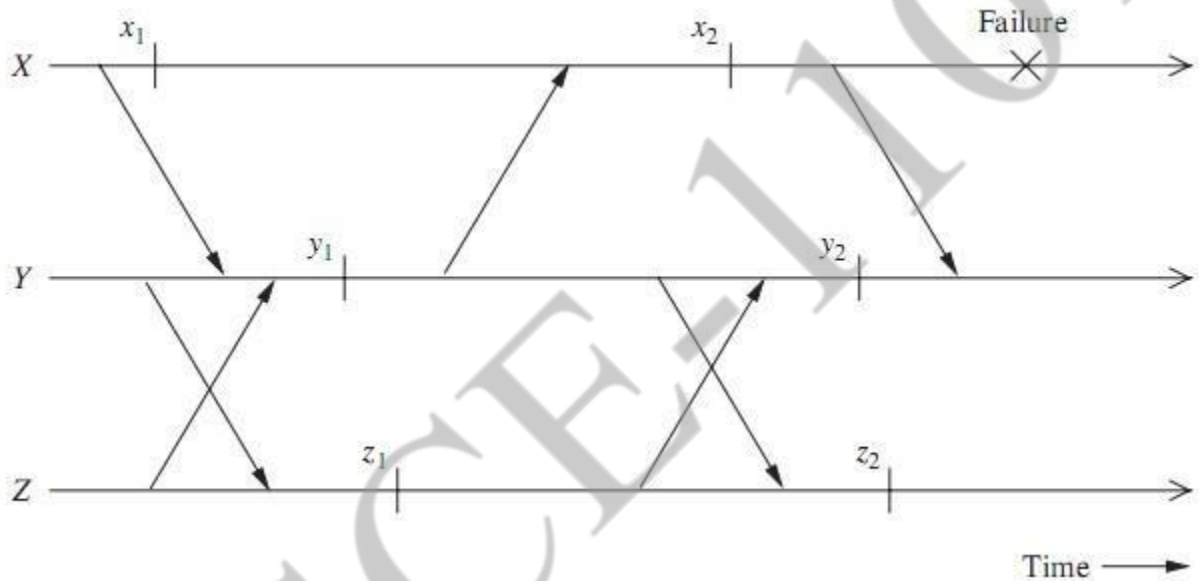
1. An initiating process P_i sends a message to all other processes to check if they all are willing to restart from their previous checkpoints.
2. A process may reply “no” to a restart request due to any reason (e.g., it is already participating in a check pointing or a recovery process initiated by some other process).
3. If P_i learns that all processes are willing to restart from their previous checkpoints, P_i decides that all processes should roll back to their previous checkpoints. Otherwise,
4. P_i aborts the roll back attempt and it may attempt a recovery at a later time.

Second Phase

1. P_i propagates its decision to all the processes.
2. On receiving P_i 's decision, a process acts accordingly.
3. During the execution of the recovery algorithm, a process cannot send messages related to the underlying computation while it is waiting for P_i 's decision.

Correctness: Resume from a consistent state

Optimization: May not to recover all, since some of the processes did not change anything



The above protocol, in the event of failure of process X, the above protocol will require processes X, Y, and Z to restart from checkpoints x_2 , y_2 , and z_2 , respectively.

Process Z need not roll back because there has been no interaction between process Z and the other two processes since the last checkpoint at Z.

ALGORITHM FOR ASYNCHRONOUS CHECKPOINTING AND RECOVERY:

The algorithm of Juang and Venkatesan for recovery in a system that uses asynchronous checkpointing.

A. System Model and Assumptions

The algorithm makes the following assumptions about the underlying system:

- The communication channels are reliable, deliver the messages in FIFO order and have infinite buffers.
- The message transmission delay is arbitrary, but finite.
- Underlying computation/application is event-driven: process P is at state s , receives message m , processes the message, moves to state s' and send messages out. So the triplet $(s, m, msgs_sent)$ represents the state of P

Two type of log storage are maintained:

- Volatile log: short time to access but lost if processor crash. Move to stable log periodically.
- Stable log: longer time to access but remained if crashed

A. Asynchronous Check pointing

- After executing an event, the triplet is recorded without any synchronization with other processes.
- Local checkpoint consist of set of records, first are stored in volatile log, then moved to stable log.

B. The Recovery Algorithm

Notations and data structure

The following notations and data structure are used by the algorithm:

- $RCVD_{i \leftarrow j}(CkPt_i)$ represents the number of messages received by processor p_i from processor p_j , from the beginning of the computation till the checkpoint $CkPt_i$.
- $SENT_{i \rightarrow j}(CkPt_i)$ represents the number of messages sent by processor p_i to processor p_j , from the beginning of the computation till the checkpoint $CkPt_i$.

Basic idea

- Since the algorithm is based on asynchronous check pointing, the main issue in the recovery is to find a consistent set of checkpoints to which the system can be restored.
- The recovery algorithm achieves this by making each processor keep track of both the number of messages it has sent to other processors as well as the number of messages it has received from other processors.
- Whenever a processor rolls back, it is necessary for all other processors to find out if any message has become an orphan message. Orphan messages are discovered by comparing the number of messages sent to and received from neighboring processors.

For example, if $RCVD_{i \leftarrow j}(CkPt_i) > SENT_{j \rightarrow i}(CkPt_j)$ (that is, the number of messages received by processor p_i from processor p_j is greater than the number of messages sent by processor p_j to processor p_i , according to the current states the processors), then one or more messages at processor p_j are orphan messages.

The Algorithm

When a processor restarts after a failure, it broadcasts a ROLLBACK message that it had failed

Procedure RollBack_Recovery

processor p_i executes the following:

STEP (a)

if processor p_i is recovering after a failure **then**

$CkPt_i :=$ latest event logged in the stable storage

else

$CkPt_i :=$ latest event that took place in p_i {The latest event at p_i can be either in stable or in

volatile storage.}

end if

STEP (b)

for $k = 1$ to N { N is the number of processors in the system} **do**

for each neighboring processor p_j **do**

compute $SENT_{i \rightarrow j}(CkPt_i)$

send a $ROLLBACK(i, SENT_{i \rightarrow j}(CkPt_i))$ message to p_j

end for

for every $ROLLBACK(j, c)$ message received from a neighbor j **do**

if $RCVD_{i \leftarrow j}(CkPt_j) > c$ {Implies the presence of orphan messages} **then**

find the latest event e such that $RCVD_{i \leftarrow j}(e) = c$ {Such an event e may be in the volatile storage or stable storage.}

$CkPt_i := e$

end if

end for

end for{for k }

D. An Example

Consider an example shown in Figure 2 consisting of three processors. Suppose processor Y fails and restarts. If event e_{y2} is the latest checkpointed event at Y, then Y will restart from the state corresponding to e_{y2} .

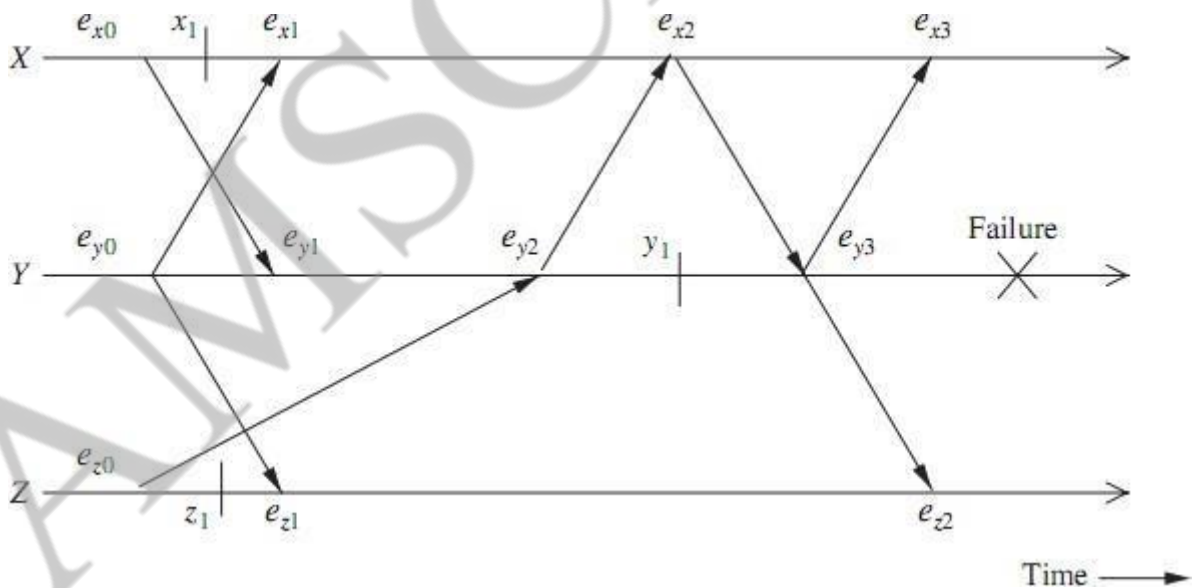


Figure 2: An example of Juan-Venkatesan algorithm.

- Because of the broadcast nature of $ROLLBACK$ messages, the recovery algorithm is initiated at processors X and Z.
- Initially, X, Y, and Z set $CkPt_X \leftarrow e_{x3}$, $CkPt_Y \leftarrow e_{y2}$ and $CkPt_Z \leftarrow e_{z2}$, respectively, and X, Y, and Z send the following messages during the first iteration:

- Y sends $\text{ROLLBACK}(Y,2)$ to X and $\text{ROLLBACK}(Y,1)$ to Z;
- X sends $\text{ROLLBACK}(X,2)$ to Y and $\text{ROLLBACK}(X,0)$ to Z;
- Z sends $\text{ROLLBACK}(Z,0)$ to X and $\text{ROLLBACK}(Z,1)$ to Y.

Since $\text{RCVDX} \leftarrow Y (\text{CkPtX}) = 3 > 2$ (2 is the value received in the $\text{ROLLBACK}(Y,2)$ message from Y), X will set CkPtX to ex2 satisfying $\text{RCVDX} \leftarrow Y (\text{ex2}) = 1 \leq 2$.

Since $\text{RCVDZ} \leftarrow Y (\text{CkPtZ}) = 2 > 1$, Z will set CkPtZ to ez1 satisfying $\text{RCVDZ} \leftarrow Y (\text{ez1}) = 1 \leq 1$.

At Y, $\text{RCVDY} \leftarrow X (\text{CkPtY}) = 1 < 2$ and $\text{RCVDY} \leftarrow Z (\text{CkPtY}) = 1 = \text{SENTZ} \leftarrow Y (\text{CkPtZ})$.

Y need not roll back further.

In the second iteration, Y sends $\text{ROLLBACK}(Y,2)$ to X and $\text{ROLLBACK}(Y,1)$ to Z;

Z sends $\text{ROLLBACK}(Z,1)$ to Y and $\text{ROLLBACK}(Z,0)$ to X;

X sends $\text{ROLLBACK}(X,0)$ to Z and $\text{ROLLBACK}(X, 1)$ to Y.

If Y rolls back beyond ey3 and loses the message from X that caused ey3 , X can resend this message to Y because ex2 is logged at X and this message available in the log. The second and third iteration will progress in the same manner. The set of recovery points chosen at the end of the first iteration, $\{\text{ex2}, \text{ey2}, \text{ez1}\}$, is consistent, and no further rollback occurs.

Unit - 5

Cloud Computing

Definition of cloud computing - Characteristics of cloud - cloud deployment models - Cloud service models - Driving factors and challenges of cloud - Virtualization - Load balancing - Scalability and Elasticity - Replication - Monitoring - cloud services and platforms. Compute services - storage services - Application Services.

Definition of cloud computing:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g.: networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Characteristics of cloud:

on demand self service

Broad network access

Resource pooling

Rapid elasticity

Measured services

performance

Reduced costs

outsourced management

Reliability

Multi-tenancy. → The cloud allows

multiple users to make use of the same shared resources.

Cloud deployment Models:

1. Public cloud:

The cloud services are available to the general public or a large group of companies. The cloud resources are shared among different users. Services are provided by a third party cloud provider. It is best suited for who want to use ^{Cloud} infrastructure for development and testing of Appln.

2. private cloud:

The cloud infrastructure is operated for exclusive use of a single organization. It is best suited for Appln where security is very important and very tight control over

their data. Cloud infrastructure can be setup on premise or off-premise and may be managed internally or by a third party.

3. Hybrid cloud:

The hybrid cloud deployed model combines the services of multiple cloud (private or public). Hybrid cloud are best suited for organization that want to take advantage of secured Appln and data hosting on a private cloud.

4. Community cloud:

The cloud services are shared by several organizations that have the same Policy and Compliance considerations. It is best suited for organizations that want access to the same Appln and data and cloud costs to be shared with the larger group.

PUBLIC CLOUD

Available for public use or a large industry group

COMMUNITY CLOUD

Available for shared ~~resource~~ use of several organizations supporting a specific community

PRIVATE CLOUD

Operated for exclusive use of a single organization

HYBRID CLOUD

Combines multiple cloud that remain unique but bound together to offer Appln and data portability

Fig) CLOUD DEPLOYMENT MODELS

cloud Service Models:

Software as a Service (SaaS)
App, mgmt and User Interfaces
provided over N/w

Platform as a Service (PaaS)
App development frameworks,
OS and development frameworks

Infrastructure as a Service (IaaS)
Virtual computing, storage and N/w
resources that can be provisioned
on demand

1. Infrastructure as a Service (IaaS)

IaaS provides the capability to provision computing and storage resources to users as virtual machine instances and virtual storage. Users can start, stop, configure and manage the virtual machine instances and virtual storage. Virtual resources provisioned by the users are billed based on a pay-per use paradigm.

Amazon EC2, Google Compute Engine,
Azure VMs.

2. Platform as a Service (PaaS):

PaaS provides the users the capability to develop and deploy Appln in the cloud using development tools, APIs, s/w libraries and services provided by the cloud service provider. The cloud service provider manages the cloud infrastructure including servers, n/w, OS and storage. The users themselves are responsible for developing, deploying, configuring and managing Appln.

Google App Engine, Github, OpenStack

3. Software as a Service (SaaS):

SaaS provides the users a complete s/w Appln or the user interface to the Appln itself. The cloud service provider manages servers, networks, OS, storage and Appln s/w and the user is unaware of underlying architecture of the cloud. Appln are provided to the user through a thin client interface (eg. browser). SaaS Appln are platform

independent and can be accessed from various client devices.

Salesforce.com, Facebook, Yahoo, Dropbox, LinkedIn, slideshare.

Virtualization:

Virtualization refers to the partitioning the resources of a physical s/m (computing, storage, n/w and memory) into multiple virtual resources. The virtualization layer partitions the physical resources into multiple virtual machines. The virtualization layer allows multiple OS instances to run currently as virtual machines on the same underlying physical resources.

hypervisor:

The virtualization layer consists of a hypervisor or a virtual machine monitor (VMM). A hypervisor is a s/w that you can use to run multiple virtual machines on a single physical machine.

The hypervisor allocates the underlying physical computing resources such as CPU and memory to individual virtual machines as required.

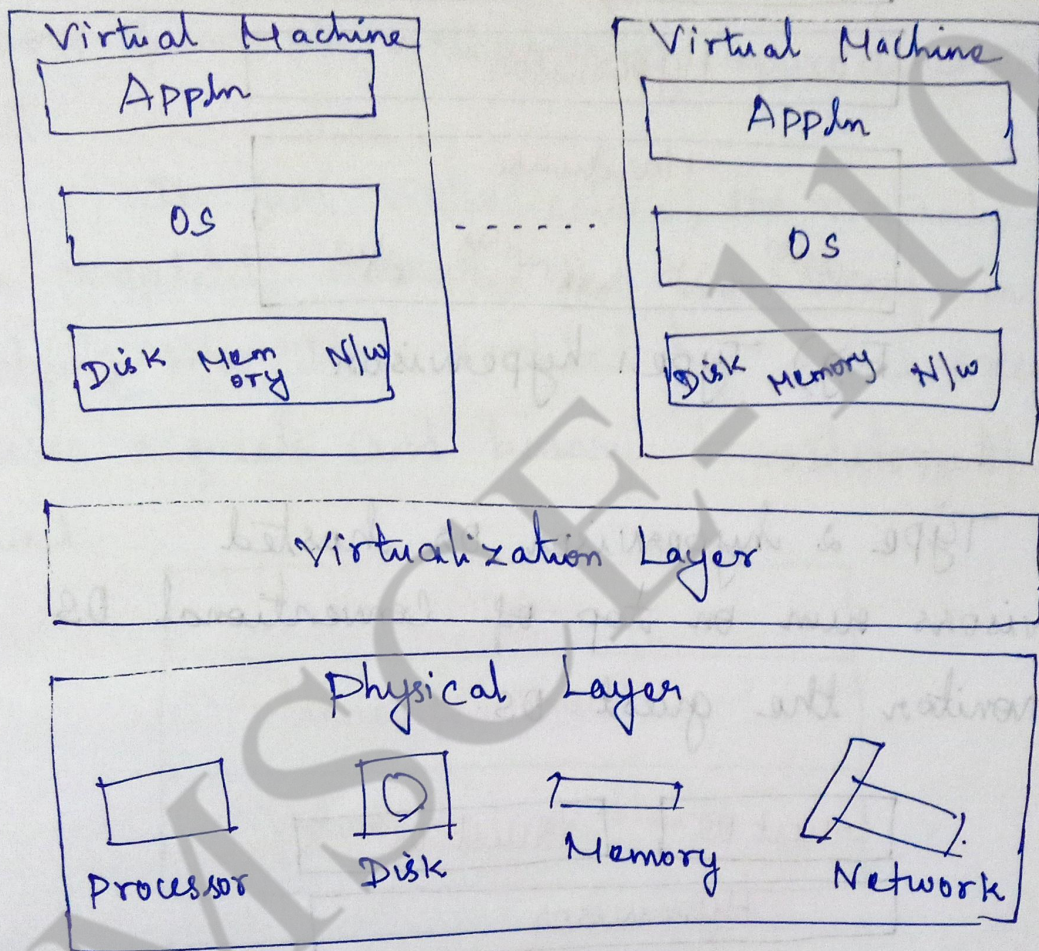


Fig) Virtualization Architecture

There are two types of hypervisor

Type 1, Type 2

Type 1 hypervisors:

Type 1 hypervisors or the native

hypervisors run directly on the Host H/w and control the H/w and monitor the guest OS.

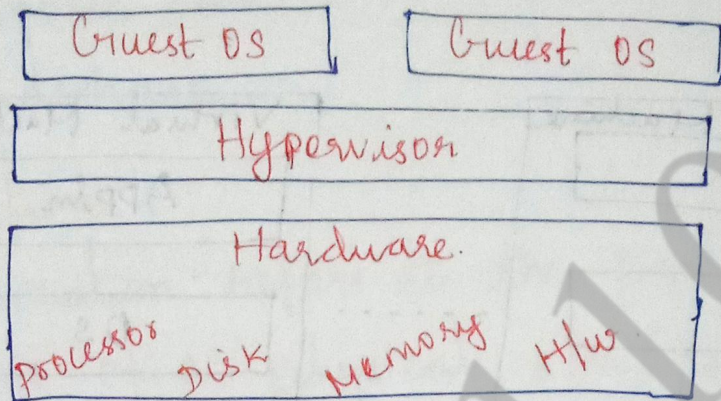


Fig) Type 1 hypervisor

Type 2 hypervisor:

Type 2 hypervisor or hosted hypervisors run on top of conventional OS and monitor the guest OS.

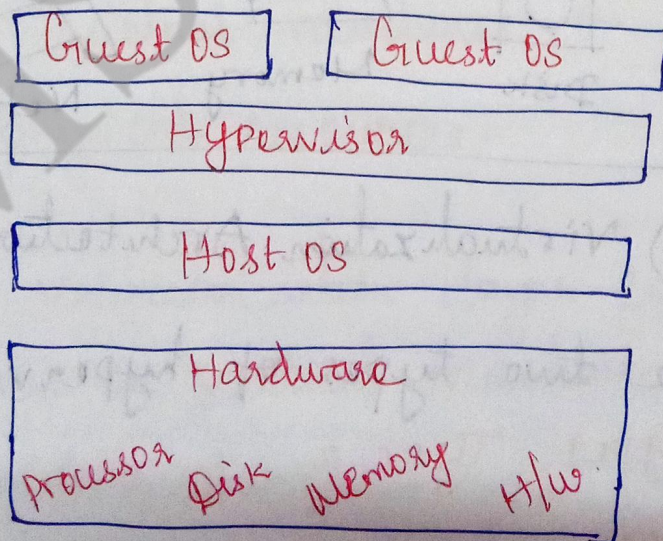


Fig) Type 2 hypervisor

Guest OS.

A guest OS is an OS that is installed in a virtual machine in addition to the host or main OS. In virtualization, the guest OS can be different from the host OS.

Various forms of virtualization approaches exist.

1. Full virtualization:

In full virtualization, the virtualization layer completely decouples the guest OS from the underlying H/w. It is enabled by direct execution of user requests and binary translation of OS request.

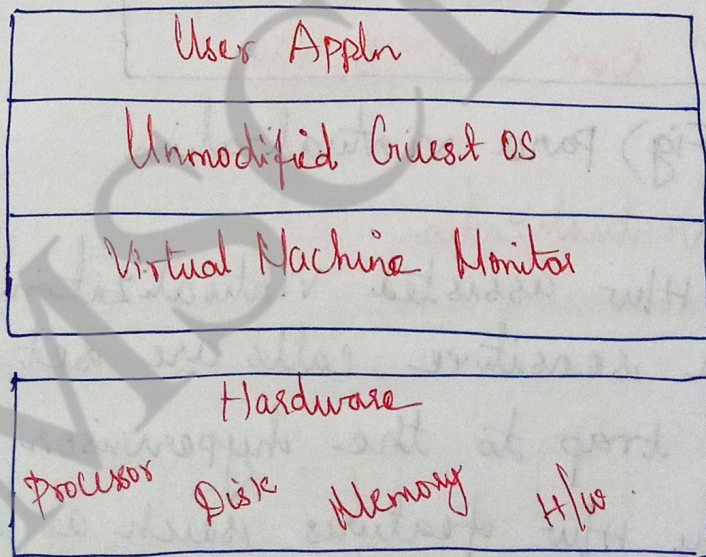


Fig) Full virtualization

2. para virtualization:

The guest OS is modified to enable communication with the hypervisor to improve

performance and efficiency. The guest OS kernel is modified to replace non-virtualizable instructions with hypercalls that communicate directly with the virtualization layer hypervisor.

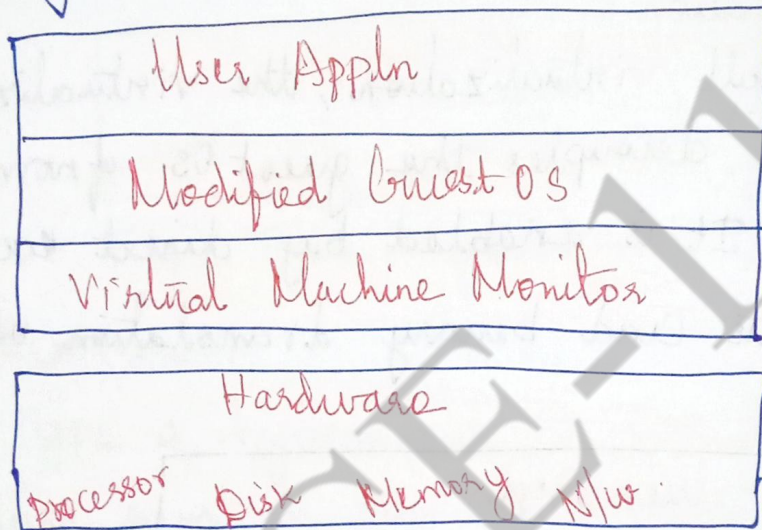


Fig) Para virtualization

3. Hardware Virtualization:

In H/w assisted virtualization, privileged and sensitive calls are set to automatically trap to the hypervisor and is enabled by H/w features such as Intel Virtualization Technology (VT-x) and AMD's AMD-V. There is no need for either binary translation or para virtualization.

Load Balancing

Load balancing distributes workloads across multiple servers to meet the app's workload. The goals of load balancing techniques are to achieve maximum utilization of resources, minimizing the response time, maximizing throughput.

Commonly used load balancing algs include.

1. round robin.

The servers are selected one by one to serve the incoming requests in a non-hierarchical circular fashion with no priority assigned to a specific server.

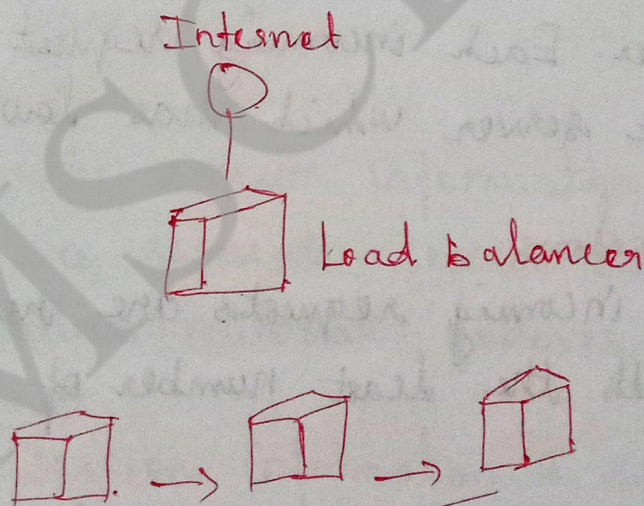


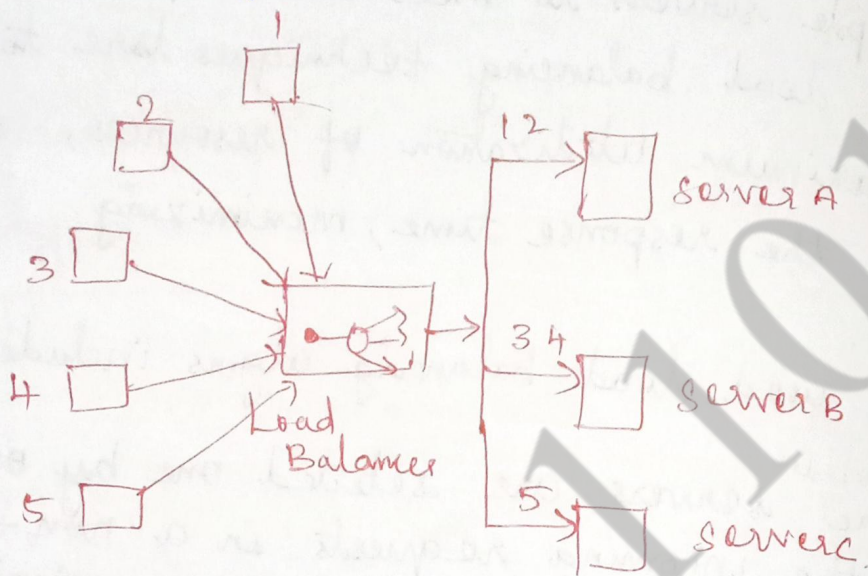
Fig) Round Robin Load balancing.

2. weighted round robin:

Servers are assigned some weights.

The incoming requests are proportionally routed

routed using static or dynamic ratio of respective weights.



3. Low Latency:

The load balancer monitors the latency of each server. Each incoming request is routed to the server which has lowest latency.

4. Least Connections:

The incoming requests are routed to the server with the least number of connections.

5. Priority:

The incoming traffic is routed to the highest priority server as long as the server is available.

6. Overflow:

Similar to priority load balancing.

When the incoming requests to highest priority server overflow, the requests are routed to a lower priority server.

Maintaining the state or the information of session is important. Three persistence approach are described below:

Sticky sessions:

All the requests belonging to a user session are routed to the same server.

Drawback: if the server fails all the session belonging to server are lost, no automatic failover possible.

session database:

All the session information is stored externally in a separate session database.

It allows automatic failover.

Browser Cookies:

The session information is stored on the client side in the form of browser cookies.

URL re-writing:

URL re-write Engine stores the session info by modifying the URLs on the client side.

Scalability and Elasticity:

Modern web Appln have multiple tiers of deployment with varying number of servers in each tier. Capacity planning involves determining the right sizing of each tier of the deployment of an Appln in terms of the number of resources and the capacity of each resources.

When the workloads of Appln increase, the traditional approaches have been either to scale up or scale out.

Scaling out involves addition of more resources of the same type.

Scaling up involves upgrading the H/w resources.

Replication:

Replication is used to create and maintain multiple copies of data in the cloud.

Cloud based data replication approaches

provide replication of data in multiple

locations, automated recovery, low recovery point

Objective (RPO) and low recovery time objective

(RTO).

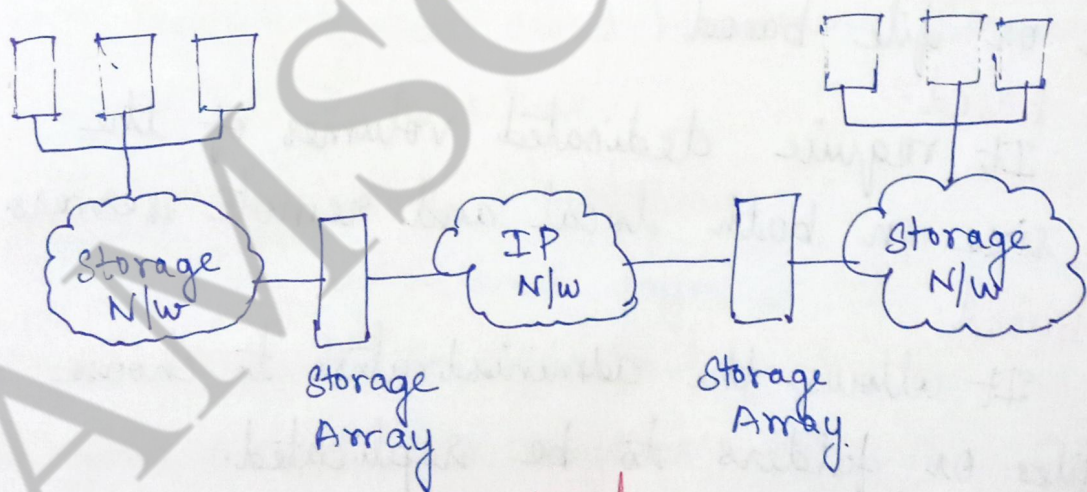
There are three types of replication approaches described as follows,

1. Array based Replication:

Array based replication uses compatible storage arrays to automatically copy data from a local storage array to a remote storage array. It works in heterogeneous environments with different OS. Array based replication uses N/w Attached Storage (NAS) or SAN (Storage Area Network).

Drawback:

It requires similar arrays at local and remote locations, Higher Cost



ii) Network based Replication:

N/w based replication uses an appliance that sits on the N/w and intercepts

packets that are sent from hosts and storage arrays. The intercepted packets are replicated to a storage location.

Benefits :

It supports heterogeneous environments and requires a single point of mgmt.

iii) Host-based Replication:

Host based replication runs on standard servers and uses s/w to transfer data from a local to remote location.

Host based replication can either be block based or file based.

Block based:

It require dedicated volumes of the same size on both local and remote servers.

File based:

It allows the administrators to choose the files or folders to be replicated.

It requires less storage as compared to block based replication.

Monitoring:

Cloud resources can be monitored by monitoring services provided by the cloud service providers. Monitoring services allow cloud users to collect and analyze the data on various monitoring metrics.

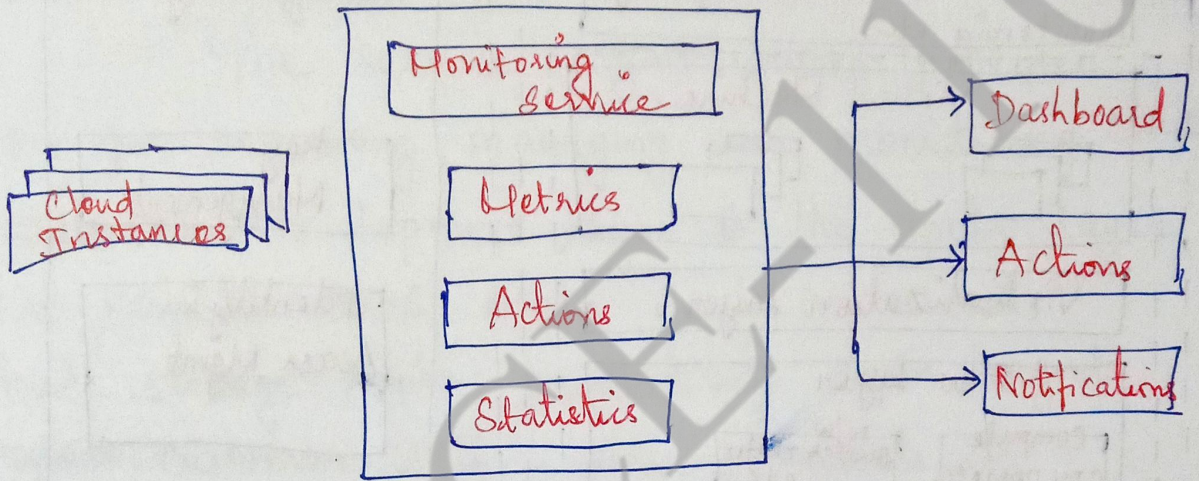


Fig) Typical cloud Monitoring Service Architecture

Cloud Services and Platforms:

The bottom most layer in the cloud reference model is the infrastructure and facilities layer that includes the physical infrastructure such as data center facilities, electrical and mechanical equipment.

On top of the H/w layer that includes

physical compute, n/w and storage H/w.

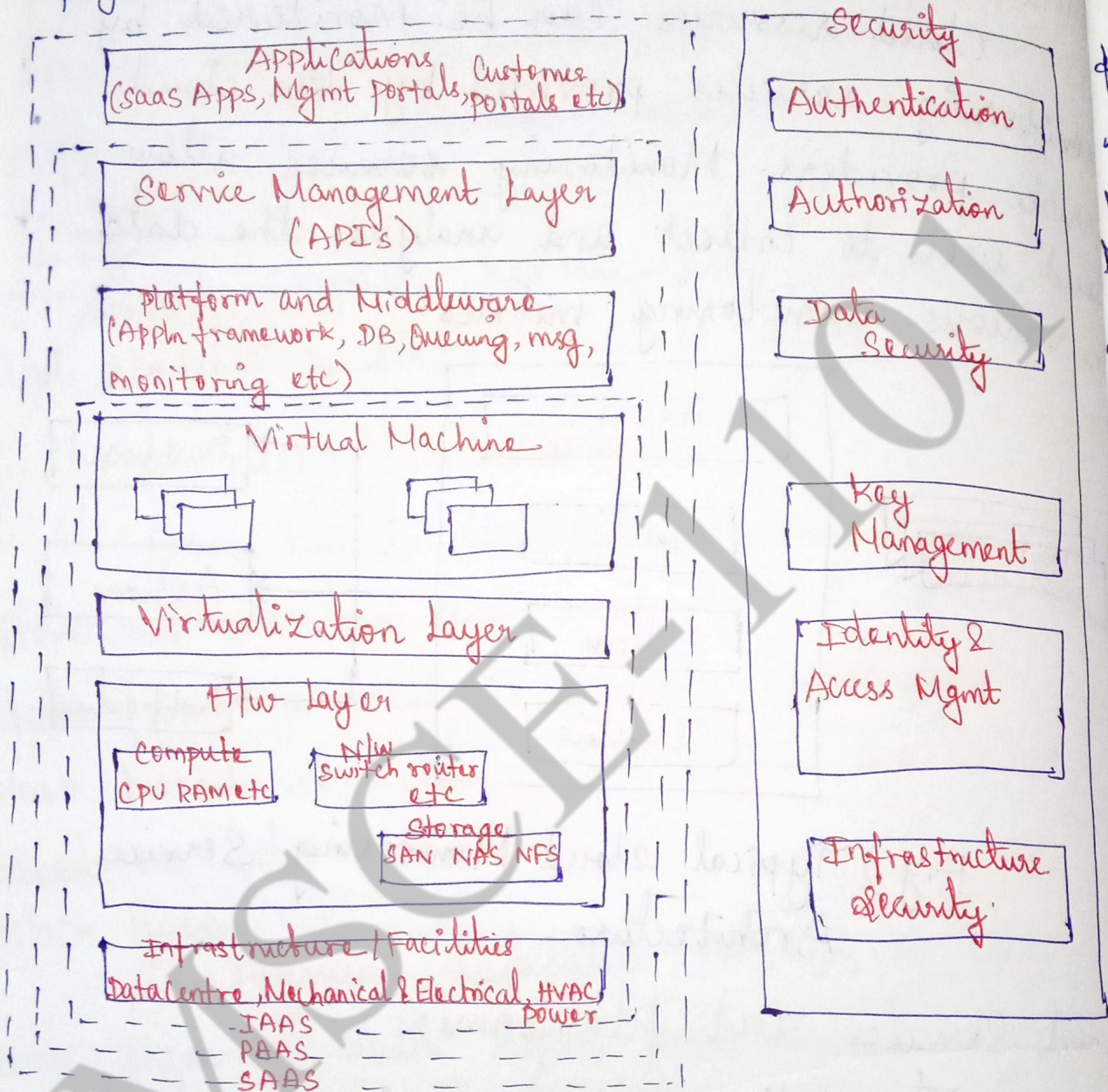


Fig) Cloud reference Model

On top of the H/w layer the virtualization layer partitions the physical H/w resources into multiple virtual resources that enables pooling of resources.

The computing services are delivered in the

form of virtual machines (VMs) along with the storage and I/O resources. The platform and middleware layer builds upon the IaaS layer below and provides standardized stacks of services such as database service, monitoring services, analytics services.

The service mgmt layer provides APIs for requesting, managing and monitoring cloud resources. The topmost layer is the appln layer that includes SaaS Appln such as email, cloud storage Appln, productivity Appln, mgmt portals, customer self-service portals etc.

Compute Services:

Compute services provide dynamically scalable compute capacity in the cloud. Virtual machines can be created from standard images provided by the cloud service provider (eg: Ubuntu image, window image etc.) or custom image created by the users.

image is a machine template that contains a I/O configuration [OS, Appln server, and Appln].