CS3551 DISTRIBUTED COMPUTING L T P C 3 0 0 3

UNIT I INTRODUCTION 8

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 10

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 10

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 10

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 97

UNIT V CLOUD COMPUTING 7

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

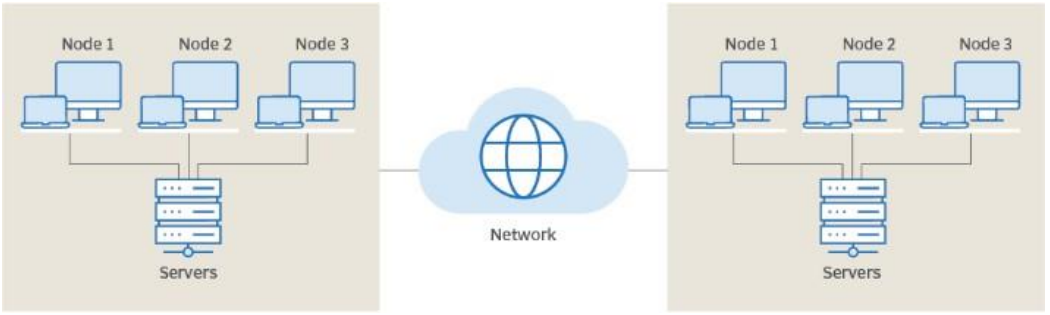
UNIT I - INTRODUCTION

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.

DISTRIBUTED SYSTEM- DEFINITION

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

  A distributed system is the one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages each having its own memory and operating system.



Characteristics of distributed system:

- It is a collection of autonomous processors communicating over a communication network have the following features:
- **No common physical clock.**
- **No shared memory.**
  - Hence it requires message-passing for communication and implies the absence of common physical clock.
  - Distributed system provides the abstraction of common address space via distributed shared memory abstraction.
- **Geographical separation**
  - The geographically wider apart processors are the representative of a distributed system i.e.; it may be in wide-area network (WAN) or the network/cluster of workstations (NOW/COW) configuration connecting processors on a

LAN

- NOW configuration is the low-cost high-speed off-the-shelf processors. Example: Google search engine.

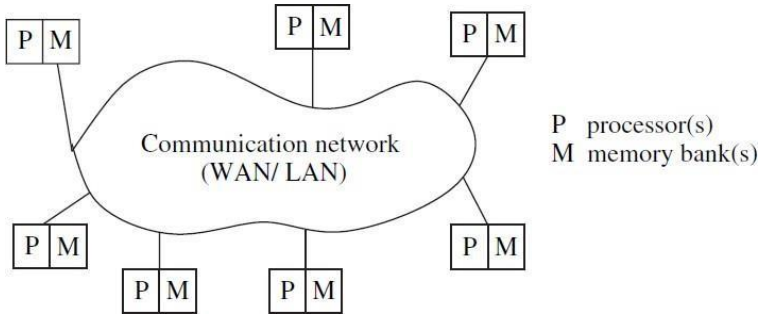
- **Autonomy and heterogeneity**
  - The processors are "loosely coupled" having different speeds and each runs different operating system but cooperate with one another by offering services for solving a problem jointly

RELATION TO COMPUTER SYSTEM COMPONENTS

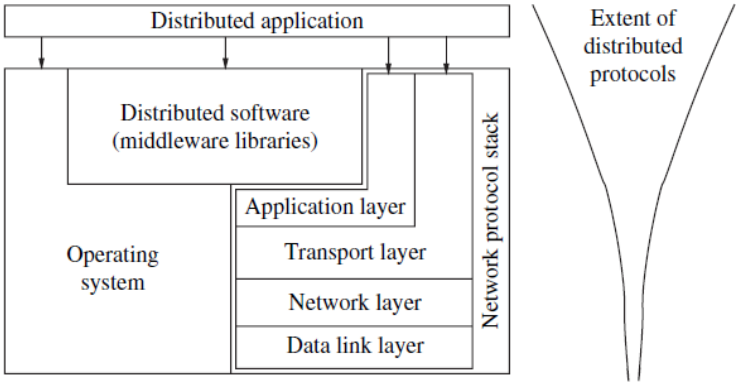
- As shown in the Figure 1.1, in distributed system each computer has a memory-processing unit and is connected by a communication network.

Figure 1.2 shows the relationships of software components that run on computers use the local operating system and network protocol stack for functioning.

- Distributed software is also termed as middleware.
- A distributed execution is the execution of processes across the distributed system to collaboratively achieve a common goal which is also termed a computation or a run.
- A distributed system follows a layered architecture that reduces the complexity of the system design.
- Middleware hides the heterogeneity transparently at the platform level.



(Fig.1.1 A distributed system connects processors by a communication network)



(Fig.1.2 Interaction of the software components at each processor)

- It is assumed that the middleware layer does not contain the application layer functions like http, mail, ftp, and telnet.
- User program code includes the code to invoke libraries of the middleware layer to support the reliable and ordered multicasting.
- There are several standards such as
- Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA)
- RPC software
  - Sends a message across the network to invoke the remote procedure.
  - Waits for a reply.
  - After which the procedure call completes from the program perspective that invoked it.
- Some of the commercial versions of middleware often in use are CORBA, DCOM (distributed component object model), Java, and RMI (remote method invocation), message-passing interface (MPI).

MOTIVATION

The motivation of using a distributed system is because of the following requirements:

1. Inherently distributed computations

Applications like money transfer in banking require the computation that is inherently distributed.

2. Resource sharing

Resources like peripherals, databases, data (variable/files) cannot be fully replicated at all the sites. Further, they can't be placed at a single site as it leads to the bottleneck problem. Hence, such resources are distributed across the system.

3. Access to geographically remote data and resources

As the data may be too large and sensitive it cannot be replicated. Example, payroll data. Hence stored at a central server (like super computers) this can be queried using remote login. Advances in mobile devices and the wireless technology have proven the importance of distributed protocols and middleware.

4. Enhanced reliability

A distributed system has provided increased reliability by the replicating resources and executions in geographically distributed a system which does not crash/malfunction at the same time under normal circumstances.

Reliability is defined in the aspect of

- Availability, i.e., the resource should be accessible at all times;
- Integrity, i.e., the value/state of the resource must be correct, in the face of concurrent access from multiple processors,
- Fault-tolerance, i.e., the ability to recover from system failures.

5. Increased performance/cost ratio

By resource sharing and accessing remote data will increase the performance/cost ratio. The distribution of the tasks across various computers provides a better performance/cost ratio, for example in NOW configuration.

In addition to meeting the above requirements, a distributed system also offers the following advantages:

6. Scalability

As the processors are connected by a wide-area network, adding more processors does not impose a bottleneck for communication network.

7. Modularity and incremental expandability

Heterogeneous processors can be easily added without affecting the performance, as processors runs the same middleware algorithms. Similarly, existing processors can be easily replaced by other processors.

MESSAGE-PASSING SYSTEMS VERSUS SHARED MEMORY SYSTEMS

Introduction:

- In Shared memory systems there is a (common) shared address space throughout the system.
- Communication among processors takes place via shared data variables, and control variables for synchronization (Semaphores and monitors) among the processors.
- If a shared memory is distributed then it is called distributed shared memory.
- All multicomputer (NUMA as well as message-passing) systems that do not have a shared address space hence communicate by message passing.

Emulating message-passing on a shared memory system (MP →SM**)**

- The shared address space is partitioned into disjoint parts; one part being assigned to each processor.
- "Send" and "receive" operations are implemented for writing to and

reading from the destination/sender processor's address space, respectively.

- Specifically, a separate location is reserved as **mailbox** (assumed to have unbounded in size) for each ordered pair of processes.
- A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox.
- The write and read operations are controlled using synchronization primitives to inform the receiver/sender after the data has been sent/received.

Emulating shared memory on a message-passing system (SM →MP**)**

- This involves use of "send" and "receive" operations for "write" and "read" 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 a "read" by sending a query message.
- As accessing another processor's memory requires send and receive operations, this emulation is expensive.
- In a MIMD message-passing multicomputer system, each "processor" may be a tightly coupled multiprocessor system with shared memory. Within the multiprocessor system, the processors communicate via shared memory.
- Between two computers, the communication by message passing are more suited for wide-area distributed systems.

PRIMITIVES FOR DISTRIBUTED COMMUNICATION

Blocking/non-blocking, synchronous/asynchronous primitives

- Message send and receive communication primitives are denoted Send() and Receive(), respectively.
- A Send primitive has at least two parameters the destination, and the buffer in the user space, containing the data to be sent.
- a Receive primitive has at least two parameters – the source of the data, and the user buffer into which the data is to be received.
- There are two ways of sending data while Send is invoked – the buffered option and the unbuffered option.
- The buffered option - copies the data from user buffer to kernel buffer. The data later gets copied from kernel buffer onto the network.
- The unbuffered option - the data gets copied directly from user buffer onto the network.
- For Receive, buffered option is required as the data has already

arrived when the primitive is invoked, and needs a storage place in the kernel.

The following are some definitions of blocking/non-blocking and synchronous/ asynchronous primitives:

- **Synchronous primitives** 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 other corresponding Receive primitive has also been completed. The processing for the Receive primitive completes when the data to be received is copied into the receiver's user buffer.
- **Asynchronous primitives** A Send primitive is said to be asynchronous if control returns back to the invoking process after the data item to be sent is copied out of the user-specified buffer.
- **Blocking primitives** A primitive is blocking if control returns to the invoking process after the processing completes.
- **Non-blocking primitives**
  - A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even 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.
  - For non-blocking primitives, a return parameter of the 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 of the call in two ways.
    1. keep checking (in loop or periodically), if the handle has been flagged or posted.
    2. issue a Wait with a list of handles as parameters which will block until posted.
- The code for a non-blocking Send would look as shown in Figure 1.3.

```
Send(X, destination, handle_k)                    // handle_k is a return parameter
...
...
Wait(handle_1, handle_2, ..., handle_k, ..., handle_m)    // Wait always blocks
```

Figure 1.3 A non-blocking send primitive. When the Wait call returns, at least one of its parameters is posted

- If the processing of the primitive has not completed, the Wait blocks and waits for a signal to wake it up.
- When the processing for the primitive completes, the communication

subsystem software sets the value of handlek and wakes up (signals) any process with a Wait call blocked on this handlek. This is called posting the completion of the operation.

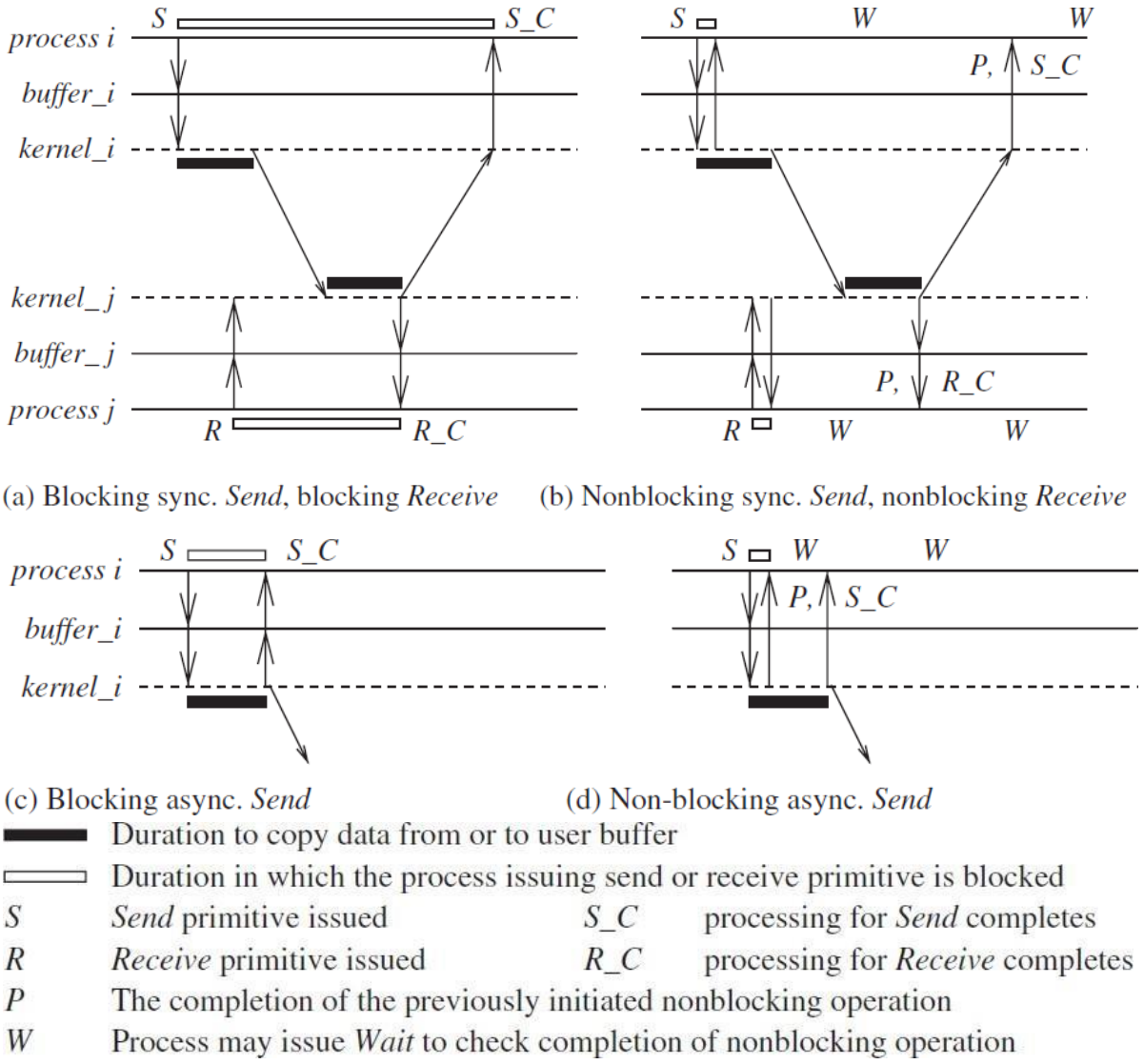


Figure 1.4 Blocking/non-blocking and synchronous/asynchronous primitives. Process Pi is sending and process Pj is receiving. (a) Blocking synchronous Send and blocking (synchronous) Receive. (b) Non-blocking synchronous Send and nonblocking (synchronous) Receive. (c) Blocking asynchronous Send. (d) Non- blocking asynchronous Send.

- Here, three time lines are shown for each process: (1) for the process execution, (2) for the user buffer from/to which data is sent/received, and (3) for the kernel/communication subsystem.
- **Blocking synchronous Send** : The data gets copied from user buffer to 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 it copies the data from user buffer to kernel buffer are initiated.
  - A parameter in non-blocking call gets set with the handle of a location that a user process can check later for the completion of synchronous send operation.
- **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. For the unbuffered option, until the data is copied from the user's buffer to the network.
- **Non-blocking asynchronous Send**
  - Send is blocked until the transfer of data from the user's buffer to the kernel buffer is initiated. For the unbuffered option, it is blocked until data gets transferred from user's buffer to network is initiated.
  - Control returns to the user process as soon as this transfer is initiated, and a parameter in non-blocking call gets set with the handle to check later using Wait operation for the completion of the asynchronous Send operation.
- **Blocking Receive**
  It 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**
  - It 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.
  - The user process can check for the completion of the non-blocking Receive by invoking the Wait operation on the returned handle.
- The non-blocking asynchronous Send is useful when a large data item is being sent because it allows the process to perform other instructions in parallel with the completion of the Send.
- The non-blocking synchronous Send also avoids the large delays for handshaking, particularly when the receiver has not yet issued the Receive call.
- The non-blocking Receive is useful when a large data item is being received and/or when the sender has not yet issued the Send call, because it allows the process to perform other instructions in parallel with the completion of the Receive.

Processor synchrony

- Processor synchrony indicates that all the processors execute in lock-step with their clocks synchronized.
- As this synchrony difficult in distributed system, a large granularity of code, is termed as a step, the processors are synchronized.
- This synchronization ensures 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.

1. Many commercial software products (banking, payroll applications) use proprietary primitive libraries supplied with software vendors (i.e., IBM CICS software).
2. The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community
3. Commercial software is often written using remote procedure calls (RPC) mechanism in which procedures that resides across the network are invoked transparently to the user, in the same manner that a local procedure is invoked.
4. socket primitives or socket-like transport layer primitives are invoked to call the procedure remotely.
5. There exist many implementations of RPC like Sun RPC, and distributed computing environment (DCE) RPC.
6. "Messaging" and "streaming" are two other mechanisms for communication.
7. For object based software, libraries for remote method invocation (RMI) and remote object invocation (ROI) is used.
8. CORBA (common object request broker architecture) and DCOM (distributed component object model) are two other standardized architectures with their own set of primitives.

SYNCHRONOUS VERSUS ASYNCHRONOUS EXECUTIONS

- An asynchronous execution is an execution in which (i) there is no processor synchrony and there is no bound on the drift rate of processor clocks, (ii) message delays (transmission + propagation times) are finite but unbounded, and (iii) there is no upper bound on the time taken by a process to execute a step.

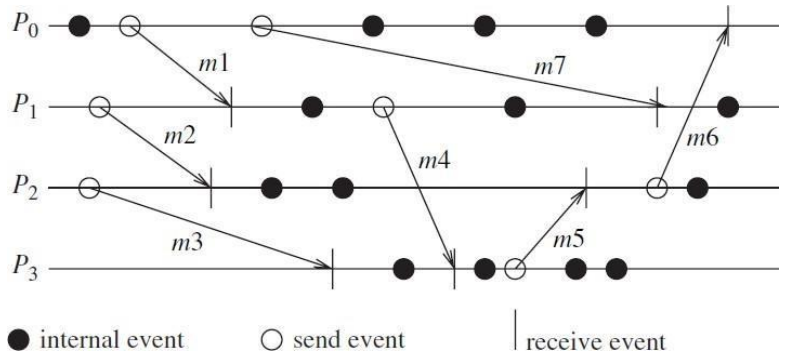


Figure 1.5 An example timing diagram of an asynchronous execution in a message-passing system.

- A synchronous execution is an execution in which (i) processors are synchronized and the clock drift rate between any two processors is bounded, (ii) message delivery (transmission + delivery) times are such that they occur in one logical step or round, and (iii) there is a known upper bound on the time taken by a process to execute a step.

If processors are allowed to have an asynchronous execution for a period of time and then they synchronize, then the granularity of the synchrony is coarse. This is really a virtually synchronous execution, and the abstraction is sometimes termed as virtual synchrony.

Ideally, many programs want the processes to execute a series of instructions in rounds (also termed as steps or phases) asynchronously, with the requirement that after each round/step/phase, all the processes should be synchronized and all messages sent should be delivered.

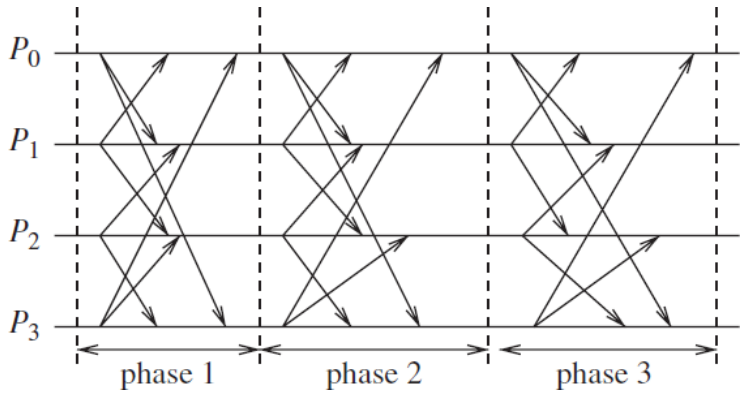


Figure 1.6 An example of a synchronous execution in a message-passing system. All the messages sent in a round are received within that same round.

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

An asynchronous program can be emulated on a synchronous system as 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 (written for a synchronous system) can be emulated on an asynchronous system using a tool called synchronizer.

Emulations

Using the emulations shown, any class can be emulated by any other. 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. Likewise, if a problem is solvable in A, it is also solvable in B.

Hence, all four classes are equivalent in terms of "computability" i.e., what can and cannot be computed – in failure-free systems.

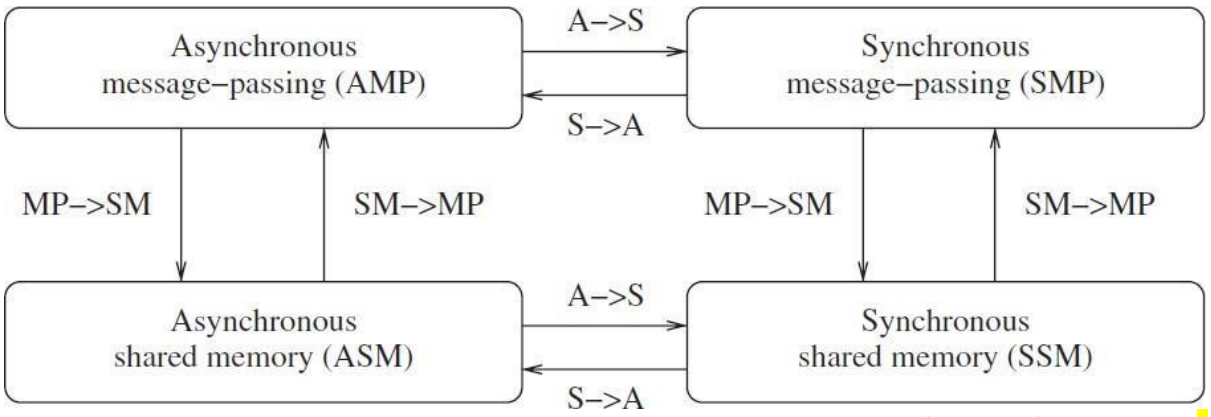
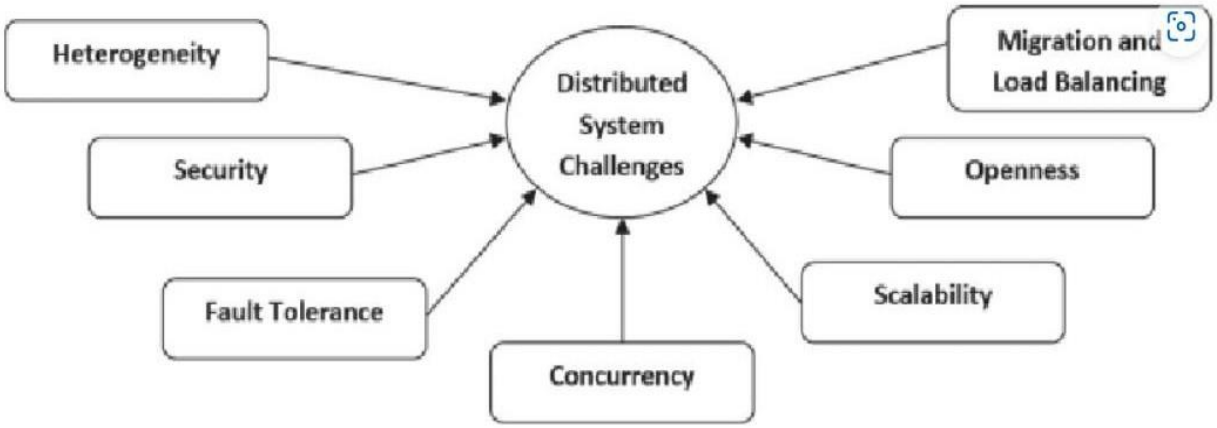


Figure 1.7 Emulations among the principal system classes in a failure-free system

DESIGN ISSUES AND CHALLENGES



- The important design issues and challenges is categorized as
  - related to systems design and operating systems design, or
  - component related to algorithm design, or
  - emerging from recent technology

(i) Distributed systems challenges from a system perspective

The following functions must be addressed when designing and building a distributed system:

- **Communication** This task involves designing appropriate mechanisms for communication among the processes in the network. Example: remote procedure call (RPC), remote object invocation (ROI), etc.
- **Processes** Some of the issues involved are: management of processes and threads at clients/servers; code migration; and the 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. Naming in mobile systems provides additional challenges.
- **Synchronization** Mutual exclusion is an example of synchronization, other forms of synchronization are leader election and synchronizing physical clocks.
- **Data storage and access** Schemes for data storage, and implicitly for accessing the data in a fast and scalable manner across the network are important for efficiency.
  - **Consistency and replication** To avoid bottlenecks, to provide fast access to data, and to provide scalability, replication of data objects is highly desirable. This leads to issues of managing the replicas, and dealing with consistency among the replicas/caches in a distributed setting.
- **Fault tolerance** to maintaining correct and efficient operation in spite of any failures of links, nodes, and processes. Process resilience, reliable communication, distributed commit, and check pointing and recovery are some of the fault-tolerance mechanisms.
- **Security** Distributed systems security involves various aspects of cryptography, secure channels, access control, authorization, and secure group management.
- **Applications Programming Interface (API) and transparency** The API for communication and other services for the ease of use and wider adoption of distributed systems services by non-technical users.
- Transparency deals with hiding the implementation policies from user, and is classified as follows:
  - Access transparency: hides differences in data representation on different systems and provides uniform operations to access system resources.
  - Location transparency: makes the locations of resources transparent to the users.
  - Migration transparency: allows relocating resources without changing names.
  - Concurrency transparency deals with masking concurrent use of shared resources for user.

- Failure transparency: refers to the system being reliable and fault-tolerant.

- **Scalability and modularity** The algorithms, data (objects), and services must be as distributed as possible. Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.

(ii) Algorithmic challenges in distributed computing

- The key algorithmic challenges in distributed computing is as summarized below:
- **Designing useful execution models and frameworks**
  - The interleaving model and partial order model are two widely adopted models of distributed system executions are useful for operational reasoning and the design of distributed algorithms.
  - The input/output automata model and the TLA (temporal logic of actions) are two other examples of models that provide different degrees of infrastructure for proving the correctness of distributed programs.
- **Dynamic distributed graph algorithms and distributed routing algorithms**
- The distributed system is modeled as a distributed graph.
- The graph algorithms form the building blocks for a large number of higher level communication, data dissemination, object location, and object search functions.
- The algorithms need to deal with dynamically changing graph characteristics, such as varying link loads, user-perceived latency, congestion in the network in a routing algorithm. Hence, the design of efficient distributed graph algorithms is important.
- **Time and global state in a distributed system**
- The challenges pertain to providing accurate physical time, and to providing a variant of time, called logical time.
- Logical time is relative time, and can (i) capture the logic and inter-process dependencies within the distributed program, and also (ii) track the relative progress at each process.
- Observing the global state of the system (across space) also involves the time dimension for consistent observation.

Synchronization/coordination mechanisms

- The processes must be allowed to execute concurrently, except when they need to synchronize to exchange information, i.e., communicate about shared data.
- Synchronization is essential for the distributed processes to overcome the limited observation of the system state from the viewpoint of any one process.
- The synchronization mechanisms is viewed as resource and concurrency

management mechanisms to control the behavior of the processes.

- some examples of problems requiring synchronization:
  - **Physical clock synchronization** Physical clocks ususally 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 – called a leader process which is necessary for many distributed algorithms to initiate some action like a broadcast or collecting the state of the system.
  - **Mutual exclusion** This is clearly a synchronization problem because access to the critical resource(s) has to be coordinated.
  - **Deadlock detection and resolution**
    - Deadlock detection needs coordination to avoid duplicate work, and
    - deadlock resolution needs coordination to avoid unnecessary aborts of processes.
  - **Termination detection** This requires cooperation among the processes to detect the specific global state of quiescence.
  - **Garbage collection** Garbage refers to objects that are no longer in use and that are not pointed to by any other process. Detecting garbage requires coordination among the processes.

Group communication, multicast, and ordered message delivery

- A group is a collection of processes on an application domain.
- Specific algorithms need to be designed to enable efficient group communication and group management wherein processes can join and leave groups dynamically, or even fail.
- When multiple processes send messages concurrently, different recipients may receive the messages in different orders that violates the semantics of distributed program. Hence, formal specifications for ordered delivery need to be formulated.

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, and are useful for applications like debugging, sensing the environment, and in industrial process control hence for monitoring such predicates are important.
- An important paradigm for monitoring distributed events is that of event streaming Distributed program design and verification tools
- Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering.
- Designing mechanisms to achieve these design and verification goals is a challenge.

Debugging distributed programs

- debugging distributed programs is much harder because of the concurrency in actions and uncertainty due to the large number of possible executions defined by the interleaved concurrent actions.
- Adequate debugging mechanisms and tools need to be designed to meet this challenge.

Data replication, consistency models, and caching

- Fast access to data and other resources requires replication in the distributed system.
- Managing such replicas in the face of updates introduces consistency problems among the replicas and cached copies.
- Additionally, placement of the replicas in the systems is also a challenge.

World Wide Web design – caching, searching, scheduling

- The Web is an example of widespread distributed system with direct interface to the end user where the operations are read-intensive on most objects.
- The issues of object replication and caching has to be considered.
- Example: Prefetching can be used for subscribing of Content Distribution Servers.
- Minimizing response time to minimize user perceived latencies is an important challenge.
- Object search and navigation on the web are resource-intensive. Designing mechanisms to do this efficiently and accurately is a great challenge.

Distributed shared memory abstraction

- A shared memory abstraction deals only with read and write operations, and no message communication primitives.
- But the middleware layer abstraction has to be implemented using message-passing.
- Hence, in terms of overheads, the shared memory abstraction is not less expensive.
- **Wait-free algorithms**
  - Wait-freedom is defined as the ability of a process to complete its execution irrespective of the actions of other processes.
  - While wait-free algorithms are highly desirable and expensive hence it is a a challenge.
- **Mutual exclusion**
  - the Bakery algorithm and semaphores are used for mutual exclusion in a multiprocessing (uniprocessor or multiprocessor) shared memory setting.
- **Register constructions**
  - emerging technologies like biocomputing and quantum computing alter the present foundations of computer "hardware" design assumptions of memory access of current systems that are exclusively based on semiconductor technology and the von Neumann architecture.

- The study of register constructions deals with the design of registers from scratch, with very weak assumptions on the accesses allowed to a register.
- This forms a foundation for future architectures that allow concurrent access even to primitive units of memory (independent of technology) without any restrictions on the concurrency.

- **Consistency models**
  - For multiple copies of a variable/object, varying degrees of consistency among the replicas can be allowed.
  - a strict definition of consistency in a uniprocessor system would be expensive to implement in terms of high latency, high message overhead, and low concurrency.
  - But still meaningful models of consistency are desirable.

Reliable and fault-tolerant distributed systems

- A reliable and fault-tolerant environment has multiple requirements and aspects and addressed using various strategies:

➢ **Consensus algorithms**
  - It relies on message passing, and the recipients take actions based on the contents of the received messages.
  - It allows correct functioning of processes to reach agreement among themselves in spite of the existence of some malicious (adversarial) processes whose identities are not known to the correctly functioning processes.

➢ **Replication and replica management**
  - Replication ie., having backup servers is a classical method of providing fault- tolerance.
  - The triple modular redundancy (TMR) technique is used in software as well as hardware installations.

- **Voting and quorum systems** Providing redundancy in the active (e.g., processes) or passive (e.g., hardware resources) 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**
  - For distributed databases, the ACID properties of the transaction (atomicity, consistency, isolation, durability) need to be preserved in distributed setting.
  - The "transaction commit" protocols is a fairly mature area that can be applied for guarantees on message delivery in group communication in the presence of failures.
- **Self-stabilizing systems**
  - All system executions have associated good (or legal) states and bad

(or illegal) states; during correct functioning, the system makes transitions among the good states.
  - Faults, internal or external to the program and system, may cause a bad state to arise in the execution.
  - A self-stabilizing algorithm is any algorithm that is guaranteed to eventually take the system to a good state even if a bad state were to arise due to some error.
  - Designing efficient self-stabilizing algorithms is a challenge.
- **Check pointing and recovery algorithms**
  - Check pointing involves 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.
  - Check pointing in a distributed environment is difficult because if the checkpoints at the different processes are not coordinated.
- **Failure detectors**
  - In asynchronous distributed systems there is no bound on time for message transmission.
  - Hence, it is impossible to distinguish a sent-but-not-yet-arrived message from a message that was never sent ie., alive or failed.
  - Failure detectors represent a class of algorithms that 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 goal of load balancing is to gain higher throughput, and reduce the user perceived latency.
  - Need: high network traffic or high request rate causing the network connection to be a bottleneck, or high computational load and to service incoming client requests with the least turnaround time.
  - 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 a redistribution 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 is important for mission-critical applications, to accomplish the task execution on schedule.
  - The problem becomes more challenging in a distributed system where a global view of the system state is absent.
  - message propagation delays are hard to control or predict, which makes meeting real- time guarantees that are inherently dependent on

communication among the processes harder.

- **Performance**
  - Although high throughput is not the primary goal of using a distributed system, achieving good performance is important.
  - In large distributed systems, network latency and access to shared resources can lead to large delays which must be minimized.

(iii) Applications of distributed computing and newer challenges

- **Mobile systems**

  Mobile systems typically use wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium.

  Characteristics and issues are
  - communication : range and power of transmission,
  - Engineering: battery power conservation, interfacing with wired Internet, signal processing and interference.
  - Computer science Perspective: routing, location management, channel allocation, localization and position estimation, and the overall management of mobility.
- There are two architectures for a mobile network.
  1. base-station approach are cellular approach,
  - Where a cell is the geographical region within range of a static and powerfulbase transmission station is associated with base station.
  - All mobile processes in that cell communicate via the base station.
  2. ad-hoc network approach
  - where there is no base station
  - All responsibility for communication is distributed among the mobile nodes,
  - Mobile nodes participate in routing by forwarding packets of other pairs of communicating nodes.
  - Hence complex and poses many challenges.
- **Sensor networks**
  - A sensor is a processor with an electro-mechanical interface i.e., capable of sensing physical parameters like temperature, velocity, pressure, humidity, and chemicals.
  - Recent developments in cost-effective hardware technology made very large low-cost sensors.
  - In event streaming, the streaming data reported from a sensor network differs from the streaming data reported by "computer processes". This limits the nature of information about the reported event in a sensor network.
  - Sensors have to self-configure to form an ad-hoc network, that

creates a new set of challenges like position estimation and time estimation.

Ubiquitous or pervasive computing

- It is a class of computing where the processors embedded in the environment and perform computing appears anytime and everywhere like in sci-fi movies.
- The intelligent home and the smart workplace are some example of ubiquitous environments currently under intense research and development.
- It is a distributed system with wireless communication, sensor and actuator mechanisms. They are self-organizing, network-centric and resource constrained.
- It has small processors operating collectively in a dynamic network. The processors is connected to networks and processes the resources for collating data.

Peer-to-peer computing

- Peer-to-peer (P2P) computing represents computing over an application layer network wherein all interactions among the processors are at a "peer" level.
- all processors are equal and play a symmetric role in computation.
- P2P networks are typically self-organizing, and may or may not have a regular structure to the network.
- No central directories for name resolution and object lookup are allowed.
- key challenges include:

object storage mechanisms, efficient object lookup, and retrieval;
dynamic reconfiguration as the nodes and objects join and leave the network randomly;
anonymity, privacy, and security.

Publish-subscribe, content distribution, and multimedia

- There exists large amount of information, hence there is a greater need to receive and access only information of interest. Such information can be specified using filters.
- In a dynamic environment the information constantly fluctuates (stock prices), there needs to be an efficient mechanism for :
  (i) distributing this information (publish),
  (ii) to allow end users to indicate interest in receiving specific kinds of information (subscribe), and
  (iii) aggregating large volumes of information and filtering based on user's subscription.
- Content distribution refers to multimedia data, where the multimedia data is very large and information-intensive, requires compression, and often requires special synchronization during storage and playback.

Distributed agents

- Agents are software processes or robots that move around the system to do a specific task for which they are specially programmed.
- Agents collect and process information, and can exchange such information with other agents. It cooperate like an ant colony.
- Challenges: coordination mechanisms among the agents, controlling the mobility of the agents, and their software design and interfaces.

Distributed data mining

- Data mining algorithms examine large amounts of data to detect patterns and trends in the data, to mine or extract useful information.
- Example: examining the purchasing patterns of customers to enhance the marketing.
- The mining is done by applying database and artificial intelligence techniques to a data repository.
- In many situations, data is distributed and cannot be collected in a single repository like banking applications where the data is private and sensitive, or in atmospheric weather prediction where the data sets are far too massive to collect and process at a single repository in real-time.
- In such cases, efficient distributed data mining algorithms are required.

Grid computing

- It is the technology that utilizes the idle CPU cycles of machines connected to the network and make it available to others.
- Challenges: scheduling jobs in a distributed environment, to implementing quality of service and real-time guarantees, and security.

Security in distributed systems

Challenges are:

- confidentiality – only authorized processes can access information.
- Authentication – whether the information is from the correct source, identity
- Availability – maintaining allowed access to services despite malicious actions.
- Goal: meet these challenges with efficient and scalable solutions.
- For peer-to-peer, grid, and pervasive computing, these challenges are difficult because of the resource-constrained environment, a broadcast medium, the lack of structure and in the network.

Challenges in Distributed Systems

- A. Heterogeneity - Within a distributed system, we have variety in networks, computer hardware,
- operating systems, programming languages, etc. B. Openness - New services are added to distributed systems. To do that, specifications of components, or at least the interfaces to the components, must be published.
- C. Transparency - One distributed system made to look like a single

computer by concealing the distribution mechanism.

- D. Performance - One of the objectives of distributed systems is achieving high performance while using cheap computers.
- E. Scalability - A distributed system may include thousands of computers. Whether the system works is the question in that large scale.
- F. Failure Handling - One distributed system is composed of many components. That results in high probability of having failure in the system.
- G. Security - Because many stake-holders are involved in a distributed system, the interaction must be authenticated, the data must be concealed from unauthorized users, and so on.
- H. Concurrency - Many programs run simultaneously in a system and they share resources. They should not interfere with each other.

Heterogeneity

A distributed system may be composed of a heterogeneous collection of computers. Heterogeneity arises in the following areas-

networks: Even if the same Internet protocol is used to communicate, the performance parameters may widely vary within the inter-network.

computer hardware: Internal representation of data is different for different processors.

operating systems: The interface for exchanging messages is different from one operating system to another.

programming languages: Characters and data structures are represented differently by different programming languages.

implementations by different developers: Unless common standards are observed, different implementations cannot communicate.

Openness

- Openness means disclosing information: the usage of services provided by remote computers in particular.
- Open systems are easier to extend and reuse.
- By making services open, servers can be used by various clients.
- The clients which use services provided by other servers can extend the services and again provide services to other clients.
- The openness of distributed systems let us add new services and increase availability of services to different clients.

Transparency

Transparency involves not being able to see something, or seeing through it.

Transparency is an important issue to realize the single system image which makes systems as easy to use as a single processor system. e.g. in WWW we can access whatever information by clicking links without knowing whereabouts of the host.

Classification of Transparency

- Access transparency: Data and resources can be used in a consistent way.
- Location transparency: A user cannot tell where resources are located
- Migration transparency: Resources can move at will without changing their names.
- Replication transparency: A user cannot tell how many copies exist.
- Concurrency transparency: Multiple users can share resources automatically.
- Failure transparency: A user does not notice resource failure.
- Performance transparency: Systems are reconfigured to improve performance as loads vary.
- Scaling transparency: Systems can expand in size without changing the system structure and the application programs.

Performance

Fine-grained parallelism: Small programs are executed in parallel.

- Large number of messages.
- Communication overhead decreases the performance gain with parallel processing.

Coarse-grained parallelism:

- Long compute-bound programs executed in parallel.
- Communication overhead is less in this case.

Scalability

Scalability is the issue whether a distributed system works and the performance increases when more computers are added to the system.

The followings are potential bottle-necks in very large distributed systems

- Centralized components: A single mail server for all users.
- Centralized tables: A single on-line telephone book
- Centralized algorithms: Routing based on complete information

Use decentralized algorithms for scalability:

- No machine has complete information about the system state.
- Machines make decisions based only on local information.
- Failure of one machine does not completely invalidate the algorithm.

Reliability

We have high probability to have faulty components in a distributed system because the system includes large number of components.

On the other hand, it is theoretically possible to build a distributed system such that if a machine goes down, the other machine takes over the job.

Reliability has several aspects.

Availability: The fraction of time that the system is available. It can be expressed by the following equation-

R =usable_ time / total_time

Fault tolerance: Distributed systems can hide failures from the users.

Performance

Maximum aggregate performance of the system can be measured in terms of Maximum aggregate floating- point operations.

P=N'C'F'R

- Where P performance in flops, N number of nodes, C number of CPUs, F floating point ops per clock period -FLOP, R clock rate.
- The similar measures with MOP/MIP.
- It is computed as

S = T(1)/T(N)

Scalability

Where T(1) is the wall clock time for a program to run on a single processor.

T(N) is the runtime over N processors.

- A scalability figure close to N means the program scales well.

Scalability metric helps estimate the optimal number of processors for an application.

Utilization

It is calculated as,

U = S(N)/N

- Values close to unity or 100% are ideally sought.

MODEL OF DISTRIBUTED COMPUTATIONS

A distributed system consists of a set of processors that are connected by a communication network. The communication network provides the facility of information exchange among processors. The communication delay is finite but unpredictable. 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. The system can be modeled as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels

A DISTRIBUTED PROGRAM

- A distributed program is composed of a set of n asynchronous processes p_1, p_2,..,p_i,…,p_n that communicate by message passing over the communication network.
- The communication delay is finite and unpredictable. The processes do not share a global memory and communicate by passing messages.
- Let C_{ij} denote the channel from process pi to process pj and
- Let mij denote a message sent by p_i to p_j.
- Process execution and message transfer are asynchronous – ie., a

process does not wait for the delivery of the message to be complete after sending it.

- 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 the state of its local memory and depends upon the context.
- The state of a channel is the set of messages in transit on 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,
  - message receive events.
- Let e_i^x denote the xth event at process p_i. denote send and receive events, respectively.
- The occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state.
- An internal event changes the state of the process at which it occurs.
- A send event changes
  - the state of the process that sends or receives and
  - the state of the channel on which the message is sent.
- An internal event only affects the process at which it occurs.
- The events at a process are linearly ordered by their order of occurrence.

The execution of process p_i produces a sequence of events $e_i^1, e_i^2, \ldots, e_i^x, e_i^{x+1}, \ldots$ and is denoted by $\mathcal{H}_i$:

where h_i is the set of events produced by p_i and binary relation $\rightarrow_i$ defines a linear order on these events. Relation $\rightarrow_i$ expresses causal dependencies among the events of p_i.

- For every message m that is exchanged between two processes, have

$$\text{Send}(m) \rightarrow \text{msgrec}(m)$$

- Relation $\rightarrow$msg defines causal dependencies between send and receive events.
- Fig 2.1 shows a distributed execution using space–time diagram that involves three processes.
- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.

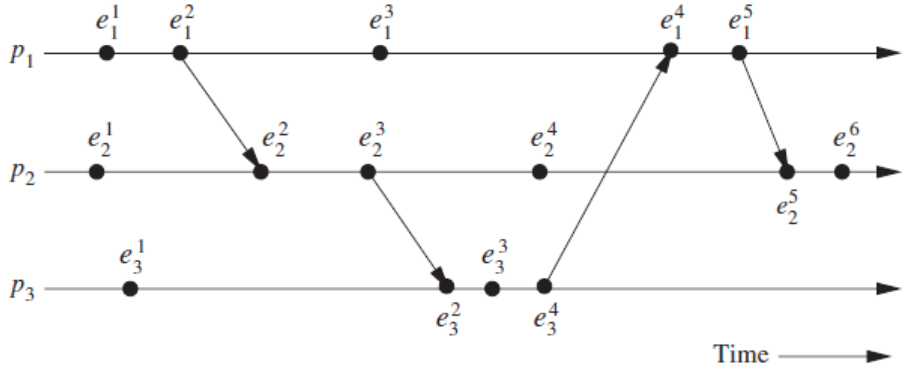


Fig:1.8 The space–time diagram of a distributed execution.

- In this figure, for process p1, the 2nd event is a message send event, the 3rd event is an internal event, and the 4th event is a message receive event.

Causal precedence relation

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let H =Uihi denote the set of events executed in a distributed computation.
- a binary relation on the set H is denoted as →, that expresses causal dependencies between events in the distributed execution.

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

- the relation → is Lamport's "happens before" relation
- For any two events ei and ej, if ei→ej , then event ej is directly or transitively

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

dependent on event ei; in the figure 2.1,

- Note that relation → denotes flow of information in a distributed computation i.e., all information available at ei is accessible at ej
- For any two events ei and ej denotes the fact that event ej does not directly or transitively

$$e_i \not\rightarrow e_j$$

dependent on event ei.

- For example in the figure 1.8:

$e_1^3 \not\rightarrow e_3^3$ and $e_2^4 \not\rightarrow e_3^1$

- Note the following rules:
  - for any two events e_i and e_j, $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$
  - for any two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$.

said to be concurrent and the relation is denoted as $e_i \parallel e_j$. In the execution of Figure 2.1, $e_1^3 \parallel e_3^3$ and $e_2^4 \parallel e_3^1$. Note that relation $\parallel$ is not transitive; that is, $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$. For example, in Figure 2.1, $e_3^3 \parallel e_2^4$ and $e_2^4 \parallel e_1^5$, however, $e_3^3 \not\parallel e_1^5$.

Logical vs. physical concurrency

- Physical concurrency if and only if the events occur at the same instant in

physical time.

- In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other. For example, in Figure 2.1, events in the set $\{e_3^1, e_4^2, e_3^3\}$ are logically concurrent, but they occurred at different instants in physical time.
- Whether a set of logically concurrent events coincide in the physical time or not they does not change the outcome of the computation.

MODELS OF COMMUNICATION NETWORKS

Models of the service provided by communication networks are:

1. In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
2. 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.
3. The "causal ordering" model is based on Lamport's "happens before" relation. A system that supports the causal ordering model satisfies the following property:
   CO: For any two messages mij and mkj,if send(mij) $\rightarrow$ send(mkj) then
   rec(mij) $\rightarrow$ rec(mkj)

- Causally ordered delivery of messages implies FIFO message delivery.
- Note that CO $\subset$ FIFO $\subset$ Non-FIFO.
- Causal ordering model is useful in developing distributed algorithms. Example:replicated database systems, every process that updates a replica must receives updates in the same order to maintain database consistency.

GLOBAL STATE OF A DISTRIBUTED SYSTEM

- The global state GS of a distributed system is a collection of the local states of the processes and the channels is defined as

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

- For a global snapshot, the states of all the components of the distributed system must be recorded at the same instant. This is possible if the local clocks at processes were perfectly synchronized by the processes.

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

$$\forall m_{ij} : send(m_{ij}) \not\in LS_i^{x_i} \Rightarrow m_{ij} \not\in SC_{ij}^{x_i,y_j} \wedge rec(m_{ij}) \not\in LS_j^{y_j})$$

- Basic idea is that a message cannot be received if it was not sent i.e., the state should not violate causality. Such states are called consistent global states.

25

- Inconsistent global states are not meaningful in a distributed system.global state GS consisting of local states
  - $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is inconsistent
  - $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent;

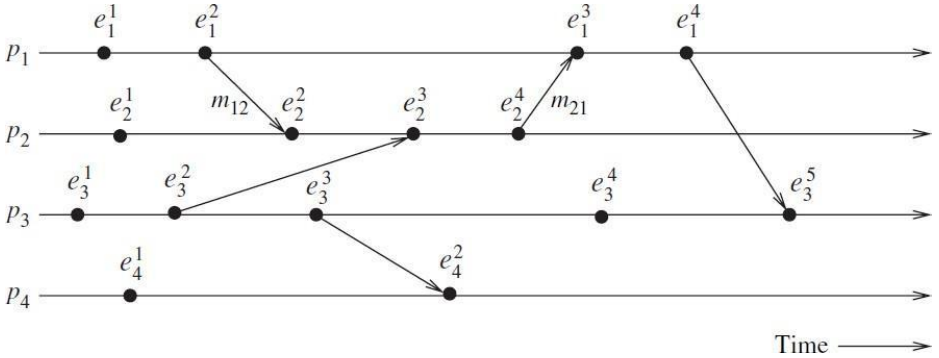


Fig:1.9 The space–time diagram of a distributed execution)

- A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{i,k} SC_{ik}^{y_j,z_k}\}$ is *transitless* iff

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

- A global state is strongly consistent if it is transitless as well as consistent.
  - $\{LS_1^2, LS_2^3, LS_3^4, LS_4^2\}$ is strongly consistent.

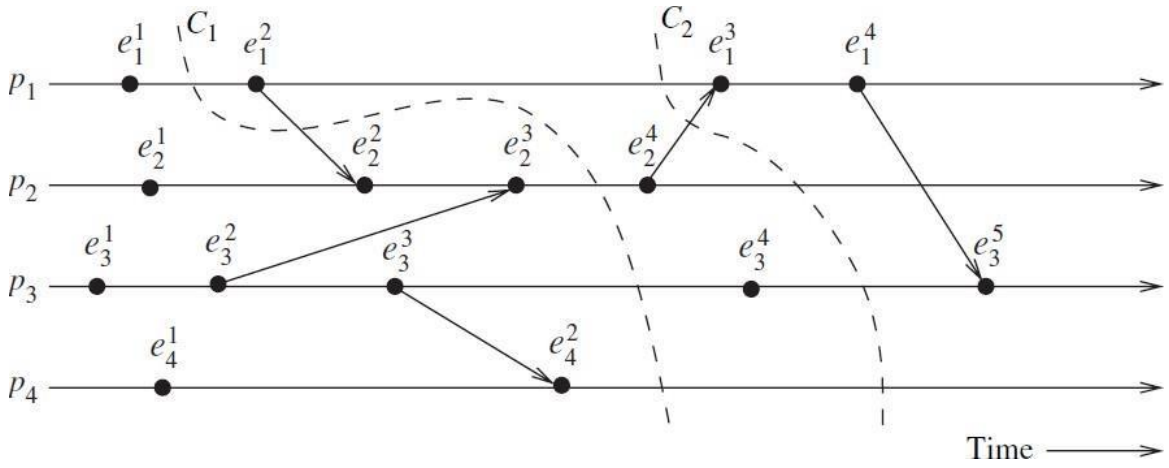
CUTS OF A DISTRIBUTED COMPUTATION

- In the space–time diagram of a distributed computation, a zigzag line joining one arbitrary point on each process line is termed as **cut** in the computation.
- The PAST(C) contains all the events to the left of the cut C and the FUTURE(C) contains all the events to the right of the cut C.
- Every cut corresponds to a global state.

definition 2.1 *If $e_i^{Max\_PAST_i(C)}$ denotes the latest event at process p_i that is in the* **PAST** *of a cut C, then the global state represented by the cut is* $\{\bigcup_i LS_i^{Max\_PAST_i(C)}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ where $SC_{jk}^{y_j,z_k} = \{m \mid send(m) \in \mathbf{PAST(C)} \wedge rec(m) \in \mathbf{FUTURE}(C)\}$.

- A consistent global state corresponds to a cut in which every message received in the PAST of the cut is sent in the PAST of that cut. Such a cut is known as a **consistent cut**.
- All messages that cross the cut from the PAST to FUTURE are in transit of consistent global state.

- A cut is **inconsistent** if a message crosses the cut from the FUTURE to PAST.



(Illustration of cuts in a distributed execution)

- Cuts in a space–time diagram is a powerful graphical aid to represent and reason about global states of a computation.

PAST AND FUTURE CONES OF AN EVENT

- Let Past(ej) denote all events in the past of ej in a computation (H,→). Then,Past (ej) = { ei |∀ei ∈ H, ei → ej }
- Pasti(ej) be the set of all those events of Past(ej) that are on process pi.
- Max_Past(ej) = ∪∀i{max(Pasti(ej)). Max_Pasti(ej) consists of the latest event at everyprocess that affected event ej called as the surface of the past cone of ej.
- max(Pasti(ej)) is always a message send event.
- The future of an event ej ,Future(ej) contains all the events ei that are causally affectedby ej.
- In a computation (H, →), Future(ej) is defined as:Future(ej) = {ei | ∀ei∈H, ej → ei}
- Futurei(ej) as the set of events of Future(ej) on process pi and
- min(Futurei(ej)) as the first event on process pi that is affected by ej . It is always a message receive event.
- Min_Past(ej) is ∪ ∀i{min_Futurei(ej))}, consists of first event at every process i.e.,causally affected by event ej is called the surface of the future cone of ej.
- All events at a process pi that occurred after max_Pasti(ej)) but before min_Futurei(ej)are concurrent with ej .

- Therefore, all and only those events of computation H that belong to the set "H − Past(ej) − Future(ej)" are concurrent with event ej.

MODELS OF PROCESS COMMUNICATIONS

- There are two models of process communications synchronous and asynchronous.
- **Synchronous communication model:**
  - It is a blocking type where the sender process blocks until the message received by the receiver process.
  - the sender and receiver processes must synchronize to exchange a message.
  - Synchronous communication is simple to handle and implement.
  - The frequent blocking lead to poor performance and deadlocks.
- **Asychronous Communication Model:**
  - It is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
  - After sent a message, the sender process buffers the message and is delivered to the receiver process when it is ready to accept the message.
  - A buffer overflow may occur if sender sends a large number of messages.
  - It provides higher parallelism as the sender executes while the message is in transit to the receiver.
  - Implementation of asynchronous communication requires complex buffer management.
  - Due to higher degree of parallelism and non-determinism, it is difficult to design, verify, and implement.

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.

PHYSICAL CLOCK

Most computers today keep track of the passage of time with a battery-backed-up CMOS clock circuit, driven by a quartz resonator. This allows timekeeping to take place even if the machine or the CPU is powered off. When on, an operating system will generally program a timer circuit (typically an Advanced Programmable Interrupt Controller, or APIC, in Intel-based systems) to generate an interrupt periodically. Many Linux systems, for example, 250 interrupts per second by de-fault. The interrupt service procedure simply adds one to a counter in memory to maintain a monotonically increasing value that represents the passage of time. This value is known as the **software clock or kernel clock** to differentiate it from the **hardware clock** (also known as the **CMOS clock**).

While the best quartz resonators can achieve an accuracy of one second in 10 years, they are sensitive to changes in temperature and acceleration and their resonating frequency can change as they age. Standard resonators are accurate to 6 parts per million at 31° C, which corresponds to ±½ second per day.

The problem with maintaining a concept of time is when multiple entities expect each other to have the same idea of what the current time is. Two watches hardly ever agree. Computers have the same problem: a quartz crystal on one computer will oscillate at a slightly different frequency than on another computer, causing the clocks to tick at different rates.

The phenomenon of clocks ticking at different rates, creating an ever widening gap in perceived time is known as **clock drift**.

The difference between two clocks at any point in time is called **clock skew** and is due to both clock drift and the possibility that the clocks may have been set differently on different machines. Figure 1 illustrates this phenomenon with two clocks, A and B, where clock B runs slightly faster than clock A by approximately two seconds per hour.

This is the clock drift of B relative to A. At one point in time (five seconds past five o'clock according to A's clock), the difference in time between the two clocks is approximately four seconds. This is the clock skew at that particular time.

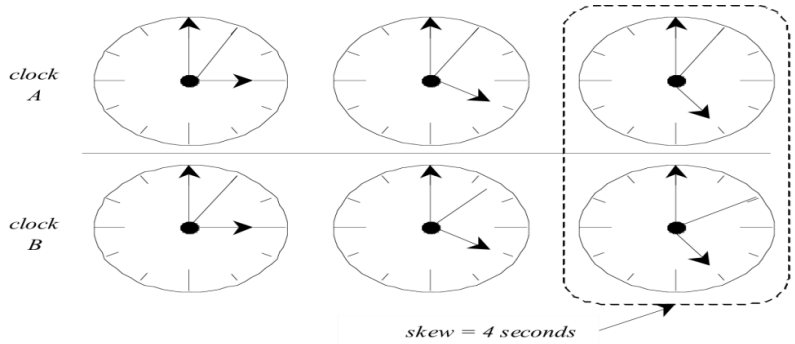
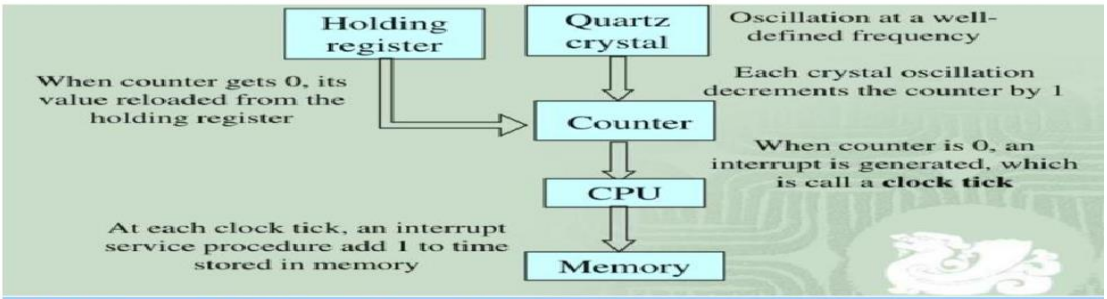


Figure 1. Clock drift and clock skew

- Also called Timer, usually a quartz crystal, oscillating at a well-defined frequency. A timer is associated with two registers: a counter and a holding register, and counter decreasing one at each oscillation.
- When the counter gets to zero, an interruption is generated and is called one clock tick.
- Crystals run at slightly different rates, the difference in time value is called a clock skew.
- Clock skew causes time-related failures.



How Clocks Work in Computer

How to synchronize local clock (1)

- If a local clock is slower, we can adjust it advancing forward (lost a few clock ticks), but how about it was faster than UTC?
  - set clock backward might cause time-related failures
- Use a soft clock to provide continuous time :
  - Let S be a soft clock, H the local physical clock
  - $S(t) = H(t) + \delta(t)$ (1)
  - The simplest compensating factor δ is a linear function of the physical clock: $\delta(t) = aH(t) + b$ (2)
  - Now, our problem is how to find constant a and b

How to synchronize local clock (2)

- Replace formula (1), we have
  - $S(t) = (1 + a)H(t) + b$ (3)
  - Let the value of S be T_{skew}, and the UTC at h be T_{real}, we may have that $T_{skew} > T_{real}$ or $T_{skew} < T_{real}$.
  - So S is to give the actual time after N further ticks, we must have:

    $T_{skew} = (1 + a)h + b$ (4)

    $T_{real} + N = (1 + a)(h + N) + b$ (5)
- Solve (4) and (5), we have:

  $a = (T_{real} - T_{skew})/N$

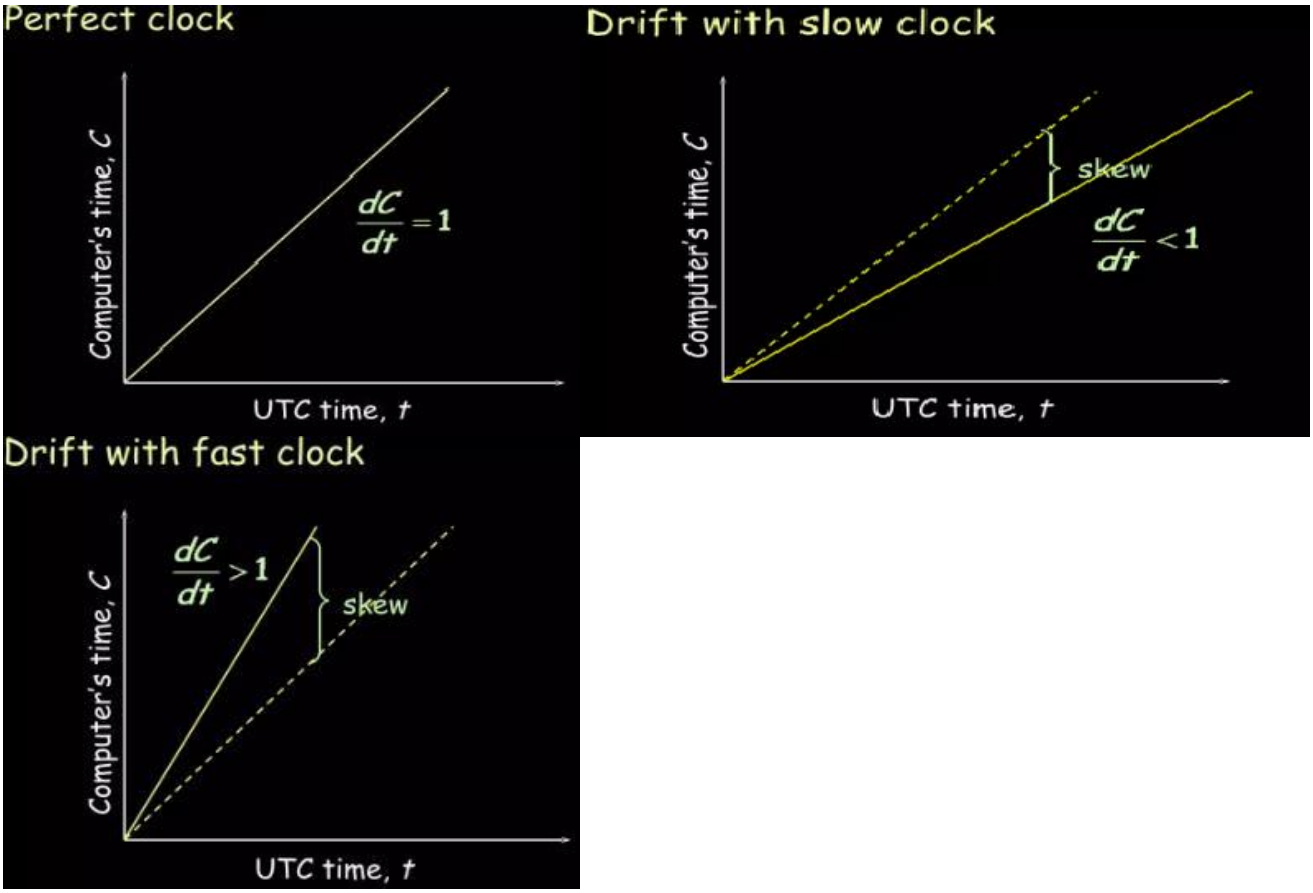
  $b = T_{skew} - (1 + a)h$

How to synchronize distributed clocks

- Assume at UTC time t, a physical clock time is H(t):
  - If they agree, then $dH/dt = 1$
  - But it is virtually impossible, for each physical clock, there is a constant ρ (given by manufacturers, called maximum drift rate), such that

    $1 - \rho \le dH/dt \le 1 + \rho$
  - If two clocks drift away from UCT in the opposite direction, then after Δt, they are $2\rho\Delta t$ apart.
  - Thus, if we want to guarantee that no two clocks ever differ by more than δ, clocks must be re-synchronized at least every $\delta/2\rho$ seconds.

Figure:2 Example for Clock Drift

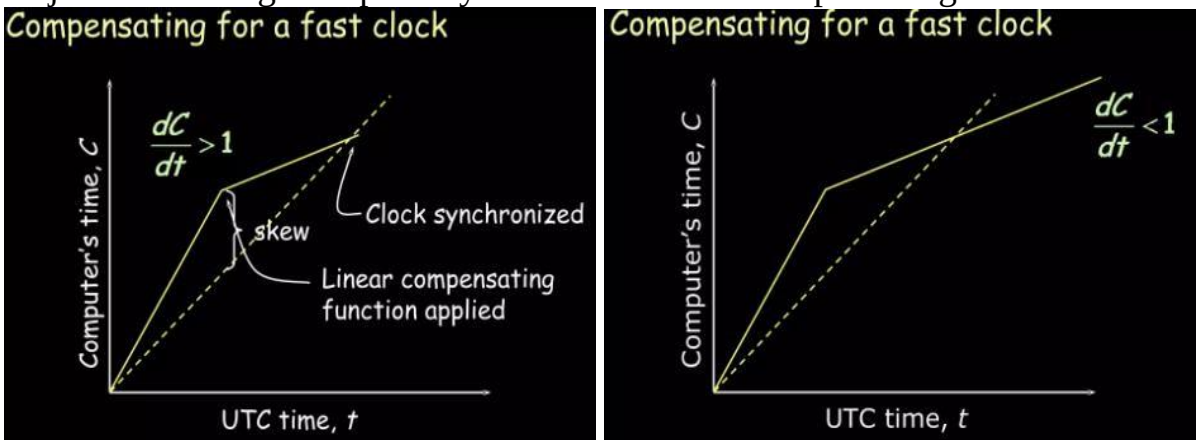


Dealing with Drift

Assume we set computer to true time,it is not good idea to set clock back.

- Illusion of time moving backwards can confuse message ordering and software development environments.
- Go for gradual clock correction

  If fast: Make clock run slower untill it synchronizes. If Slow: Make clock run faster untill it synchronizes.
- OS can do this: if system requests interrupts every 17 msec but clock is too slow then request interrupts at 15 msec. (or) Software correction: redefine the interval.
- Adjustment changes slope of system time: Linear compensating function.



Compensating for drift

We can envision clock drift graphically by considering true (UTC) time flowing on the x-axis and the corresponding computer's clock reading on the y-axis. A perfectly accurate clock will exhibit a slope of one. A faster clock will create a slope greater than unity while a slower clock will create a slope less than unity. Suppose that we have a means of obtaining the true time. One easy, and frequently adopted, solution is to simply update the system time to the true time. This works well for personal computers but may cause problems on servers and other systems that are actively running processes as these processes may see a spontaneous change in time, possibly a jump back in time!

To avoid this, one constraint that we will impose on clock synchronization is that it is not a good idea to set the clock back. The illusion of time moving backwards can confuse real-time-based message ordering, users, and software development environments.

If a clock is fast, it simply has to be made to run slower until it synchronizes. If a clock is slow, the clock can be made to run faster until it synchronizes. In theory, the operating system can do this by changing the rate at which it requests interrupts. For example, suppose the system requests an interrupt every 17 milliseconds (pseudo-milliseconds, really – the computer's idea of what a millisecond is) and the clock runs a bit too slowly. The system can request interrupts at a faster rate, say every 16 or 15 milliseconds, until the clock catches up. However, this is not always a Practical approach since we may not have enough precision in the timer. It is easier to avoid muck ing around with the hardware and just redefine the rate at which system time is advanced with each interrupt. Hence, whenever the operating system will read the software clock, it will apply an adjustment to the counter to compensate for drift.

This adjustment changes the slope of the system time and is known as a **linear compensation function** (Figure 3). After the synchronization period is reached, one can choose to resynchronize periodically and/or keep track of these adjustments and apply them continually to get a better running clock. This is analogous to noticing that your watch loses a minute every two months and making a mental note to adjust the clock by that amount every two months (except the system does it continually). For an example of clock adjustment, see the Linux man page for adjtime.

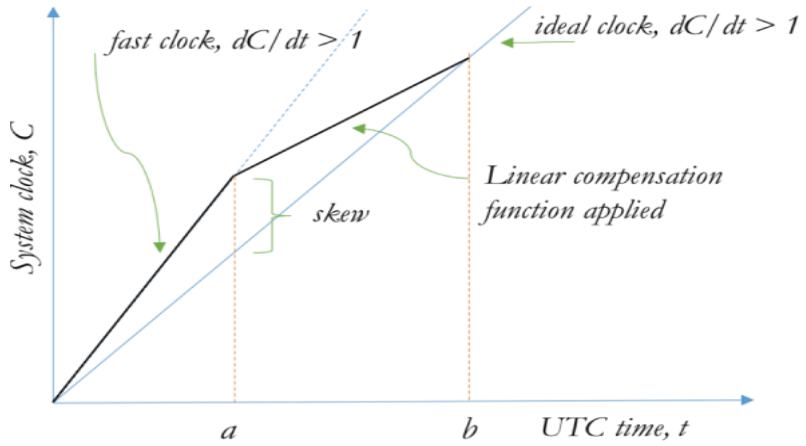


Figure 3: Compensating for drift with a linear compensation function

Setting the time on physical clocks

With physical clocks, our interest is not in advancing them just to ensure proper message ordering, but to have the system clock keep good time. We looked at methods for adjusting the clock to compensate for skew and drift, but it is essential that we can find the precise time first so that we would know how we need to adjust our clock.

One possibility is to attach a GPS (Global Positioning System) receiver to each computer. A GPS receiver will provide time from within ±100 ns[1] to ±1 µs of UTC time. USB-connected ones can be had for under US
$30. Some devices, such as phones and tablets, have a GPS receiver built into them (the chip cost is under
$5). Unfortunately, they often do not work well indoors. If the ma- chine is in the U.S., one can attach a WWV radio receiver to obtain time broadcasts from the National Institute of Standards and Technology at Boulder, Colorado or Washington, DC, with accuracies of ±3–10 ms, depending on the distance from the source. Another option is to obtain a GOES (Geostationary Operational Environment Satellites) receiver, which will provide time with in ± 0.1 ms of UTC time. For reasons of economy, convenience, and reception, these are not practical solutions for every machine. Most systems will set their time by asking another computer for the time, preferably one with one of the afore mentioned time sources connected to it. A computer that provides this information is called a time server.

1 Just a reminder: 1ns is 1 nanosecond, or one billionth of a second. 1 µs is 1 microsecond, or one millionth of a second. 1 ms is 1 millisecond, or 1 thousandth of a second.

MODELS OF PROCESS COMMUNICATIONS

There are two models of process communications synchronous and asynchronous.

- **Synchronous communication model:**
  - It is a blocking type where the sender process blocks until the message received by the receiver process.
  - the sender and receiver processes must synchronize to exchange a message.
  - Synchronous communication is simple to handle and implement.
  - The frequent blocking lead to poor performance and deadlocks.
- **Asynchronous Communication Model:**
  - It is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
  - After sent a message, the sender process buffers the message and is delivered to the receiver process when it is ready to accept the message.
  - A buffer overflow may occur if sender sends a large number of messages.
  - It provides higher parallelism as the sender executes while the message is in transit to the receiver.

Implementation of asynchronous communication requires complex buffer management.

- Due to higher degree of parallelism and non-determinism, it is difficult to design, verify, and implement.

Clock synchronization deals with understanding the temporal ordering of events produced by concurrent processes. It is useful for synchronizing senders and receivers of messages, determining whether messages are related and their proper ordering, controlling joint activity, and serializing concurrent access to shared objects. Multiple autonomous processes running on different machines need to be able to agree on and be able to make consistent decisions about the ordering of certain events in a system.

Our real-world view of clock synchronization is one of ensuring that multiple processes on multiple machines all see the same time of day. All modern computers have a time-of-day clock and synchronization becomes a matter of keeping these clocks set to an agreed-upon value, usually standard time as defined by UTC, Coordinated Universal Time.

Having synchronized clocks is extremely useful but is not always sufficient. Consider our ability to identify the sequencing and interdependence of events, such as sending or receiving messages or executing a transaction. A timestamp from a time-of-day clock on each such event will identify when the event happened has two potential pitfalls.

First, if two events take place at approximately the same time, they may be reported as taking place at the same time since timestamps have a limited precision. Worse, the

clocks on different systems may not be precisely synchronized so that an event on one computer may be assigned a later timestamp than that on another even if it took place earlier in time than the other event. That produces the false impression that the first event happened after the event that took place on the other computer. As an example, consider the case where process A sends a message with an embedded timestamp of 4:15:00 and machine B, on another computer, sends a message with a timestamp of 4:15:05. It is quite possible that process B's message was actually sent prior to that of process A if B's clock was over 5 seconds faster. Even if A's and B's clocks were synchronized, it is likely that the clocks run at slightly different speeds, drift apart over time, and eventually re- port different times.

Second, just by looking at two timestamps, you cannot tell if one event may be the result of another event or if they are completely independent. The messages sent have identical or even mis-leading timestamps, as in the above example. If we use algorithms that rely on choosing one timestamp over another, we may not be able to make a consistent decision either by comparing two message timestamps. Worse yet, if we are using a distributed algorithm where each process compares the timestamp in a received message with its own clock, there is no assurance that all systems will yield the same result.

To enable us to get both the time of day as well as an ability to compare events in a meaningful manner, we will use two forms of clocks: **physical** and **logical clocks**.

A **physical clock**, on the other hand, reports the time of day. Physical clock synchronization deals with synchronizing time-of-day clocks among groups of machines. In this case, we want to ensure that all machines can report the same time, regardless of how imprecise their clocks may be or what the network latencies are between the machines.

LOGICAL CLOCKS

Let us consider cases that involve assigning sequence numbers ("timestamps") to events upon which all cooperating processes can agree. What matters in these cases is not the time of day at which the event occurred but that all processes can agree on the order in which related events occur. Our interest is in getting event sequence numbers that make sense system-wide. These clocks are called logical clocks.

The concept of a **logical clock** is one where the clock does not have any bearing on the time of day but rather is a number that can be used for comparing sets of events, such as messages, within a distributed system.

If we can do this across all events in the system, we have something called **total ordering**: every event is assigned a unique timestamp (number) and every such

timestamp is unique. However, we don't always need total ordering. If processes do not interact then we do not care when their events occur. If we only care about assigning timestamps to potentially related (causal) events, then we have something known as **partial ordering**.

With logical time, we would like to assign a time value (sequence number) to each event such that everyone will agree on the final order of events. That is, if a→b then clock(a) < clock(b) since the clock (our sequence generator) must never run back-wards. If a and b occur on different processes that do not exchange messages (even through third parties) then a→b is not true. These events are said to be concurrent: there is no way that a could have influenced b.

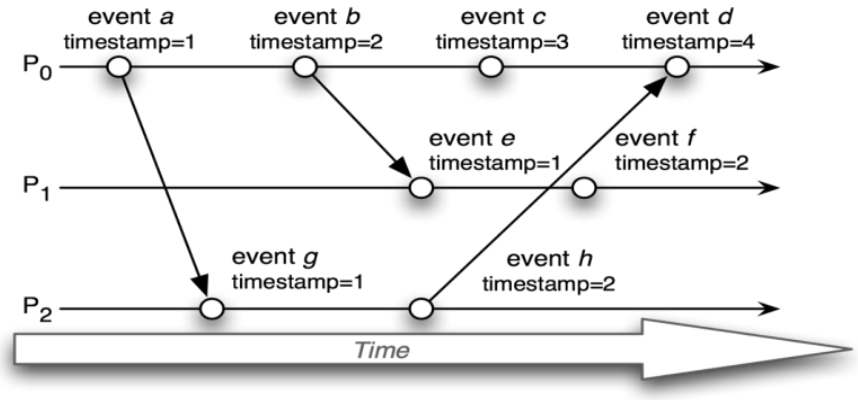


Figure:4 Un sequenced event stamps

In the above figure 4, there are three processes. Each event is assigned a timestamp by its respective process. Each process simply maintains its own counter that is incremented before each event gets its timestamp. If we examine the timestamps from a global perspective, we can observe a number of peculiarities. Event g, the event representing the receipt of the message sent by event a, has the exact same timestamp as event a when it clearly had to take place after event a. Event e has an earlier timestamp 1 than the event sent the message with timestamp.

(1) Event Ordering

In a centralized system, we can always determine the order in which two events occurred, since the system has a single common memory and clock. Many applications may require us to determine order. For example, in a resource allocation scheme, we specify that a resource can be used only after the resource has been granted. A distributed system, however, has no common memory and no common clock. Therefore, it is sometimes impossible to say which of two events occurred first. The Happened-before relation is only a partial ordering of the events in distributed systems.

Since the ability to define a total ordering is crucial in many applications, we present a distributed algorithm for the happened-before relation to a consistent total ordering of all the events in the system.

Leslie Lamport defined a happens before notation to express the relationship between events: a→b means that a happens before b. If a represents the timestamp of a message sent and b is the timestamp of that message being received, then a→b must be true; a message cannot be received before it is sent. This relationship is transitive. If a→b and b→c then a→c. If a and b are events that take place in the same process then a→b is true if a occurs before b.

The Happened-Before Relation

Since we are considering only sequential processes, all events executed in a single process are totally ordered. Also, by the law of causality, a message can be received only after it has been sent. Therefore, we can define the happened before relation (denoted by -») on a set of events as follows (assuming that sending and receiving a message constitutes an event):

1. If A and B are events in the same process, and A was executed before B, then A -» B.

2. If A is the event of sending a message by one process and B is the event of receiving that message by another process, then A —»• B.

3. If A -> B and B -» C then A -» C. Since an event cannot happen before itself, the -> relation is an irreflexive partial ordering.

If two events, A and B, are not related by the —> relation (that is, A did not happen before B, and B did not happen before A), then we say that these two events were executed concurrently. In this case, neither event can causally affect the other. If, however, A -> B, then it is possible for event A to affect event B causally.

A space-time diagram, such as that in Figure 5, can best illustrate the definitions of concurrency and happened-before. The horizontal direction represents space (that is, different processes), and the vertical direction represents time. The labeled vertical lines denote processes (or processors). The labeled dots denote events.

A wavy line denotes a message sent from one process to another. Events are concurrent if and only if no path exists between them. For example, these are some of the events related by the happened-before relation in Figure

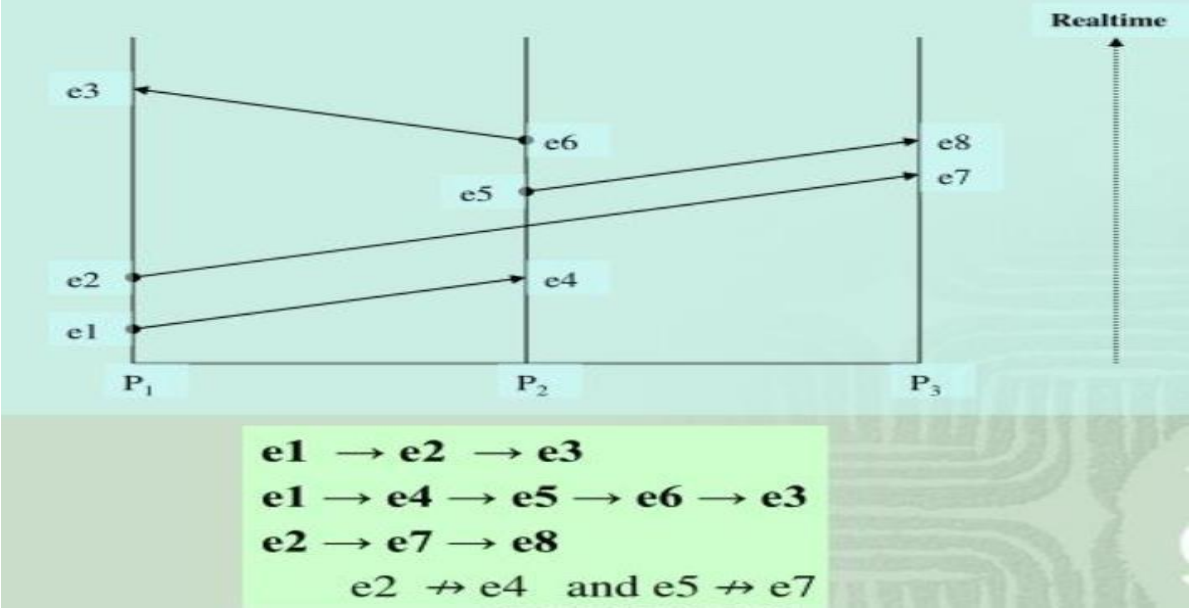


Figure: 5 Event Ordering

(2) Lamport Timestamp

LOGICAL CLOCK:

Is a mechanism for capturing chronological and causal relationships in a distributed system. Distributed systems may have no physically synchronous global clock, so a logical clock allows global ordering on events from different processes in such systems. The first implementation, the Lamport timestamps, was proposed by Leslie Lamport in 1978.

Lamport's Logical Clocks:

The algorithm of Lamport timestamps is a simple algorithm used to determine the order of events in a distributed computer system. As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead, and conceptually provide a starting point for the more advanced vector clock method.

For synchronization of logical clocks, Lamport established a relation termed as "happens-before" relation.

1. Happens- before relation is a transitive relation, therefore if p→q and q→r, then p→r.
2. If two events, a and b, occur in different processes which not at all exchange messages amongst them, then a→b is not true, but neither is b→a which is antisymmetry.

The algorithm follows some simple rules:

- A process increments its counter before each event in that process.
- When a process sends a message, it includes its counter value with the message.
- On receiving a message, the counter of the recipient is updated, if necessary, to the greater of its current counter and the timestamp in the received message. The counter is then incremented by 1 before the message is considered received.

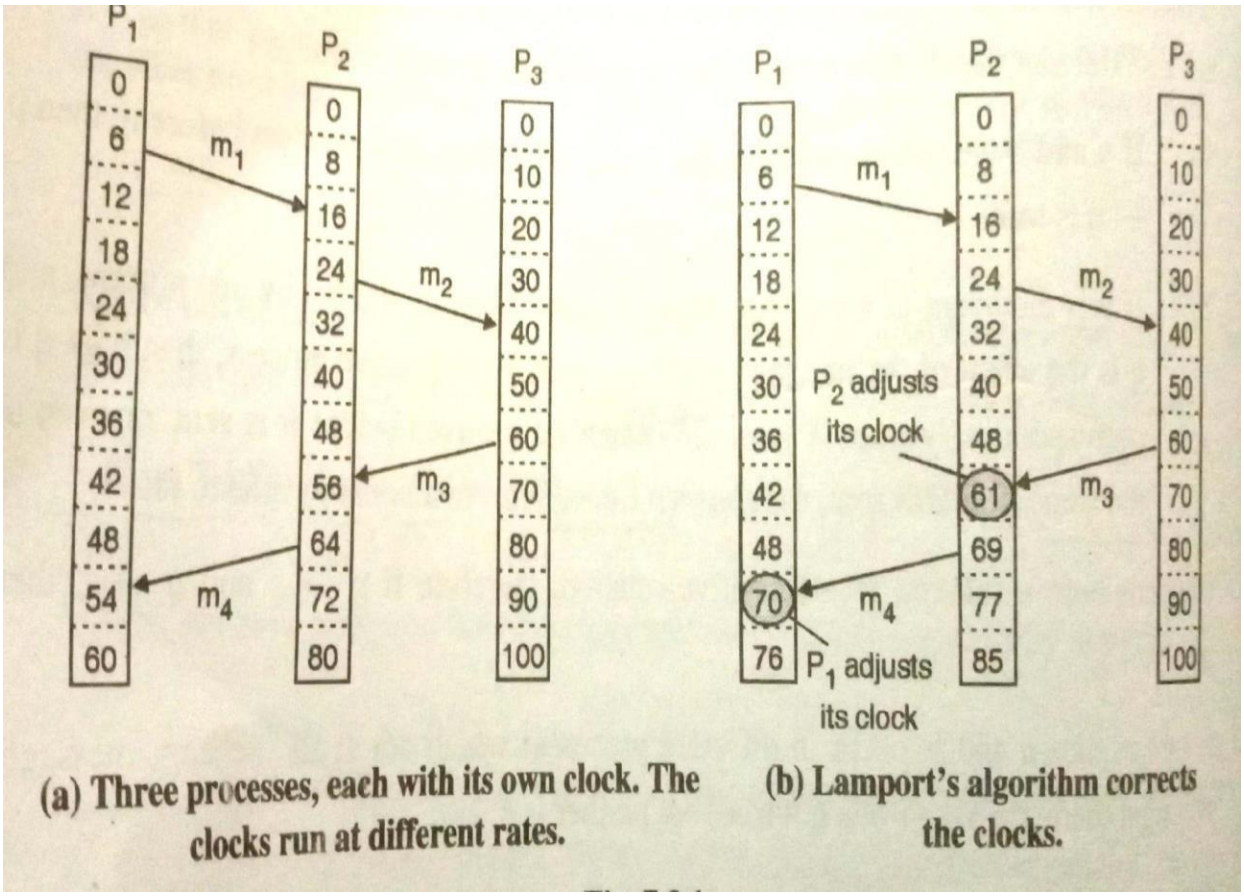
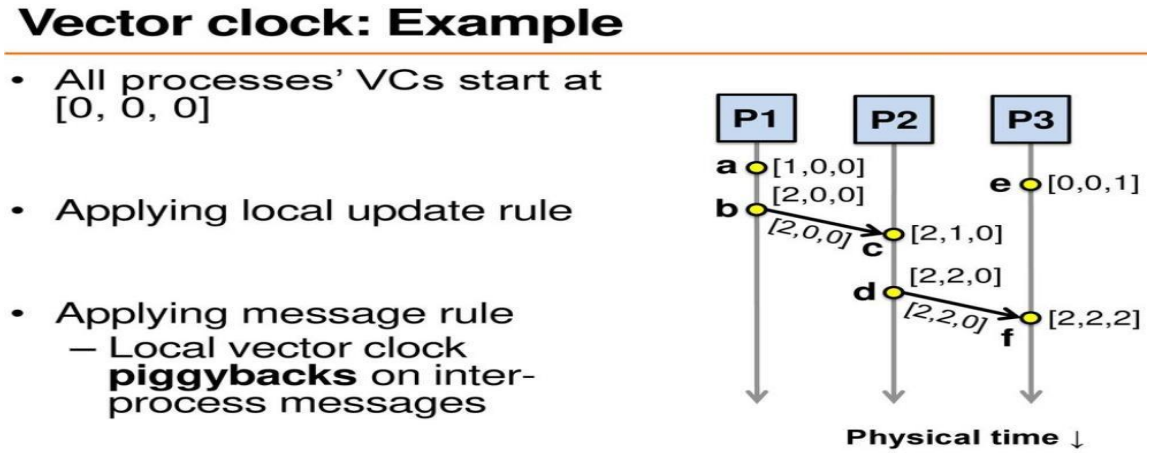


Figure: 6 Example for lamport synchronize clocks

(3) Vector Timestamp



- With Lamport's clocks, one cannot directly compare the timestamps of two events to determine their precedence relationship.
- The main problem is that a simple integer clock cannot order both events within a process and events in different processes. Collin Fidge developed an algorithm that overcomes this problem.
- A vector clock system is a mechanism that associates timestamps with events (local states) such that comparing two events' timestamps indicates whether those events (local states) are causally related.
- Fidge's clock is represented as a vector [C1, C2, ..., Cn] with an integer clock value for each process (Ci contains the clock value of process i).
- Each process Pi has an array VCi [1..n], where VCi [j] denotes the number of events that process Pi knows have taken place at process Pj.
- When Pi sends a message m, it adds 1 to VCi [i] and sends VCi along with m as vector timestamp vt(m). Result: upon arrival, recipient knows Pi 's timestamp.
- When a process Pj delivers a message m that it received from Pi with vector timestamp ts(m), it updates each VCj [k] to max(VCj [k], ts(m)[k]} and increments VCj [j] by 1.
- We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

- Pi increments VCi [i] only when sending a message, and Pj "adjusts" VCj when receiving a message (i.e., effectively does not change VCj [j]).
- In the time-stamping system, each process Pi has a vector of integers VCi[1..n] (initialized to [,0,...,0]) that is maintained as follows:
- Each time process Pi produces an event (send, receive, or internal), it increments its vector clock entry VCi[i] (VCi[i]:= VCi[i]+1) to indicate that it has progressed.
- When a process Pi sends a message m, it attaches to it the current value of VCi. Let m.VC denote this value.
- When Pi receives a message m, it updates its vector clock as
- Note that VCi[i] counts the number of events that Pi has so far produced. Vx: VC,[x]:= max(VC [x],m.VC[x])(abbreviated as VC;; = max(VC,,m.VC).

SCALAR TIME

In a distributed system, multiple machines are working together, and each machine may have its own clock. Still, these clocks may not be accompanied with each other, and there's no single clock that can be used to order events globally. Logical clocks give a way to handle this by assigning each event a logical timestamp, which can be used to order events and establish reason between them, indeed if they do on different machines. In substance, logical clocks give a way to produce a virtual global timepiece that's consistent across all machines in a distributed system.

Some applications need to record the time when events occur (e.g. ecommerce, banking). May need to determine the relative order in which certain events occurred. Physical events order based on observer's frame of reference .Multiple computer clocks may be skewed and need to by synchronized – no absolute global time for all computers in a distributed system

Scalar Time Implementation

There are different methods for implementing logical clocks in a distributed system, but one of the simplest is Scalar Time. In this implementation, each process keeps a local clock that is initially set to 0. There are two rules for updating the local clock −

Rule 1: Event Execution

Before executing an event (excluding the event of receiving a message), increment the local clock by 1.

```
local_clock = local_clock + 1
```

Ru

When receiving a message (the message must include the sender's local clock value), set your local clock to
the maximum of the received clock value and the local clock value. After this, increment your local clock by 1.

```
local_clock = max(local_clock, received_clock)
local_clock = local_clock + 1
```

This ensures that events are ordered similarly so that if event A contributes to event B being, the timestamp of event A is lower than the timestamp of event B.

Example

For illustration, there are 3 processes in a distributed system P1, P2, and P3. However, the message will include the original timepiece value of P1, If an event occurs in P1 and also a message is transferred to P2. When P2 receives the message, it'll modernize its own original timepiece using Rule 2, icing that events are ordered rightly.

Timestamp Implementation

Another method for implementing logical clocks is using timestamps. Each event is assigned a timestamp, and events are ordered based on the causality relationship (i.e., the "happened before" relationship) between the events. However, timestamps only work as long as they obey causality.

Example

For illustration, suppose we have a distributed system with multiple processes, and process P1 sends a communication to process P2. However, the reason relationship is violated and the ordering of events is no longer consistent, If the timestamp of the communication from P1 is lesser than the timestamp of an earlier communication from P2.

Causality in Distributed Systems

In a distributed system, causality is still based on the happen-before relationship, just like in a single PC system. If event A occurs in process P1 and event B occurs in

process P2, we still need to establish which event occurred first to maintain causality. In other words, we need to determine the order of events across different processes.

Timestamps and Causality

To do this, we can use logical clocks, which assign a timestamp to each event. The rule is still the same if event A causes event B, also the timestamp of A should be lower than the timestamp of B.

Example

For illustration, consider a distributed system where process P1 sends a communication to process P2. The transferring event in P1 has a timestamp of 1, and the entering event in P2 has a timestamp of 2. This indicates that the transferring event passed before the entering event, and we can establish an unproductive relationship between them.

thus, the reason is important in logical timepieces in distributed systems to maintain the order of events and establish a cause-and-effect relationship between them.

Conclusion

logical clocks provide a way to maintain the consistent ordering of events in a distributed system. Scalar Time is a simple implementation that uses local clocks and two rules to ensure correct order. Other methods, such as using timestamps, can also be used but must obey causality to work properly. By using logical clocks, processes in a distributed system can work together in an organized and efficient way.

A FRAMEWORK FOR A SYSTEM OF LOGICAL CLOCKS

Definition

- 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 < called as happened before
    or causal precedence.
  - The logical clock C is a function that maps an event e to the time domain T, denoted as C(e) and called the timestamp of e, and is defined as follows:

    $$C: H \mapsto T$$

    Such that the following monotonicity property is satisfied then it is called the clock
    consistency condition.

    For two events ei and ej, $ei \rightarrow ej \Rightarrow C(ei) < C(ej)$.
  - When T and C satisfy the following condition, the system of clocks is said to be **Strongly consistent.**
    For two events ei and ej, $ei \rightarrow ej \Leftrightarrow C(ei) < C(ej)$,

Implementing logical clocks

Implementation of logical clocks requires addressing two issues:

- Data structures local to every process to represent logical time and a protocol to update the data structures to ensure the consistency condition.
- Each process pi maintains data structures with the following two capabilities:
  - A local logical clock, lci, that helps process pi to measure its own progress.
  - A logical global clock, gci, represents the process pi's local view of logical global time. It allows this process to assign consistent timestamps to its local events.
  - The protocol ensures that a process's logical clock, and thus its view of global time, is managed consistently.
  - The protocol consists of the following two rules:
  - **R1** This rule governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal).
  - **R2** This rule governs 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 it is used by the process to update its view of global time.

Logical clock

The above clock synchronization approaches assume synchronize local clock and use timestamp to reason the order of events. If the time difference between two events is smaller than the accuracy, then we cannot say which event happens first, thus problems may be caused. For many problems, we care about the internal consistency of clocks, not absolute time. The approach using logical clocks is proposed. Logical clocks do not need clock synchronization and take the order in which events occur rather than the time at which they occurred into account. If the processes only care about event A happens before event B, but dont care about the time difference exactly, they can use logical clock.

Event Ordering and Algorithm

Logical clocks are to solve the problems that define a total ordering of all events that occur in a system. In a distributed system, logical clocks do not have global clock and local clocks may be unsynchronized. Besides, you cannot order events on different machines using local times.

There are some key ideas of logical clocks proposed by scientist Lamport: can use send/receive messages exchanged between processes/machines to order events since messages must be sent before received. We then use the transitivity property to reason about order of events. For example, machine A sends a message to machine B along with its logical clock value 4, and machine B receives this message at its local logical clock value 3, then we can say that the events happens before logical clock value

4 in machine A occur before the events happens after logical clock value 3 in machine B without clock synchronization. However, we cannot say the order of the events happens after logical clock value 4 in machine A and the events happens before logical clock value 3 in machine B. This indicates that this algorithm only gives us a partial ordering of events.

Rules for updating logical clock values:

1) Whenever a local event occurs at machine i, it updates LCi as LCi+ = 1.
2) When machine i wants to send a message, it includes LCi in the message.
3) When machine j receives a message, it updates its logical clock as LCj = max(LCj , LCi) + 1.

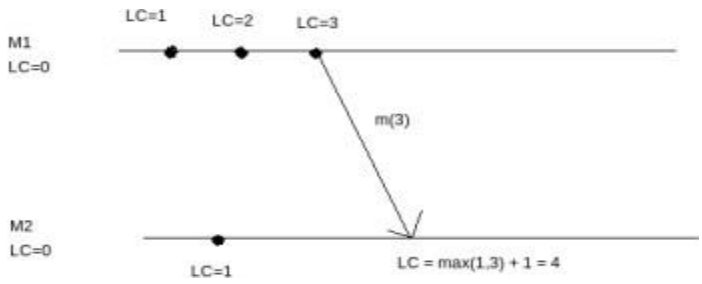


Figure: 7 Logical Clock Example

Important to note that if we know the order of two events A and B, this algorithm ensures that their timestamps will be in order (i.e if A → B, then ts(A) < ts(B)). However, we cannot be certain about the order of events, given just their timestamps i.e. ts(A) < ts(B) 6→ (A → B) because all we have is a partial ordering.

1.) Scalar Time

Definition

- The scalar time representation was proposed by Lamport to totally order events in a distributed system. Time domain is represented as the set of non-negative integers.
- The logical local clock of a process pi and its local view of global time are squashed into one integer variable Ci.
- Rules **R1** and **R2** used to update the clocks is as follows:
  R1 : Before executing an event (send, receive, or internal), process pi executes:

  $$Ci := Ci + d \qquad (d > 0)$$
- d can have a different value, may be application-dependent. Here d is kept at 1.
  R2 : Each message piggybacks the clock value of its sender at sending time. When a process pi receives a message with timestamp Cmsg, it executes the following actions:

1. Ci := max(Ci, Cmsg);
2. Execute **R1**;
3. Deliver the message.

- Figure 8 shows the evolution of scalar time with d=1.

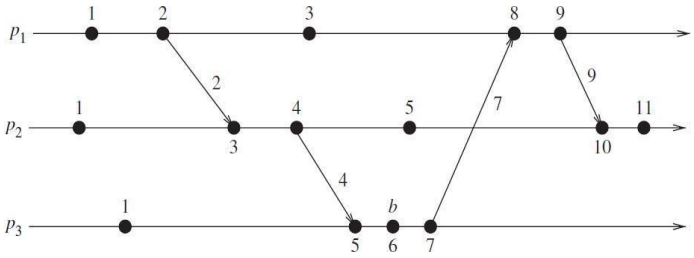


Figure 8: Evolution of scalar time

Basic properties:

(i) Consistency property

- Scalar clocks satisfy the monotonicity and consistency property. i.e., for two events ei and ej, ei $\rightarrow$ej $\Rightarrow$ C(ei) < C(ej).

(ii) Total Ordering

- Scalar clocks can be used to totally order events in a distributed system.
- Problem in totally ordering events: Two or more events at different processes may have an identical timestamp. i.e., for two events e1 and e2, C(e1) = C(e2) $\Rightarrow$ e1|| e2.
- In Figure 8, 3rd event of process P1 and 2nd event of process P2 have same scalar timestamp. Thus, a tie-breaking mechanism is needed to order such events.
- A tie among events with identical scalar timestamp is broken on the basis of their process identifiers. The lower the process identifier then it is higher in priority.
- The timestamp of an event is a tuple (t, i) where t - time of occurrence and i - identity of the process where it occurred. The total order relation $\prec$ on two events x and y with timestamps (h,i) and (k,j) is:

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

(iii) Event counting

- If the increment value of d is 1 then, if the event e has a timestamp h, then h−1 represents minimum number of events that happened before producing the event e;
- In the figure 8, five events precede event b on the longest causal path ending at b.

(iv) No strong consistency

- The system of scalar clocks is not strongly consistent; that is, for two events ei and ej, C(ei) < C(ej) ≠> ei→ej .
- In Figure 3, the 3rd event of process P1 has smaller scalar timestamp than 3rd event of process P2.

2.) Vector Time Definition

- The system of vector clocks was developed by Fidge, Mattern, and Schmuck.
- Here, the time domain is represented by a set of n-dimensional non-negative integer vectors.
- Each process pi maintains a vector vti[1..n], where vti[i] is the local logical clock of pi that specifies the progress at process.
- vti[j] represents process pi's latest knowledge of process pj local time.
- If vti[j] = x, then process pi knowledge on process pj till progressed x.
- The entire vector vti constitutes pi's view of global logical time and is used to timestamp events.
- Process pi uses the following two rules **R1** and **R2** to update its clock:

R1:

- Before executing an event, process pi updates its local logical time as follows: vti[i] = vti[i] + d (d>0)

R2:

- Each message m is piggybacked with the vector clock vt of the sender process at sending time.
- On receipt of message (m,vt), process pi executes: $1 \leq k \leq n$: vti[k] := max(vti[k], vt[k])
  1. execute **R1**;
  2. deliver the message m.
- The timestamp associated with an event is the value of vector clock of its process when the event is executed.The vector clocks progress with the increment value d = 1. Initially, it is [0, 0, 0, .. , 0].

  The following relations are defined to compare two vector timestamps, vh and vk: vh = vk ⇔ ∀x : vh[x] = vk[x]

  vh ≤ vk ⇔ ∀x : vh[x] ≤ vk[x]

  vh < vk ⇔ vh ≤ vk and ∃x : vh[x] < vk[x] vh || vk ⇔ ¬(vh < vk)A ¬(vk < vh)

Basic properties

(1) Isomorphism

- The relation "→" denotes partial order on the set of events in a distributed execution.
- If events are time stamped using vector clocks, then
- If two events x and y have timestamps vh and vk, respectively, then x→y

  $\Leftrightarrow vh < vk$

  $x \parallel y \Leftrightarrow vh \parallel vk$
- Thus, there is an isomorphism between the set of partially ordered events and their vector timestamps.
- Hence, to compare two timestamps consider the events x and y occurred at processes pi and pj are assigned timestamps vh and vk, respectively, then

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

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

(2) Strong consistency

- The system of vector clocks is strongly consistent;
- Hence, by examining the vector timestamp of two events, it can be determined that whether the events are causally related.

(3) Event counting

- If d is always 1 in rule **R1**, then the i^{th} component of the vector clock at process pi, vti[i], denotes the number of events that have occurred at pi until that instant. So if an event e has timestamp vh, vh[j] denotes the number of events executed by process pj that causally precede e.
- vh[j]−1 represents the total number of events that causally precede e in the distributed computation.

Applications

- As vector time tracks causal dependencies exactly, it's applications are as follows:
- distributed debugging,
- Implementations of causal ordering communication in distributed shared memory.
- Establishment of global breakpoints to determine consistency of checkpoints in recovery.
- A **linear extension** of a partial order (E,) is a linear ordering of E i.e., consistent with partial order, if two events are ordered in the partial order, they are also ordered in the linear order.

  It is viewed as projecting all the events from the different processes on a single time axis.

- The **dimension** of a partial order is the minimum number of linear extensions whose intersection gives exactly the partial order.

PHYSICAL CLOCK SYNCHRONIZATION

- 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.
- To correct this periodically, clock synchronization is performed. Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).
- Clocks that are not synchronized with each other will adhere to physical time termed as physical clocks.

For most applications and algorithms that runs in a distributed system requires:

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

Example Applications that need synchronization are:

- Secure systems, fault diagnosis and recovery, scheduled operations, database systems.
- Stock markets buy and sell orders.
- Aviation traffic control and positioning.
- Multimedia synchronization for real time teleconferencing.
- Interactive simulation, event synchronization and ordering.
- Distributed network gaming and training.

Classification of clock synchronization Algorithms

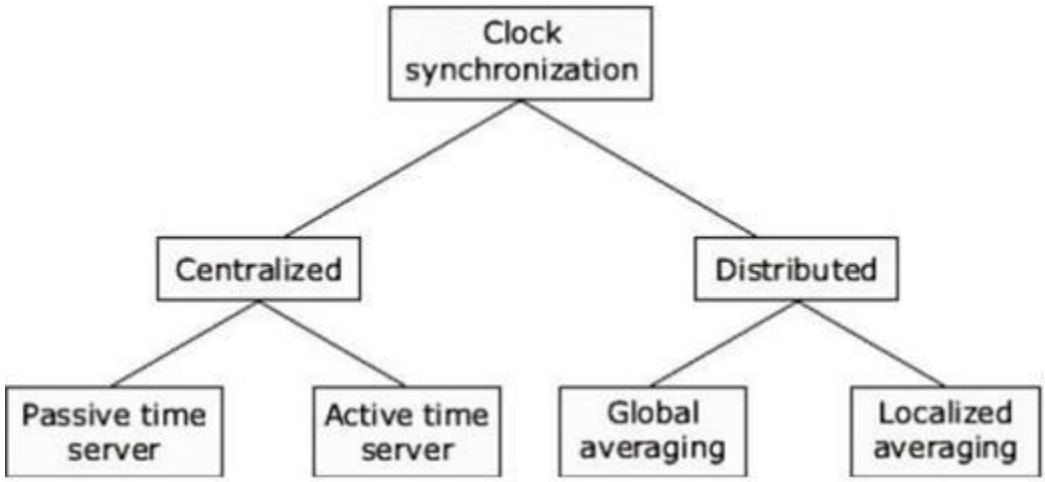


Figure: 9 Classification of synchronization Algorithms

Distributed System is a collection of computers connected via a high-speed communication network. In the distributed system, the hardware and software components communicate and coordinate their actions by message passing. Each node in distributed systems can share its resources with other nodes. So, there is a need for proper allocation of resources to preserve the state of resources and help coordinate between the several processes. To resolve such conflicts, synchronization is used. Synchronization in distributed systems is achieved via clocks. The physical clocks are used to adjust the time of nodes. Each node in the system can share its local time with other nodes in the system. The time is set based on UTC (Universal Time Coordination). UTC is used as a reference time clock for the nodes in the system. Clock synchronization can be achieved by 2 ways: External and Internal Clock Synchronization.

1. **External clock synchronization** is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.
2. **Internal clock synchronization** is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly.

There are 2 types of clock synchronization algorithms: Centralized and Distributed.

1. **Centralized** is the one in which a time server is used as a reference. The single time-server propagates it's time to the nodes, and all the nodes adjust the time accordingly. It is dependent on a single time- server, so if that node fails, the whole system will lose synchronization. **Examples** of centralized are-(i) Cristian's Algorithm, (ii) Berkeley algorithm (Passive Time Server), etc.
2. **Distributed** is the one in which there is no centralized time-server present. Instead, the nodes adjust their time by using their local time and then, taking the average of the differences in time with other nodes. Distributed algorithms overcome the issue of centralized algorithms like scalability and single point failure. **Examples** of Distributed algorithms are – Global Averaging Algorithm, Localized Averaging Algorithm, NTP (Network time protocol), etc.

Centralized clock synchronization algorithms suffer from two major drawbacks:

1. They are subject to a single-point failure. If the time-server node fails, the clock synchronization operation cannot be performed. This makes the system unreliable. Ideally, a distributed system should be more reliable than its individual nodes. If one goes down, the rest should continue to function correctly.
2. From a scalability point of view, it is generally not acceptable to get all the time requests serviced by a single-time server. In a large system, such a solution puts a heavy burden on that one process.

Distributed algorithms overcome these drawbacks as there is no centralized time-server present. Instead, a simple method for clock synchronization may be to equip each node of the system with a real-time receiver so that each node's clock can be independently synchronized in real-time.

1. CENTRALIZED ALGORITHMS

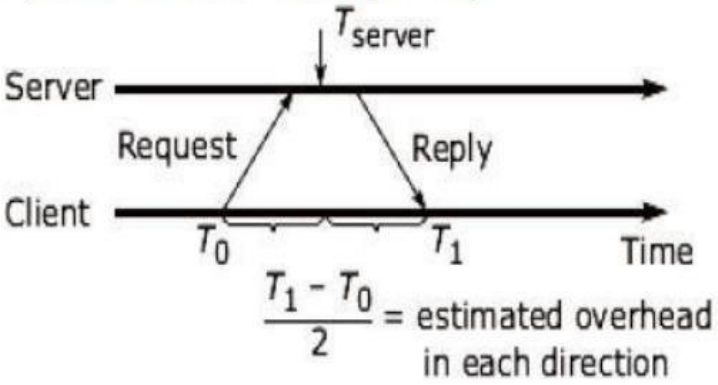
Let us see these centralized algorithms types in detail.

(i) CRISTIANS ALGORITHM FOR SYNCHRONIZING CLOCKS

Cristian's algorithm is centralized passive time server type algorithm. Client wants to correct its time based on the server time so it makes a request to the time server. It is a physical clock algorithm. It is based on client server concept. It makes use of RPC. The machine which has the WWV receiver, which receives precise UTC. It is called the time server.

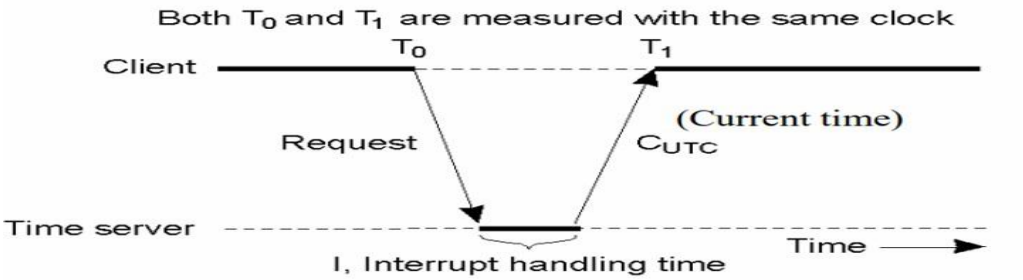
Passive time server algorithms –

- Each node periodically sends a message called 'time=?' to the time server.
- When the time server receives the message, it responds with 'time=T' message.
- Assume that client node has a clock time of T0 when it sends 'time=?' and time T1 when it receives the 'time=T' message.
- T0 and T1 are measured using the same clock, thus the time needed in propagation of message from time server to client node would be (T1- T0)/2
- When client node receives the reply from the time server, client node is readjusted to Tserver+(T1-T0)/2.
- Two methods have been proposed to improve estimated value
  - Let the approximate time taken by the time server to handle the interrupt and process the message request message 'time=?' is equal to I.
  - Hence, a better estimate for time taken for propagation of response message from time server node to client node is taken as (T1- T0-I)/2
  - Clock is adjusted to the value Tserver+(T1- T0-I)/2



Client sets time to: $T_{new} = T_{server} + \frac{T_1 - T_0}{2}$

Figure: Time approximation using passive time server algorithm



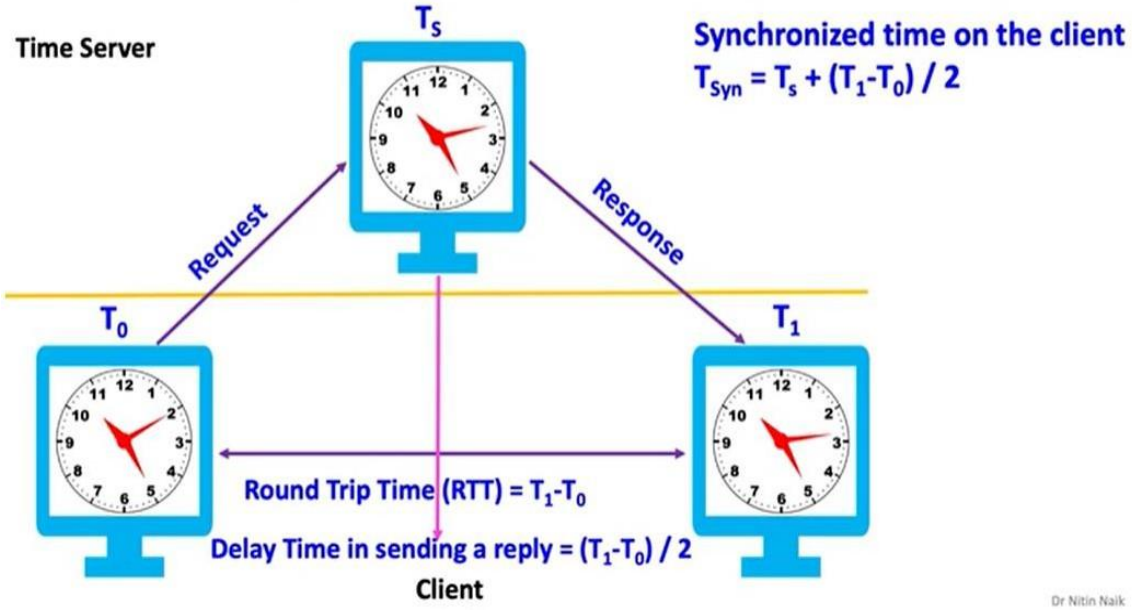
Algorithm

1. Let Time server be S
2. Process request time
3. After receiving request server prepares response& appends Ts to it

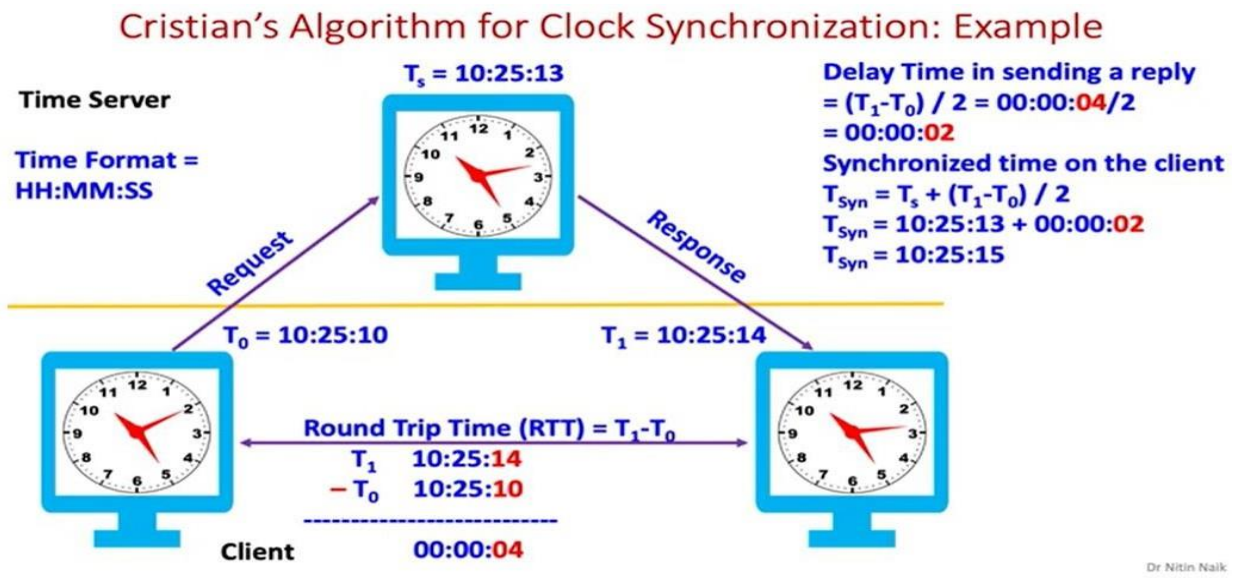
   Tp=((T1-T0)/2)+Ts
4. Now client has to correct it self, So, P then sets its time to

   Tp=((T1-T0)/2)+Ts

Cristian's Algorithm for Clock Synchronization: Computation



Dr Nitin Naik



- **Advantage**-It assumes that no additional information is available.
- **Disadvantage**- It restricts the number of measurements for estimating the value.

(ii) THE BERKLEY ALGORITHM

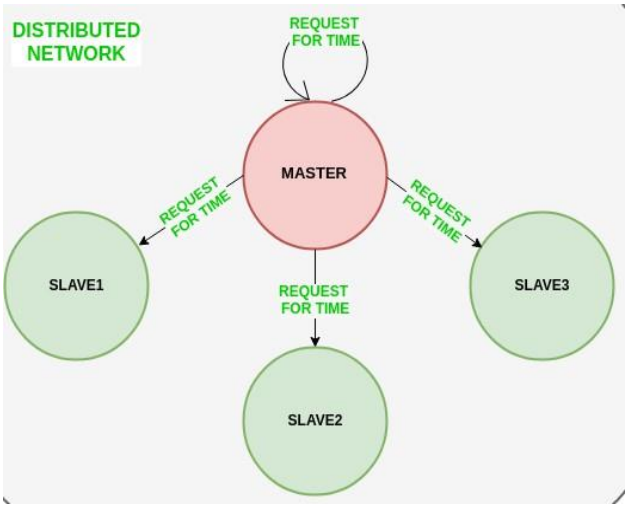
Berkeley's Algorithm is a clock synchronization technique used in distributed systems. The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess a UTC server

Algorithm

1) An individual node is chosen as the master node from a pool node in the network. This node is the main node in the network which acts as a master and the rest of the nodes act as slaves. The master node is chosen using an election process/leader election algorithm.

2) Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

3) Master node calculates the average time difference between all the clock times received and the clock time given by the master's system clock itself. This average time difference is added to the current time at the master's system clock and broadcasted over the network. The diagram below illustrates how the master sends requests to slave nodes.



The diagram above illustrates how slave nodes send back time given by their system clock.

Scope of Improvement

-

- Improvising inaccuracy of Cristian's algorithm.
- Ignoring significant outliers in the calculation of average time difference
- In case the master node fails/corrupts, a secondary leader must be ready/pre-chosen to take the place of the master node to reduce downtime caused due to the master's unavailability.
- Instead of sending the synchronized time, master broadcasts relative inverse time difference, which leads to a decrease in latency induced by traversal time in the network while the time of calculation at slave node.

Features of Berkeley's Algorithm:

Centralized time coordinator: Berkeley's Algorithm uses a centralized time coordinator, which is responsible for maintaining the global time and distributing it to all the client machines.

Clock adjustment: The algorithm adjusts the clock of each client machine based on the difference between its local time and the time received from the time coordinator.

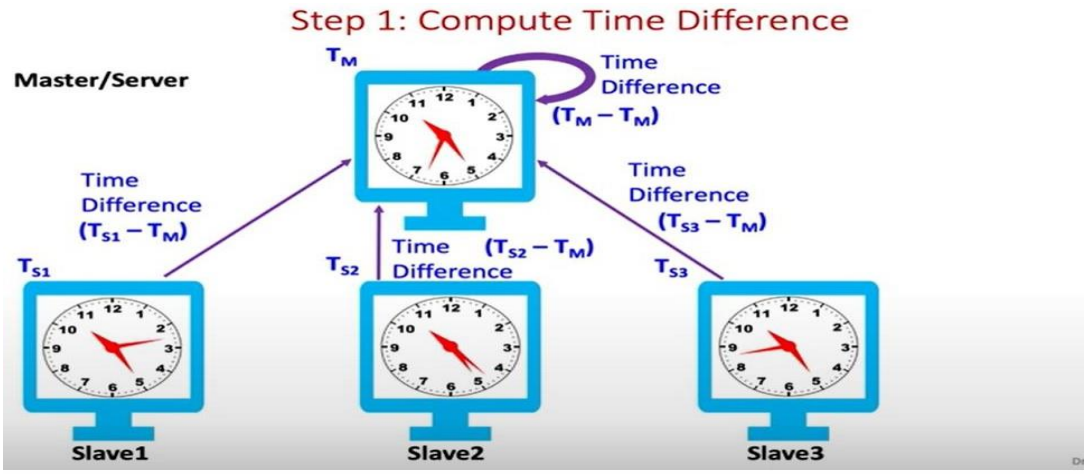
Average calculation: The algorithm calculates the average time difference between the client machines and the time coordinator to reduce the effect of any clock drift.

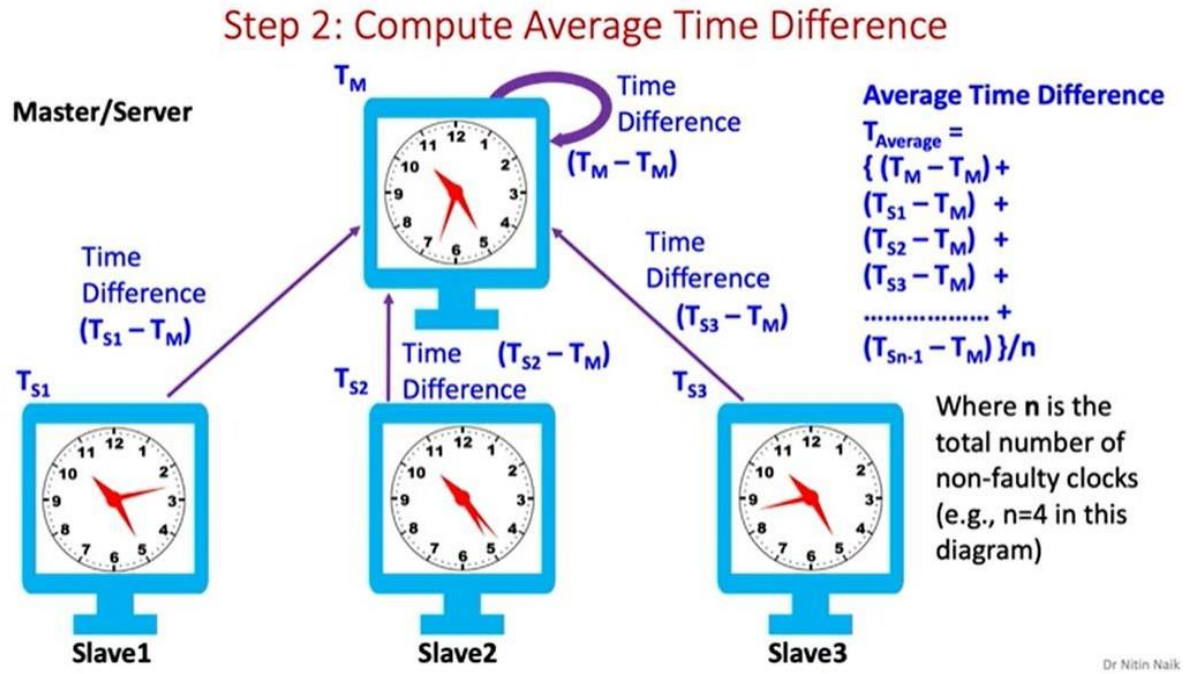
Fault tolerance: Berkeley's Algorithm is fault-tolerant, as it can handle failures in the network or the time coordinator by using backup time coordinators.

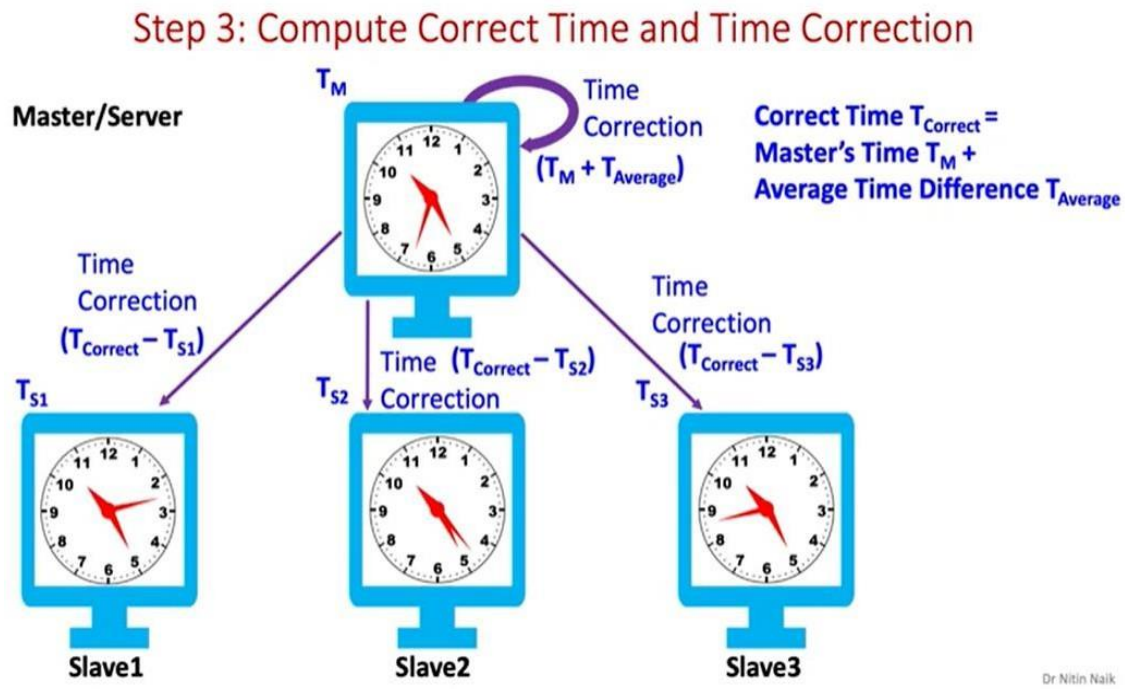
Accuracy: The algorithm provides accurate time synchronization across all the client machines, reducing the chances of errors due to time discrepancies.

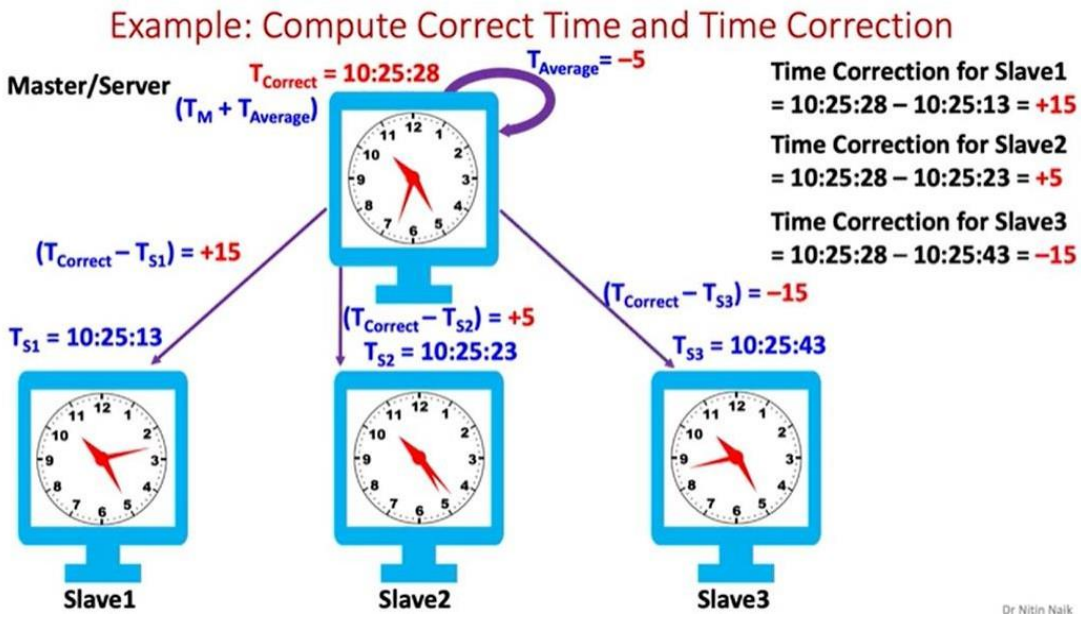
Scalability: The algorithm is scalable, as it can handle a large number of client machines, and the time coordinator can be easily replicated to provide high availability.

Security: Berkeley's Algorithm provides security mechanisms such as authentication and encryption to protect the time information from unauthorized access or tampering.









- This is an active time server approach where the time server periodically broadcasts its clock time and the other nodes receive the message to correct their own clocks.
- In this algorithm the time server periodically sends a message to all the computers in the group of computers. When this message is received each computer sends back its own clock value to the time server. The time server has a prior knowledge of the approximate time required for propagation of a message which

is used to readjust the clock values. It then takes a fault tolerant average of clock values of all the computers. The calculated average is the current time to which all clocks should be readjusted.

- The time server readjusts its own clock to this value and instead of sending the current time to other computers it sends the amount of time each computer needs for readjustment. This can be positive or negative value and is calculated based on the knowledge the time server has about the propagation of message.

2. DISTRIBUTED ALGORITHMS

Distributed algorithms overcome the problems of centralized by internally synchronizing for better accuracy. One of the two approaches can be used:

(i) GLOBAL AVERAGING DISTRIBUTED ALGORITHMS

- In this approach the clock process at each node broadcasts its local clock time in the form of a "resync" message at the beginning of every fixed-length resynchronization interval. This is done when its local time equals To+iR for some integer i, where To is a fixed time agreed by all nodes and R is a system parameter that depends on total nodes in a system.
- After broadcasting the clock value, the clock process of a node waits for time T which is determined by the algorithm.
- During this waiting the clock process collects the resync messages and the clock process records the time when the message is received which estimates the skew after the waiting is done. It then computes a fault-tolerant average of the estimated skew and uses it to correct the clocks.

(ii) LOCALIZED AVERAGING DISTRIBUTES ALGORITHMS

- The global averaging algorithms do not scale as they need a network to support broadcast facility and a lot of message traffic is generated.
- Localized averaging algorithms overcome these drawbacks as the nodes in distributed systems are logically arranged in a pattern or ring.
- Each node exchanges its clock time with its neighbors and then sets its clock time to the average of its own clock time and of its neighbors.

NTP

Network Time Protocol (NTP) is a protocol that helps the computers clock times to be synchronized in a network. This protocol is an application protocol that is responsible for the synchronization of hosts on a TCP/IP network. NTP was developed by David Mills in 1981 at the University of Delaware. This is required in a communication mechanism so that a seamless connection is present between the computers.

Features of NTP:

Some features of NTP are –

- NTP servers have access to highly precise atomic clocks and GPU clocks
- It uses Coordinated Universal Time (UTC) to synchronize CPU clock time.

- Avoids even having a fraction of vulnerabilities in information exchange communication.
- Provides consistent timekeeping for file servers

Working of NTP:

NTP is a protocol that works over the application layer, it uses a hierarchical system of time resources and provides synchronization within the stratum servers. First, at the topmost level, there is highly accurate time resources' ex. atomic or GPS clocks. These clock resources are called stratum 0 servers, and they are linked to the below NTP server called Stratum 1, 2 or 3 and so on. These servers then provide the accurate date and time so that communicating hosts are synced to each other.

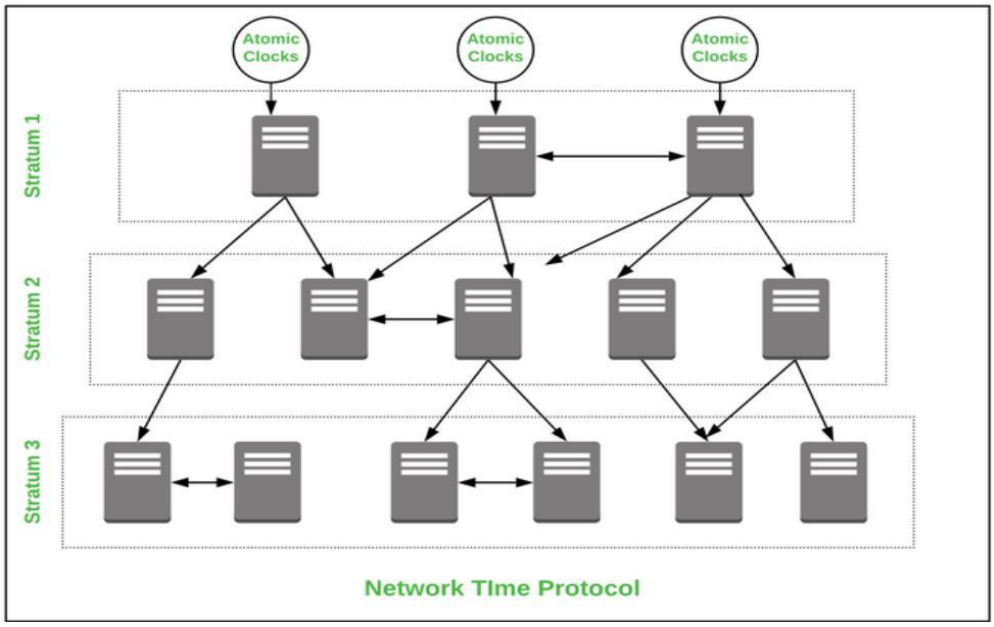


Figure: 10 Network Time Protocol

Applications of NTP:

- Used in a production system where the live sound is recorded.
- Used in the development of broadcasting infrastructures.
- Used where file system updates needed to be carried out across multiple computers depending on synchronized clock times.
- Used to implement security mechanism which depends on consistent time keeping over the network.
- Used in network acceleration systems which rely on timestamp accuracy to calculate performance.

Advantages of NTP:

- It provides internet synchronization between the devices.
- It provides enhanced security within the premises.
- It is used in the authentication systems like Kerberos.
- It provides network acceleration which helps in troubleshooting problems.
- Used in file systems that are difficult in network synchronization.

Disadvantages of NTP:

- When the servers are down the sync time is affected across a running communication.

- Servers are prone to error due to various time zones and conflict may occur.
- Minimal reduction of time accuracy.
- When NTP packets are increased synchronization is conflicted.
- Manipulation can be done in synchronization.

MESSAGE ORDERING AND GROUP COMMUNICATION

- For any two events a and b, where each can be either a send or a receive event, the notation
- a ~ b denotes that a and b occur at the same process, i.e., a ∈ Ei and b ∈ Ei for some process i. The send and receive event pair for a message called pair of corresponding events.
- For a given execution E, let the set of all send–receive event pairs be denoted as
- ↗= {(s,r) ∈ Ei × Ej | s corresponds to r}.

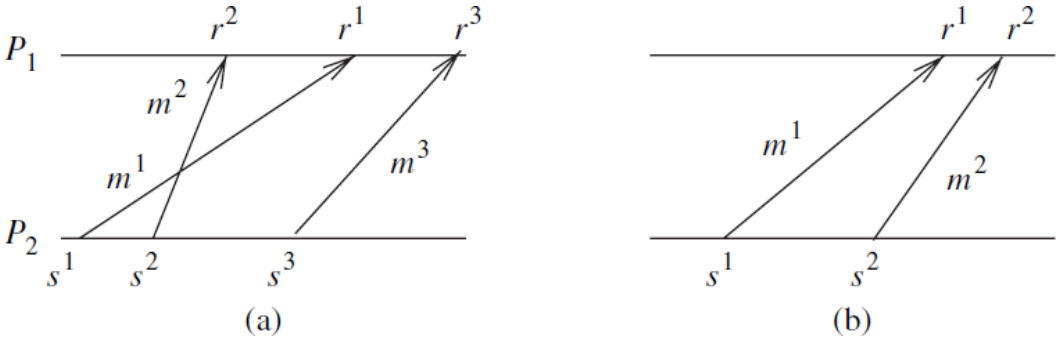
MESSAGE ORDERING PARADIGMS

- DISTRIBUTED program logic greatly depends on the order of delivery of messages.
- Several orderings on messages have been defined: (i) non-FIFO, (ii) FIFO, (iii) causal order, and (iv) synchronous order.

1.1 Asynchronous executions

Definition 1.1 (A-execution): An asynchronous execution (or A-execution) is an execution (E,≺) for which the causality relation is a partial order.

- On a logical link between two nodes (is formed as multiple paths may exist) in the system, if the messages are delivered in any order then it is known as non-FIFO executions. Example: IPv4.
- Each physical link delivers the messages sent on it in FIFO order due to the physical properties of the medium.



(Figure :1.1 Illustrating FIFO and non-FIFO executions. (a) An A-execution that is not a FIFO execution. (b) An A- execution that is also a FIFO execution.)

FIFO executions

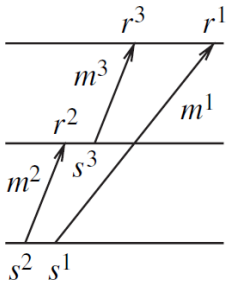
Definition 1.2 (FIFO executions): A FIFO execution is an A-execution in which, for all (s,r) and (s′,r′) ∈ T, (s ∼ s′ and r ∼ r′ and s ≺ s′) ⇒ r ≺ r′.In general on any logical link, messages are delivered in the order in which they are sent.

- To implement FIFO logical channel over a non-FIFO channel, use a separate numbering scheme to sequence the messages.
- The sender assigns and appends a <sequence_num, connection_id> 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.
- Figure 1.1(b) illustrates an A-execution under FIFO ordering.

1.2 Causally ordered (CO) executions

Definition 1.3 (Causal order (CO)): A CO execution is an A-execution in which, for all (s, r) and (s′, r′) ∈ T, (r ∼ r′ and s ≺ s′) ⇒ r ≺ r′.

- If two send events s and s′ are related by causality ordering, then their corresponding receive events r and r′ must occur in the same order at all common destinations.
- Figure 1.3 shows an execution that satisfies CO. s2 and s1 are related by causality but the destinations of the corresponding messages are different. Similarly for s2 and s3.



(Fig:1.3 CO executions)

- **Applications of Causal order:** applications that requires update to shared data, to implement distributed shared memory, and fair resource allocation in distributed mutual exclusion.

Definition 1.4 (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.

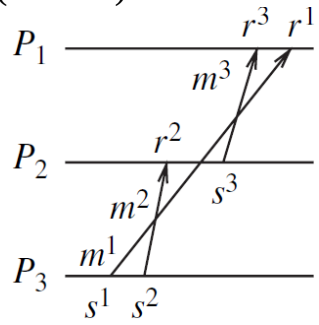
- If m^1 and m^2 are sent by the same process, then property degenerates to FIFO property.
- In a FIFO execution, no message can be overtaken by another message between the same (sender, receiver) pair of processes.
- In a CO execution, no message can be overtaken by a chain of messages between

the same (sender, receiver) pair of processes.

Definition 1.5 (Message order (MO)): A MO execution is an A-execution in which, for all (s, r) and (s′, r′) ∈ T, $s \prec s' \Rightarrow \neg(r' \prec r)$.

- Example: Consider any message pair, say m^1 and m^3 in Figure 1.5(a). $s1 \prec s3$ but ¬

(r3 ≺ r1) is false. Hence, the execution does not satisfy MO.



(a)

Figure 1.5 (a) Not a CO execution.

Definition 1.6 (Empty-interval execution) An execution (E,≺) is an empty-interval (EI) execution if for each pair of events (s, r) ∈ ↗, the open interval set {x ∈ E | s ≺ x ≺ r} in the partial order is empty.

- **Example** Consider any message, say m^2, in Figure 1.6(b). There does not exist any event x such that $s^2 \prec x \prec r^2$. This holds for all messages in the execution. Hence, the execution is EI.

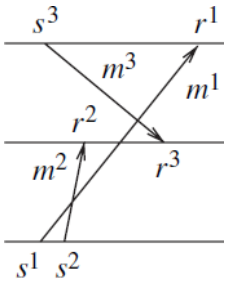


Figure:1.6(b) CO Execution

Corollary: An execution (E,≺) is CO if and only if for each pair of events (s,r) ∈ ↗ and each event e ∈ E,

• weak common past: $e \prec r \Rightarrow \neg(s \prec e)$

• weak common future: $s \prec e \Rightarrow \neg(e \prec r)$.

1.3 Synchronous execution (SYNC)

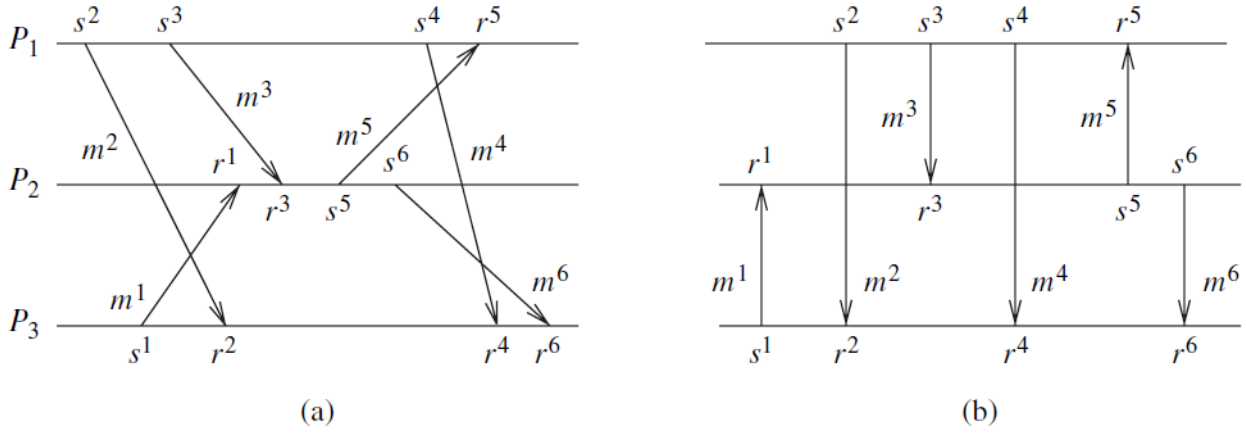
- When all the communication between pairs of processes uses synchronous send and receive primitives, the resulting order is the synchronous order.

- As each synchronous communication involves a handshake between the receiver and the sender, the corresponding send and receive events can be viewed as occuring instantaneously and atomically.
- In a timing diagram, the "instantaneous" message communication can be shown by bidirectional vertical message lines.
- The "instantaneous communication" property of synchronous executions requires that two events are viewed as being atomic and simultaneous, and neither event precedes the other.

Definition 1.7 (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 $(s, r) \in T$, then for all $x \in E$, $[(x \ll s \Longleftrightarrow x \ll r)$ and $(s \ll x \Longleftrightarrow r \ll x)]$.
- S3: If $x \ll y$ and $y \ll z$, then $x \ll z$.
- We can now formally define a synchronous execution.



(Figure 1.7 Illustration of a synchronous communication. (a) Execution in an asynchronous system. (b)

Equivalent instantaneous communication.)

Definition 1.8 (S- execution): A synchronous execution is an execution (E, $\ll$) for which the causality relation $\ll$ is a partial order.

- **Timestamping a synchronous execution:** An execution $(E,\prec)$ 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 Pi, if $ei \prec ei'$ then $T(ei) < T(ei')$.

ASYNCHRONOUS EXECUTION WITH SYNCHRONOUS COMMUNICATION

- When all the communication between pairs of processes is using synchronous send and receive primitives, then the resulting order is synchronous order.
- A distributed program that run correctly on an asynchronous system may not be executed by synchronous primitives. There is a possibility that the program may

deadlock, as shown by the code in the below Figure 10.

Process i	**Process** j
...	...
$Send(j)$	$Send(i)$
$Receive(j)$	$Receive(i)$
...	...

Figure.10 A communication program for an asynchronous system deadlocks when using
synchronous primitives.

- **Examples:** In Figure 11(a-c) using a timing diagram, will deadlock if run with

synchronous primitives.

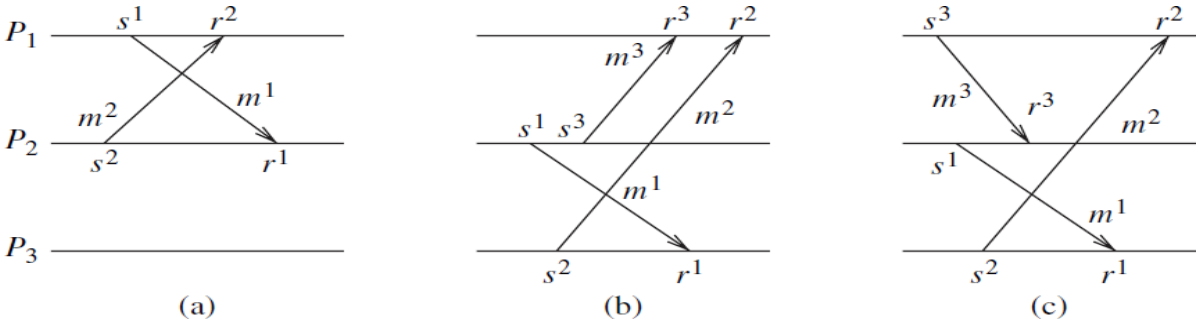


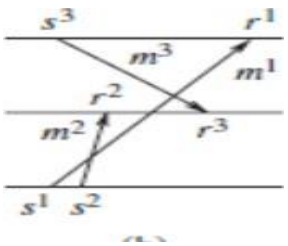
Figure 11 Illustrations of asynchronous executions and of crowns. (a) Crown of size 2. (b) Another crown of size 2. (c) Crown of size 3.

1. Executions realizable with synchronous communication (RSC)

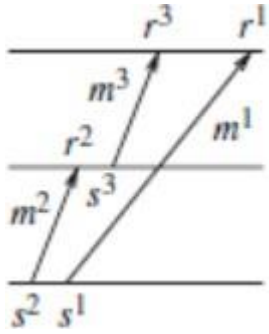
- In an A-execution, the messages can be made to appear instantaneous if there exists a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event. Such an A-execution that is realized under synchronous communication is called a realizable with synchronous communication (RSC) execution.

Non-separated linear extension: A non-separated linear extension of $(E, \prec)$ is a linear extension of $(E, \prec)$ such that for each pair $(s, r) \in T$, the interval $\{ x \in E \mid s \prec x \prec r\}$ is empty **Example:**

- In the below figure: $<s^2, r^2, s^3, r^3, s^1, r^1>$ is a linear extension that is non separated.



- In the below figure <s^2, s^1, r^2, s^3, r^3, s^1> is a linear extension that is separated.



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

Crown: Let E be an execution. A crown of size k in E is a sequence <(s^i, r^i), i ∈ {0, ..., k−1}> of pairs of corresponding send and receive events such that: $s0 \prec r^1$, $s^1 \prec r^2$,... , $s^{k-2} \prec r^{k-1}$, $s^{k-1} \prec r^0$.

- On the set of messages T, we define an ordering ↪ such that m ↪ m' if and only if $s \prec r'$.

2. Hierarchy of ordering paradigms

The orders of executions are:

- Synchronous order (SYNC)
- Causal order (CO)
- FIFO order (FIFO)
- Non FIFO order (non-FIFO)

The Execution order have the following results

– For an A-execution, A is RSC if and only if A is an S-execution.

– RSC ⊂ CO ⊂ FIFO ⊂ A

– This hierarchy is illustrated in Figure (a), and example executions of each class are shown side-by-side in Figure (b)

– The above hierarchy implies that some executions belonging to a class X will not belong to any of the classes included in X. The degree of concurrency is most in A and least in SYNC.

– A program using synchronous communication is easiest to develop and verify.

– A program using non-FIFO communication, resulting in an A execution, is hardest to design and verify.

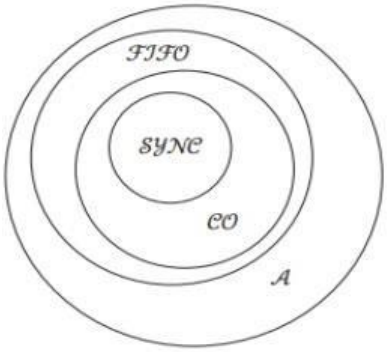


Fig (a)

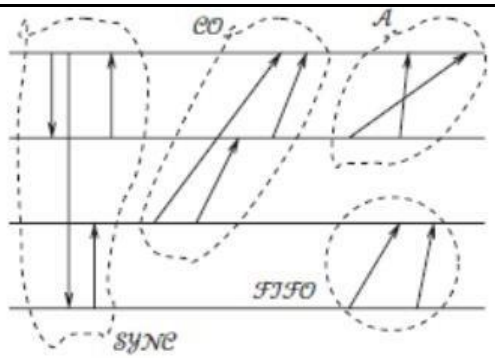
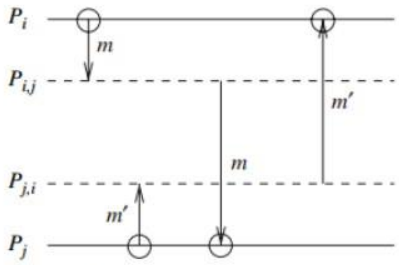


Fig (b)

3. Simulations

- The events in the RSC execution are scheduled as per some non-separated linear extension, and adjacent (s, r) events in this linear extension are executed sequentially in the synchronous system.
- The partial order of the asynchronous execution remains unchanged.
- If an A-execution is not RSC, then there is no way to schedule the events to make them RSC, without actually altering the partial order of the given A-execution.
- However, the following indirect strategy that does not alter the partial order can be used.
- Each channel $C_{i,j}$ is modeled by a control process $P_{i,j}$ that simulates the channel buffer.
- An asynchronous communication from i to j becomes a synchronous communication from i to $P_{i,j}$ followed by a synchronous communication from $P_{i,j}$ to j.
- This enables the decoupling of the sender from the receiver, a feature that is essential in asynchronous systems.



Modeling channels as processes to simulate an execution using asynchronous primitives on synchronous system

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.

SYNCHRONOUS PROGRAM ORDER ON AN ASYNCHRONOUS SYSTEM

There do not exist real systems with instantaneous communication that allows for synchronous communication to be naturally realized.

Non-determinism

- This suggests that the distributed programs are deterministic, i.e., repeated runs of the same program will produce the same partial order.
- In many cases, programs are non-deterministic in the following senses

  1. A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.

  2. 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, it results in a deadlock.
- However, there is no semantic dependency between the send and immediately following receive. If the receive call at one of the processes is scheduled before the send call, then there is no deadlock.
- **Rendezvous**

Rendezvous ("meet with each other")

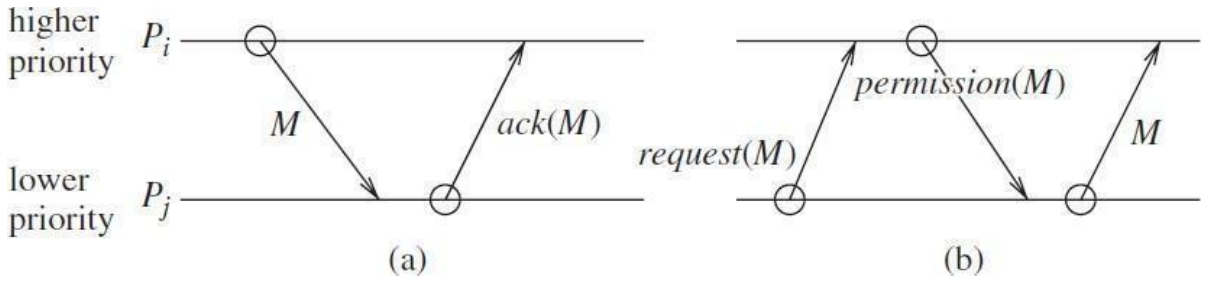
- One form of group communication is called multiway rendezvous, which is a synchronous communication among an arbitrary number of asynchronous processes.
- Rendezvous between a pair of processes at a time is called binary rendezvous as opposed to the multiway rendezvous.
- Observations about synchronous communication under binary rendezvous:
  - For the receive command, the sender must be specified eventhough the multiple recieve commands exist.
  - Send and received commands may be individually disabled or enabled.
  - Synchronous communication is implemented by scheduling messages using asynchronous communication.
- Scheduling involves pairing of matching send and receive commands that are both enabled.
- The communication events for the control messages do not alter the partial order of execution.

Algorithm for binary rendezvous

- Each process, has a set of tokens representing the current interactions that are enabled locally. If multiple interactions are enabled, a process chooses one of them and tries to "synchronize" with the partner process.
- The scheduling messages must satisfying the following constraints:

  - Schedule on-line, atomically, and in a distributed manner.
  - Schedule in a deadlock-free manner (i.e., crown-free), such that both the sender and receiver are enabled for a message when it is scheduled. Schedule to satisfy the progress property (i.e., find a schedule within a bounded number of steps) in addition to the safety (i.e., correctness) property.

- Additional features of a good algorithm are:
  (i) symmetry or some form of fairness, i.e., not favoring particular processes
  (ii) efficiency, i.e., using as few messages as possible
- A simple algorithm by **Bagrodia,** makes the following assumptions:
  1. Receive commands are forever enabled from all processes.
  2. A send command, once enabled, remains enabled until it completes.
  3. To prevent deadlock, process identifiers are used to break the crowns.
  4. Each process attempts to schedule only one send event at any time.
- The algorithm illustrates how crown-free message scheduling is achieved on-line.



Messages used to implement synchronous order. Pi has higher priority than Pj . (a) Pi issues SEND(M).

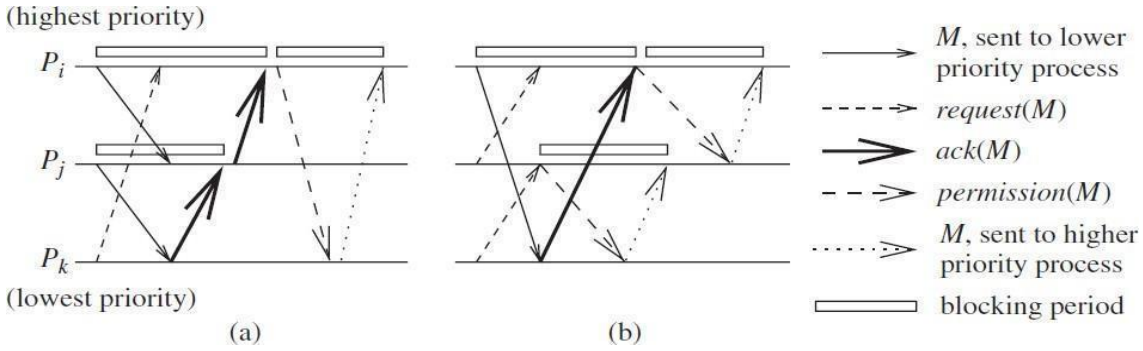
(b) The message types used are:Pj issues SEND(M).

(i) M – Message is the one i.e., exchanged between any two process during execution

(ii) ack(M) – acknowledgment for the received message M ,

(iii) request(M) – when low priority process wants to send a message M to the high priority process it issues this command.

(iv) permission(M) – response to the request(M) to low priority process from the high priority process.



(Examples showing how to schedule messages sent with synchronous primitives)

- A cyclic wait is prevented because before sending a message M to a higher priority process, a lower priority process requests the higher priority process for

permission to synchronize on M, in a non-blocking manner.

-
- While waiting for this permission, there are two possibilities:

1. If a message M′ from a higher priority process arrives, it is processed by a receive and ack(M′) is returned. Thus, a cyclic wait is prevented.
2. Also, while waiting for this permission, if a request(M′) from a lower priority process arrives, a permission(M′) is returned and the process blocks until M′ actually arrives.

Algorithm 1 A simplified implementation of synchronous order.Code shown is for process Pi , $1 \leq i \leq n$.

(message types)

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

(1) Pi **wants to execute SEND(**M**) to a lower priority process** Pj**:**

- Pi executes send(M) and blocks until it receives ack(M) from Pj. 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) Pi **wants to execute SEND(**M**) to a higher priority process** Pj**:**

(2a) Pi seeks permission from Pj by executing send(request(M)). (2b) While Pi is waiting for permission, it remains unblocked.

(i) If a message M′ arrives from a higher priority process Pk, Pi accepts M′ by scheduling a RECEIVE(M′) event and then executes send(ack(M′)) to Pk.

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

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

(3) **request(**M**) arrival at** Pi **from a lower priority process** Pj**:**

At the time a request(M) is processed by Pi, process Pi executes send(permission(M)) to Pj 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** Pi **from a higher priority process** Pj**:**

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

to Pj .

(5) **Processing when** Pi **is unblocked:**

When Pi 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).

GROUP COMMUNICATION

- Processes across a distributed system cooperate to solve a task. Hence there is need for 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 a message broadcast is sending a message to all members.
- In Multicasting the message is sent to a certain subset, identified as a group.
- In unicasting is the point-to-point message communication.

A group is a collection of processes that act together in some system or user specified way.The key property that all groups have that when a message is sent to the group; itself all the members of the group receive it.

Group communication can be implemented in several ways.

1.One – to – many 2. Many – to – one 3. Many – to – many

1. One – to - many

- a. Group management
- b. Group Addressing
- c. Message delivery to receiving process
- d. Buffered and unbuffered multicast
- e. Flexible reliability in multicast communication
- f. Atomic multicast
- g. Group communication primitives

Single sender and multiple receiver is also known as multicast communication. Broadcast is a special case of multicast communication.

- **a. Group Management**

In case of group communication, the communicating processes forms a group. Such a group may of either of 2 types.

Closed group is the one, in which only member can send message, outside process cannot send message to the group as whole but can send to single member of a group.

Open group is one in which any process in the system can send the message to group as a whole.

Message passing provides flexibility and create groups dynamically and to allow process join or leave group dynamically. Centralized group server is used for this purpose but suffers from poor reliability and scalability.

- **b. Group Addressing**

Two level naming scheme is used for group addressing.

Multicast address is a special network address; packet is delivered automatically here to all machines listening to address.

Broadcast address the all machines have to check if packet is intended for it or else simply discard so broadcast is less efficient.

- • If network doesn't support any addressing among two then one to one communications is used.
- First two methods send single packet but one to one sends many, creating much traffic.

- **c. Message delivery to receiving process**

User application uses high level group names in program. Centralized group server maintains a mapping of high level group names to their low level names, when sender sends message to the group. With high level name, kernel of server machines asks the group servers low level name and list of process identifiers.

When packet reaches m/c kernel, that m/c extract list of process IDs and forwards message to those processes belonging to its own machine. If none of ID's is matching packet is discarded.

- **d. Buffered and unbuffered multicast**

Send to all semantics:- A copy of message is sent to each process of multicast group and message is buffered until it is accepted by process.

Bulletin board semantics:- Message to be multicast is addressed to channel instead of each process.

e. Flexible Reliability in Multicast communication.

In multicast mechanism there is flexibility for user definable reliability. Degree of reliability is expressed in 4 forms.

a) O – Reliable: - No response is expected at all.

b) I – Reliable: - Sender expects response from any one of the receiver.

c) mount of n reliable :- Sender expects response from m of n receiver.

d) All reliable : - Sender expects response from all.

f. Atomic Multicast

• All or nothing property all reliable form requires atomic multicast facility.

• Message passing system should support both atomic and non- atomic multicast facility.

• It should provide flexibility to sender to specify whether atomicity is required or not.
• It is difficult to implement atomic multicast if sender or receiver fails.

• Solution to this is fault tolerated atomic multicast protocol.

• In this protocol, each message has message identifier field to distinguish message and one field to indicate data message in atomic multicast message.

g. Group communication primitives

• In one to one and one to many send or has to specify destination address and pointer to data so some send primitives can be used for both.

• But some systems use send group primitives because

1. It simplifies design and implementation.

2. It provides greater flexibilities.

(2) Many – to – one

• Multiple sender single receiver

• Single receiver may be selective or non-selective

• **Selective receiver** specifies unique sender message exchange takes place only if the sender sends the message.

• **Non-selective receiver** specifies set of sender if any from that sends message then message exchange takes place.

• No determinism issue need to be handled here.

• Rest all factors are same as for one – to – many communication.

(3) Many –to – many

• Multiple sender send messages to multiple receivers.

•Ordered message delivery ensures that all messages are delivered to all receivers in an order acceptable to the application.

For example: Suppose two senders send messages to update the same record of a database to two server processes having a replica of the database. If the message of the two sender is received by the two servers in different orders, then the final values of the updated record of the database may be different in its two replicas.

• Ordered message delivery/ event ordering are,

1. Absolute ordering
2. Consistent ordering/Total ordering
3. Causal ordering
4. FIFO Ordering

1) Absolute Ordering

• It ensures that all message delivered to all receiver in exact in which they are sent.

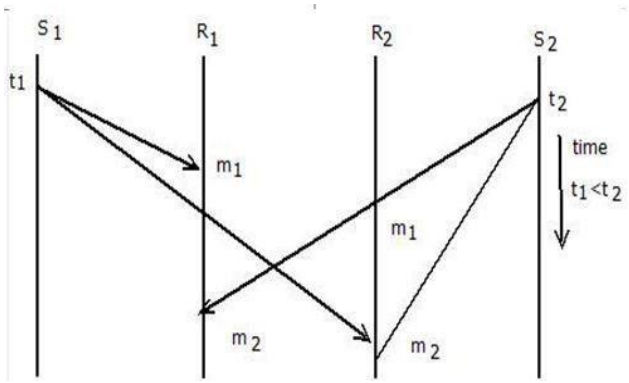
• Used to implement is to use global timestamps as message id.

• System is assumed to have clock synchronization and when sender message timestamps is taken as id of message and embedded in message.

• Kernel of receiver m/c, saves all incoming message in separate queue.

• A sliding window is used to periodically deliver message from receiver i.e fixed time interval is selected as window size.

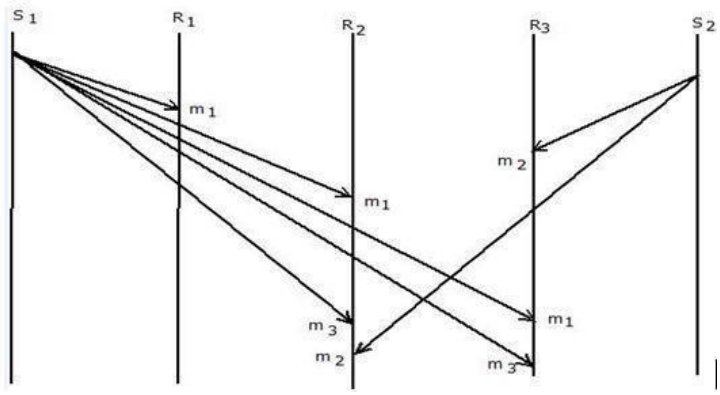
• Window size and properly chosen by considering maximum time for message to kernel from one m/c to other in n/w.



2. Consistent Ordering/ Total ordering

• For some application weaker semantics acceptable.

• Such semantic is called casual ordering.



• This semantics ensures that is the sending one message is casually related to event of sending another message, two messages are delivered to all receiver in correct order, otherwise these may be delivered in any order.

• Two message sending events are said to be casually related if they are correlated by happened before relation.

• Two message sending events are casually related if there is any possibilities of second one being influenced in a way by first one.

- Broadcast and multicast is supported by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information.
- However, the hardware or network layer protocol assisted multicast cannot efficiently provide the following 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.
- If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a closed group algorithm.
- If the sender of the multicast can be outside the destination group, then the multicast algorithm is said to be an open group algorithm.
- Open group algorithms are more general, and therefore more difficult to design and more expensive to implement, than closed group algorithms.
- Closed group algorithms cannot be used in in a large system like on-line reservation or Internet banking systems where client processes are short-lived and in large numbers.
- For multicast algorithms, the number of groups may be potentially exponential, i.e., $O(2^n)$.

CAUSAL ORDER

- Causal ordering of messages was proposed by Birman and joseph.
- Multiple sender send messages to multiple receivers.
- If send (m1) happens before send (m2) ,then every recipient must receive m1 before m2.
- If two message events are not causally related, the two messages may be delivered to the receivers in any orders.
- Two message sending events are said to be casually related if they are correlated by the happened before relation.

Consider Figure a, which shows two processes P1 and P2 that issue updates to the three replicas R1(d), R2(d), and R3(d) of data item d. Message m creates a causality between send(m1) and send(m2). If P2 issues its update causally after P1 issued its update, then P2's update should be seen by the replicas after they see P1's update, in order to preserve the semantics of the

Consider Figure a, which shows two processes P1 and P2 that issue updates to the three replicas R1(d), R2(d), and R3(d) of data item d. Message m creates a causality between send(m1) and send(m2). If P2 issues its update causally after P1 issued its update, then P2's update should be seen by the replicas after they see P1's update, in order to preserve the semantics of the application.

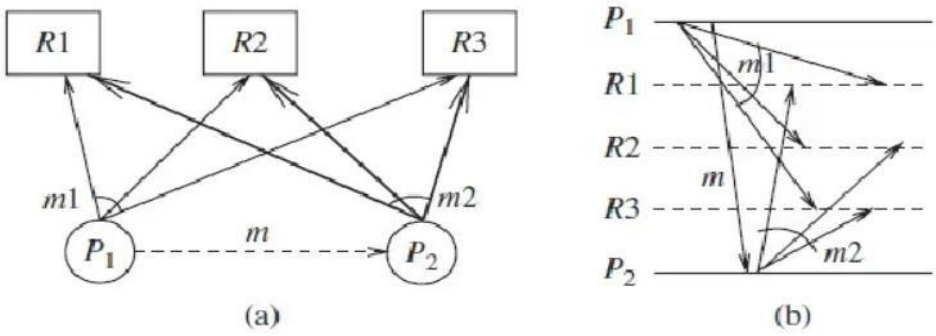


Figure (b) shows that R1 sees P2's update first, while R2 and R3 see P1's update first. Here, CO is violated.

Raynal-Schiper-Toueg Algorithm

- Each process maintain n x n matrix –the SENT matrix.
- SENT[i,j] is the number of messages sent by process pi to process pj.
- A process also maintain an array DELIV of size n where n represents number of messages delivered to p_i from all others.
- Every message transmitted by p_i is tagged with content of $SENT_i$.

- On receiving a message say m, process p_j can determine whether m can be delivered by comparing $DELIV_j$ with SENT matrix tagged m.if m can be delivered ,SENT matrix tagged m can be discarded.

```
(local variables)
array of int SENT[1 . . . n, 1 . . . n]
array of int DELIV[1 . . . n]          // DELIV[k] = # messages sent by k that are delivered locally

(1) send event, where Pi wants to send message M to Pj:
(1a) send (M, SENT) to Pj;
(1b) SENT[i, j] ← SENT[i, j] + 1.

(2) message arrival, when (M, ST) arrives at Pi from Pj:
(2a) deliver M to Pi when for each process x,
(2b)      DELIV[x] ≥ ST[x, i];
(2c) ∀x, y, SENT[x, y] ← max(SENT[x, y], ST[x, y]);
(2d) DELIV[j] ← DELIV[j] + 1.
```

TOTAL ORDER

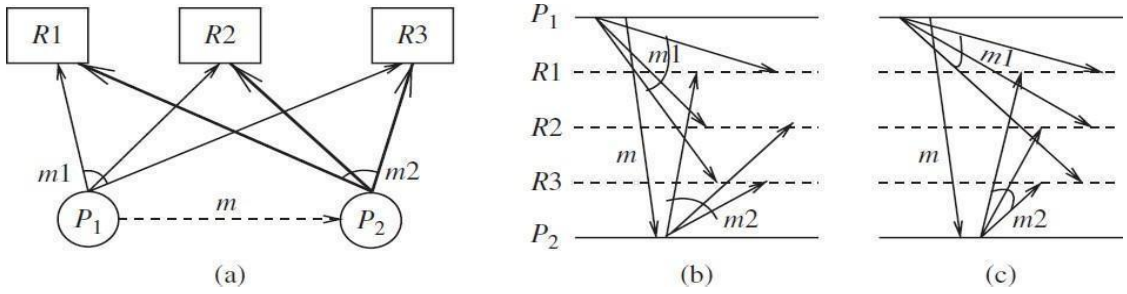
Total order, which requires that all messages be received in the same order by the recipients of the messages, is formally defined as follows:

Definition (Total order)

For each pair of processes Pi and Pj and for each pair of messages Mx and My that are delivered to both the processes, Pi is delivered Mx before My if and only if Pj is delivered Mx before My.

Example

- The execution in Figure (b) does not satisfy total order. Even if the message m did not exist, total



order would not be satisfied. The execution in Figure 6.11(c) satisfies total order.

1. Centralized algorithm for total order

- Algorithm Assumes all processes broadcast messages.
- It enforces total order and also the causal order in a system with FIFO channels.
- Each process sends the message it wants to broadcast to a centralized process.
- The centralized process relays all the messages it receives to every other process over FIFO channels.

Algorithm: centralized algorithm to implement total order & causal order of messages.

(1) When process Pi wants to multicast a message M to group G: (1a) **send** M(i,G) to central coordinator.

(2) When M(i,G) arrives from Pi at the central coordinator: (2a) **send** M(i,G) to all members of the group G.

(3) When M(i,G) arrives at Pj 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 therefore not an elegant solution

2. Three-phase distributed algorithm

- It enforces total and causal order for closed groups.
- The three phases of the algorithm are defined as follows:

Sender Phase 1

- A process multicasts the message M to the group members with the
  - a locally unique tag and
  - the local timestamp

Phase 2

- Sender process awaits for the reply from all the group members who respond with a tentative proposal for a revised timestamp for that message M.
- it is an non-blocking await i.e., any other messages received in the meanwhile are processed.
- Once all expected replies are received, the process computes the maximum of proposed timestamps for M, and uses the maximum as final timestamp.

Phase 3

- The process multicasts the final timestamp to the group members of phase 1.

Algorithm: Distributed algorithm to implement total order & causal order of messages.

Code at Pi, $1 \leq i \leq n$.

record Q_entry

M: **int**; // the application message
tag: **int**; // unique message identifier sender_id: **int**;// sender of the message
timestamp: **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 Pi, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts) // Phase 2 message sent by Pj , with revised timestamp, to

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

(1) When process Pi wants to multicast a message M with a tag tag: (1a) clock←clock+1;

(1b) **send** REVISE_TS(M, i, tag, clock) to all processes; (1c) temp_ts←0;

(1d) **await** PROPOSED_TS(j, i, tag, tsj) from each process Pj ; (1e) ∀j ∈ N, **do** temp_ts←max(temp_ts, tsj);

(1f) **send** FINAL_TS(i, tag, temp_ts) to all processes; (1g) clock←max(clock, temp_ts).

(2) When REVISE_TS(M, j, tag, clk) arrives from Pj : (2a) priority←max_priority+1(clk);

(2b) **insert** (M, tag, j, priority, undeliverable) in temp_Q; // at end of queue (2c) **send** PROPOSED_TS(i, j, tag_ priority) to Pj .

(3) When FINAL_TS(j, x, clk) arrives from Pj :

(3a) Identify entry Q_e in temp_Q, where Q_e.tag = x; (3b) **mark** Q_e.deliverable as true;

(3c) Update Q_e.timestamp to clk and re-sort temp_Q based on the timestamp field; (3d) **if (**head(temp_Q)).tag = Q_e.tag **then**

(3e) **move** Q_e **from** temp_Q **to** delivery_Q;

(3f) **while** (head(temp_Q)).deliverable is true **do**

(3g) **dequeue** head(temp_Q) and insert in delivery_Q.

(4) When Pi removes a message (M, tag, j, ts, deliverable) from head(delivery_Qi): (4a) clock←max(clock, ts)+1.

Receivers Phase 1

- The receiver receives the message with a tentative/proposed 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 (and the tag) back to the sender.
- The receiver then waits in a non-blocking manner for the final timestamp (correlated by the message tag).

Phase 3

- In the third phase, the final timestamp is received from the multicaster.
- 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.

- 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 in that order.

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 11. Here, A and B multicast to a set of destinations and C and D are the common destinations for both multicasts.

- Figure 11 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 timestamps 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 11(b)** The 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. 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.

9. 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.

10. 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.

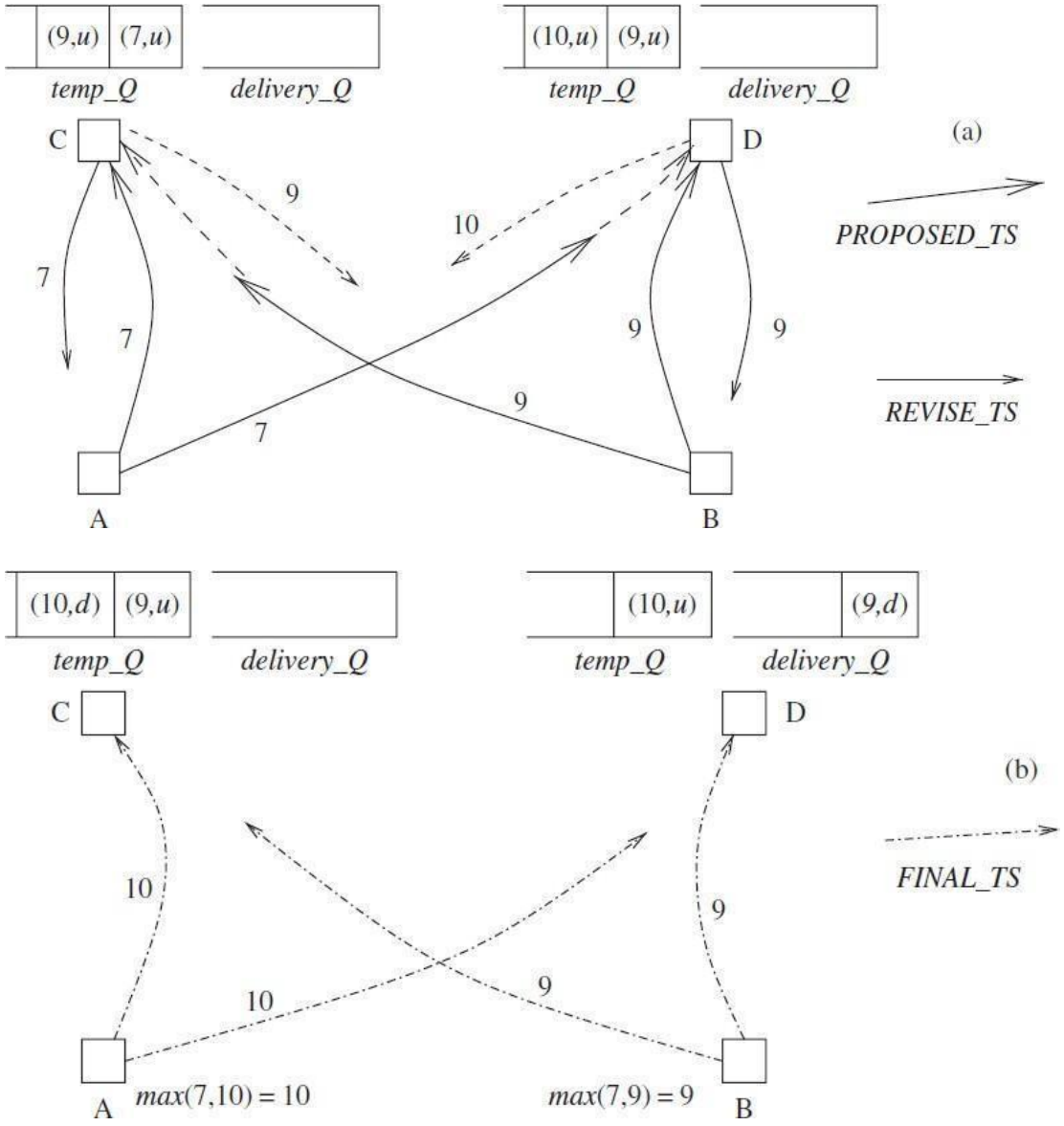


Figure 11 An example to illustrate the three-phase total ordering algorithm. (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages in the example.

GLOBAL STATE AND SNAPSHOT RECORDING ALGORITHMS

1. Introduction

- A distributed computing system consists of spatially separated processes that do not share a common memory and communicate asynchronously with each other by message passing over communication channels.
- Each component of a distributed system has a local state. The state of a process is the state of its local memory and a history of its activity.
- The state of a channel is the set of messages in the transit.
- The global state of a distributed system is the collection of states of the process and the channel.
- Applications that use the global state information are :
  - deadlocks detection
  - failure recovery,
  - for debugging distributed software
- If shared memory is available then an up-to-date state of the entire system is available to the processes sharing the memory.
- The absence of shared memory makes difficult to have the coherent and complete view of the system based on the local states of individual processes.
- A global snapshot can be obtained if the components of distributed system record their local states at the same time. This is possible if the local clocks at processes were perfectly synchronized or a global system clock that is instantaneously read by the processes.
- However, it is infeasible to have perfectly synchronized clocks at various sites as the clocks are bound to drift. If processes read time from a single common clock (maintained at one process), various indeterminate transmission delays may happen.
- In both cases, collection of local state observations is not meaningful, as discussed below.
- **Example:**
- Let S1 and S2 be two distinct sites of a distributed system which maintain bank accounts A and B, respectively. Let the communication channels from site S1 to site S2 and from site S2 to site S1 be denoted by C12 and C21, respectively.
- Consider the following sequence of actions, which are also illustrated in the timing
- diagram of Figure 4.1:
- Time t0: Initially, Account A=$600, Account B=$200, C12 =$0, C21=$0.

- Time t1: Site S1 initiates a transfer of $50 from A to
  B. Hence, A= $550, B=$200, C12=$50, C21=$0.
- Time t2: Site S2 initiates a transfer of $80 from Account B to
  A. Hence, A= $550,B=$120, C12 =$50, C21=$80.
- Time t3: Site S1 receives the message for a $80 credit to Account
  A. Hence, A=$630, B=$120, C12 =$50, C21 =$0.
- Time t4: Site S2 receives the message for a $50 credit to Account B. Hence, A=$630, B=$170, C12=$0, C21=$0.Suppose the local state of Account A is recorded at time t0 which is $600 and the local state of Account B and channels C12 and C21 are recorded at time t2 are $120,
  $50, and $80, respectively.
- Then the recorded global state shows $850 in the system. An extra $50 appears in the system.
- Reason: Global state recording activities of individual components must be coordinated.

SYSTEM MODEL AND DEFINITIONS

- The system consists of a collection of n processes, p_1, p_2,…,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 processstate.
- A **channel** is a distributed entity and its state depends on the local states of the processes on which it is incident.
  Transit: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j\}$
- 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, achannel acts like a set in which the sender process adds messages and thereceiver process removes messages from it in a random order.

- In causal delivery of messages satisfies the following property: "for any two messages mij and mkj,

  if send(mij) → send(mkj), then rec (mij) → rec(mkj)."
- Causally ordered delivery of messages implies FIFO message delivery.
- The causal ordering model is useful in developing distributed algorithms and may simplify the design of algorithms.

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. Notationally, global state GS is defined as GS ={UiLSi, Ui,jSCij}.
- A global state GS is a consistent global state iff it satisfies the following two conditions:

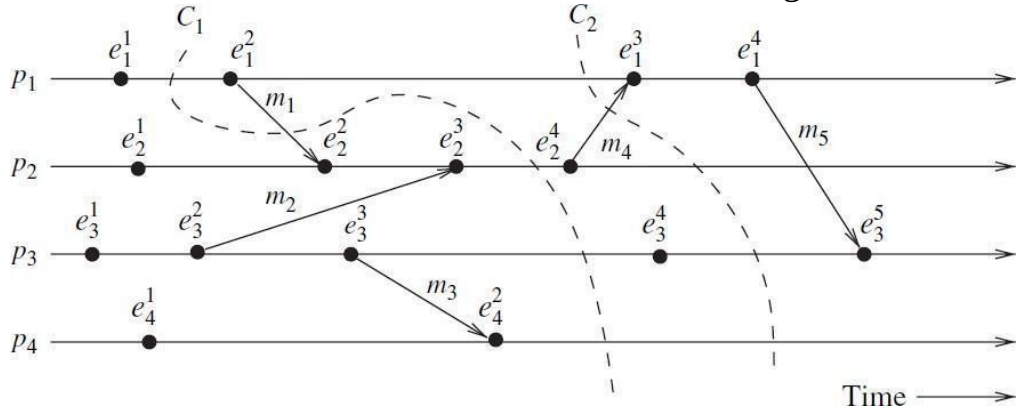
  C1: send(mij)∈LSi ⇒ mij ∈SCij ⊕ rec(mij)∈LSj (⊕ is the Ex-OR operator).

  C2: send(mij) ∉LSi ⇒ mij ∉ SCij A rec(mij) ∉ LSj .

- In a consistent global state, every message that is recorded as received is also recorded as sent. These are meaningful global states.
- The inconsistent global states are not meaningful ie., without send if receive of the respective message exists.

3. Interpretation in terms of cuts

- Cuts is a zig-zag line that connects a point in the space–time diagram at some arbitrary point in the process line.
- Cut is a powerful graphical aid for representing and reasoning about the global states of a computation.
- Left side of the cut is referred as PAST event and right is referred as FUTURE



event.

- 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. Example: Cut C2 in the above figure.

- All the messages that cross the cut from the PAST to the FUTURE are captured in the corresponding channel state.

If the flow is from the FUTURE to the PAST is inconsistent. **Example**: Cut C1.Issues in recording a global state

- If a global physical clock is used then the following simple procedure is used to record a consistent global snapshot of a distributed system.
  - Initiator of the snapshot decides a future time at which the snapshot is to be taken and broadcasts this time to every process.
  - All processes take their local snapshots at that instant in the global time.
  - The snapshot of channel Cij includes all the messages that process pj receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot.
- However, a global physical clock is not available in a distributed system. Hence the following two issues need to be addressed to record a consistent global snapshot.
- **I1:** How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?

  Answer:
  - Any message i.e., 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**).
- **I2:** How to determine the instant when a process takes its snapshot?

  Answer:

  A process pj must record its snapshot before processing a message mij that was sent by process pi after recording its snapshot.
- These algorithms use two types of messages: computation messages and control messages. The former are exchanged by the underlying application and the latter are exchanged by the snapshot algorithm.

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.

Defnition : 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

1.Chandy–Lamport algorithm

- This 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, 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**.
- Since all messages that follow a marker on channel Cij have been sent by process pi after pi has taken its snapshot, process pj must record its snapshot if not recorded earlier and record the state of the channel that was received along the marker message. This addresses issue **I2**.

Algorithm

- The algorithm is initiated by any process by executing the marker sending rule. The algorithm terminates after each process has received a marker on all of its incoming channels.
- **Algorithm** The Chandy–Lamport algorithm.

Marker sending rule for process pi

(1) Process pi records its state.

(2) For each outgoing channel C on which a marker has not been sent, pi sends a marker along C before pi sends further messages along C.

Marker receiving rule for process pj

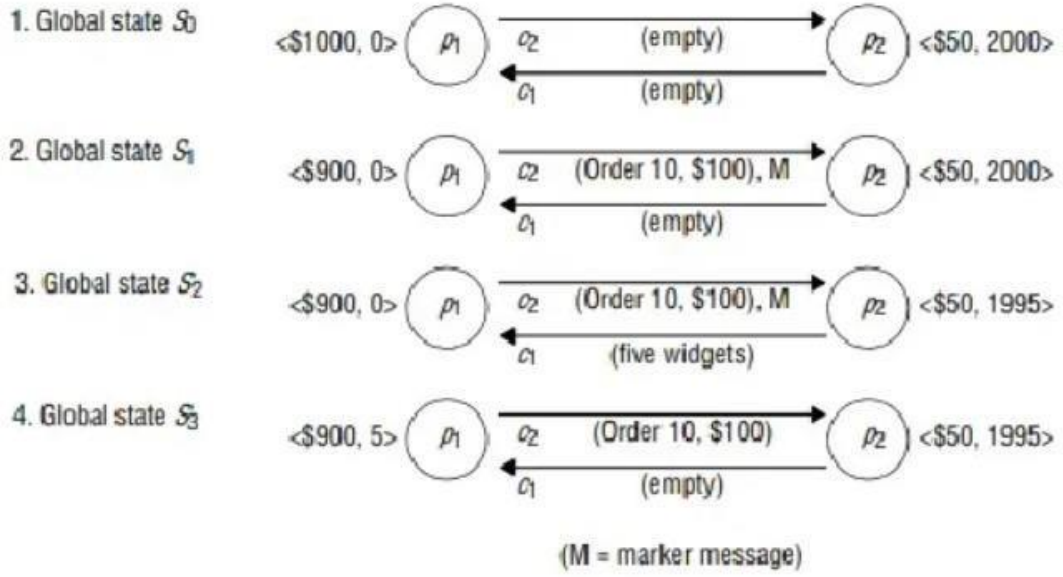
On receiving a marker along channel C:

if pj 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 pj,s state was recorded and before pj received the marker along C

Example



Process p1 records its state in the actual global state S0 , when the state of p1 is <$1000, 0>.Following the marker sending rule, process p1 then emits a marker message over its outgoing channel c2 before it sends the next application-level message: (Order 10, $100), over channel c2 . The system enters actual global state S1 . Before p2 receives the marker, it emits an application message (five widgets) over c1 in response to p1 's previous order, yielding a new actual global state S2 .Now process p1 receives p2 's message (five widgets), and p2 receives the marker. Following the marker receiving rule, p2 records its state as <$50, 1995> and that of channel c2 as the empty sequence. Following the marker sending rule, it sends a marker message over c1 .When process p1 receives p2 's marker message, it records the state of channel c1 as the single message (five widgets) that it received after it first recorded its state.The final recorded state is p1 : <$1000, 0>; p2 : <$50, 1995>; c1 : <(five widgets)>; c2 : < >. Note that this state differs from all the global states through which the system actually passed.

Correctness

- To prove the correctness of the algorithm, it is shown that a recorded snapshot satisfies conditions **C1** and **C2**.
- 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.

- Moreover, 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 pj receives message mij that precedes the marker on channel Cij, it acts as follows:
- If process pj has not taken its snapshot yet, then it includes mij in its recorded snapshot. Otherwise, it records mij in the state of the channel Cij. 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.

UNIT III DISTRIBUTED MUTEX AND DEADLOCK

Distributed Mutual Exclusion Algorithms: Introduction – Preliminaries – Lamport's algorithm – Ricart Agrawala'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.

DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

INTRODUCTION

- Mutual exclusion is a fundamental problem in distributed computing systems.
- Mutual exclusion ensures that concurrent access of processes to a shared resourceor data is serialized, that is, executed in mutually exclusive manner.

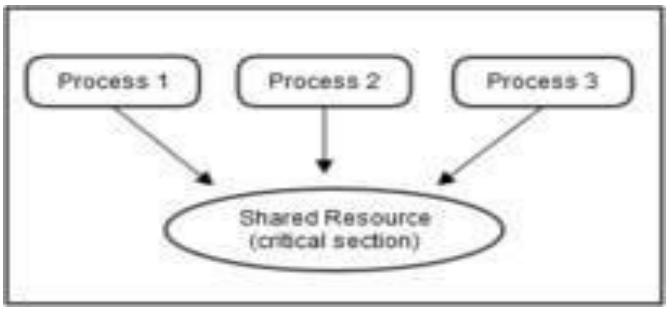


Figure 1: Three processes accessing a shared resource (critical section) simultaneously.

- 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**
2. **Non-token based approach**
3. **Quorum based approach**

1. In the **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 and it continues to hold the token until the execution of the CS is over. Mutual exclusion isensured because the token is unique.

2. In the **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. A site enters the **Critical Section (CS)** when an assertion, defined on its local variables, becomes true. Mutual Exclusion is enforced because the assertion becomes true only at one site at any given time.

3. In the **quorum-based approach**, each site requests permission to execute the CS from a subset of sites (called a quorum). The quorums are formed in such a way that when two sites concurrently request access to the CS, one site receives both the requests and which is responsible to make sure that only one request executes the CS at any time.

OBJECTIVES OF MUTUAL EXCLUSION ALGORITHMS

1. **Guarantee mutual exclusion** (required)
2. **Freedom from deadlocks** (desirable)
3. **Freedom from starvation** -- every requesting site should get to enter CS in a finitetime (desirable)
4. **Fairness** -- requests should be executed in the order of arrivals, which would be based on logical clocks (desirable)
5. **Fault tolerance** -- failure in the distributed system will be recognized and therefore not cause any unduly prolonged disruptions (desirable)

PRELIMINARIES

We describe here,

1. **System model.**
2. **Requirements of mutual exclusion algorithms.**
3. **Metrics we use to measure the performance of mutual exclusion algorithms.**

1. SYSTEM MODEL

- The system consists of N sites, **S1, S2, ..., SN.** We assume that a single process is running on each site.
- The process at site Si is denoted by pi.
- A process wishing to enter the CS, requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering theCS. While waiting the process is not allowed to make further requests to enter the CS.
- *A site can be in one of the following three states*:
  1. **Requesting the Critical Section.**
  2. **Executing the Critical Section.**
  3. **Neither requesting nor executing the CS (i.e., idle).**
- 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.

Note:

- We assume that channels reliably deliver all messages, sites do not crash, and the network does not get partitioned.
- Some mutual exclusion algorithms are designed to handle such situations. Many algorithms use Lamport-style logical clocks to assign a timestamp to critical section requests.
- Timestamps are used to decide the priority of requests in case of a conflict.
- A general rule followed is that the **smaller the timestamp of a request**, the **higher its priority to execute the CS.**

- We use the following notations:
  - **N** denotes the **number of processes or sites** involved in invoking the critical section,
  - **T** denotes the average **Message Time Delay**,
  - **E** denotes the average critical section **Execution Time**.

2. REQUIREMENTS OF MUTUAL EXCLUSION ALGORITHMS

A mutual exclusion algorithm should satisfy the following properties:

a. Safety Property: The safety property states that at any instant, only one process can execute the critical section. This is an essential property of a mutual exclusion algorithm.

b. Liveness Property: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive. In addition, a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS. That is, every requesting site should get an opportunity to execute the CS in finite time.

c. Fairness: Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS.

Note: The first property is absolutely necessary and the other two properties are considered important in mutual exclusion algorithms

3. PERFORMANCE METRICS

The performance of mutual exclusion algorithms is generally measured by the following four metrics:

a. Message complexity: It is the number of messages that are required per CS execution bya site.

b. Synchronization delay: After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 9.1).

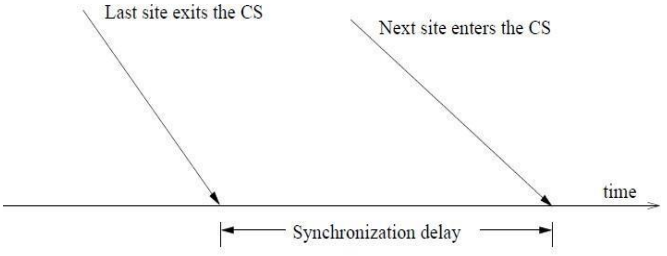


Figure 9.1: Synchronization Delay

c. Response time: It is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 9.2).

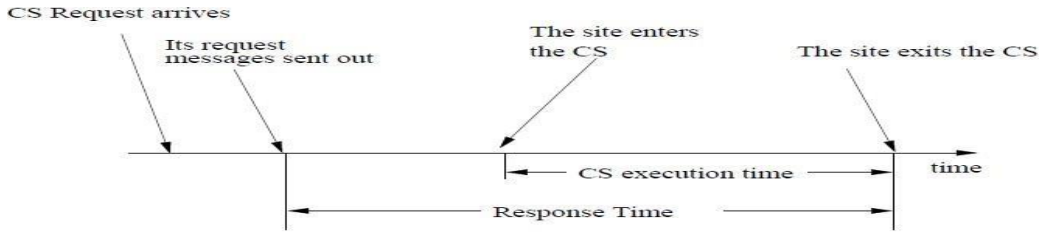


Figure 9.2: Response Time

Figure 2: Response Time

d. **System throughput:** It is the rate at which the system executes requests for the CS. If SD is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation:

System Throughput=1/ (SD+E)

Generally, the value of a performance metric fluctuates statistically from request to request and we generally consider the average value of such a metric.

Low and High Load Performance: The load is determined by the arrival rate of CS execution requests. Two special loading conditions, viz., **"low load"** and **"high load".**

- 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 ata site.

LAMPORT'S ALGORITHM

- The algorithm is fair in the sense that a request for CS is executed in the order of their timestamps and time is determined by logical clocks.
- When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp.
- The algorithm executes CS requests in the increasing order of timestamps.
- Every site Si keeps a queue, **request_queuei**,

This algorithm requires communication channels to deliver messages the FIFO order.

The Algorithm

1. Requesting the critical section:

- When a site Si wants to enter the CS, it broadcasts a **REQUEST (tsi, i)** message to all other sites and places the request on **request_queuei.** ((tsi, i) denotes the timestamp of the request.)
- When a site Sj receives the **REQUEST (tsi, i)** message from site Si, places site Si'sRequest on request_queuej and it returns a time stamped REPLY message to Si.

2. Executing the critical section:

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

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

L2: Si's request is at the top of request_queuei.

3. Releasing the critical section:

- Site Si, upon exiting the CS, removes its request from the top of its request queue and broadcasts a time stamped RELEASE message to all other sites.

- When a site Sj receives a RELEASE message from site Si, it removes Si's request from its request queue. When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. Clearly, when a site receives a REQUEST, REPLY or RELEASE message, it updates its clock using the timestamp in the message.

Correctness

Theorem 1: ***Lamport's algorithm achieves mutual exclusion.***

Proof: **Proof is by contradiction.** Suppose two sites Si and Sj 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 Si and Sj have their own requests at the top of their request_queues and condition L1 hold at them. Without loss of generality, assume that Si's request has smaller timestamp than the request of Sj. From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of Si must be present in request_queuej when Sj was executing its CS. This implies that Sj's own request is at the top of its own request_queue when a smaller timestamp request, Si's request, is present in there quest_queuej – a contradiction!! Hence, Lamport's algorithm achieves mutual exclusion.

Theorem 2: ***Lamport's algorithm is fair.***

Proof: A distributed mutual exclusion algorithm is fair if the requests for CS are executed in the order of their timestamps. The proof is by contradiction. Suppose a site Si's request hasa smaller timestamp than the request of another site Sj and Sj is able to execute the CS before Si. For Sj to execute the CS, it has to satisfy the conditions L1 and L2.

This implies thatat some instant in time say t, Sj 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_queueat a site is ordered by timestamp, and according to our assumption Si has lower timestamp. So Si's request must be placed ahead of the Sj's request in the request_queuej. This is a contradiction. Hence Lamport's algorithm is a fair mutual exclusion algorithm.

An Example

In Figures 9.3 to 9.6, we illustrate the operation of Lamport's algorithm. In Figure 9.3, site S1 and s2 are making a request for the CS and send out REQUEST messages to other sites. The time stamps of the requests are (1,2) and (2,1). In Figure 9.4, both the sites S1 and S2 have received REPLY messages from all other sites. S2 has its request at the top of its request_queue but site S1 does not have its request at the top of its request_queue. Consequently, site S2 enters the CS. In Figure 9.5, S2 exits and sends RELEASE messages to all other sites. In Figure 9.6, site S1 has received REPLY from all other sites and also received a RELEASE message from site S3. Site S1 updates its request_queue and its request is now at the top of its request_queue. Consequently, it enters the CS next.

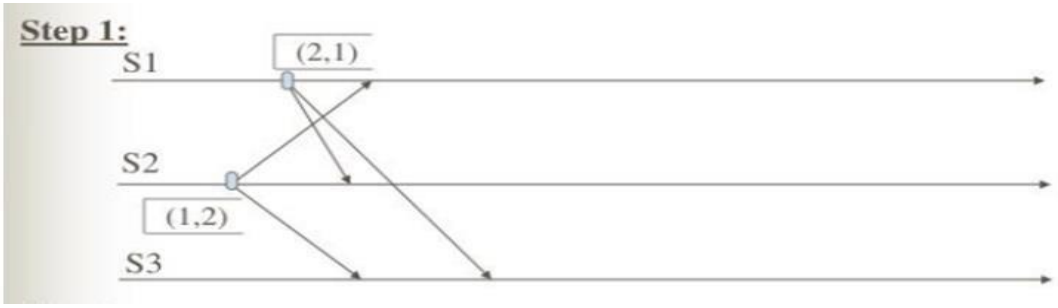


Figure: 9.3 S2 and S1 are sending requests for CS

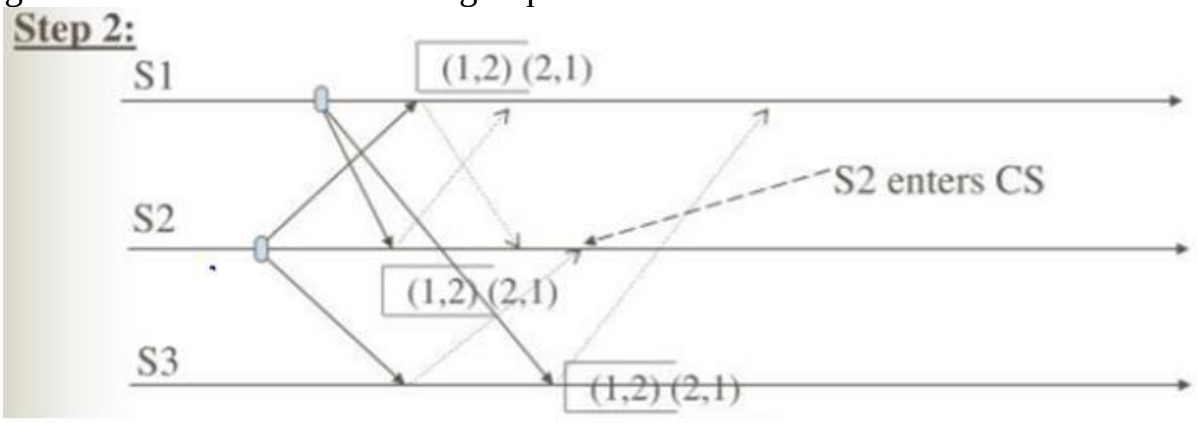


Figure: 9.4 After getting reply's S2 enters CS

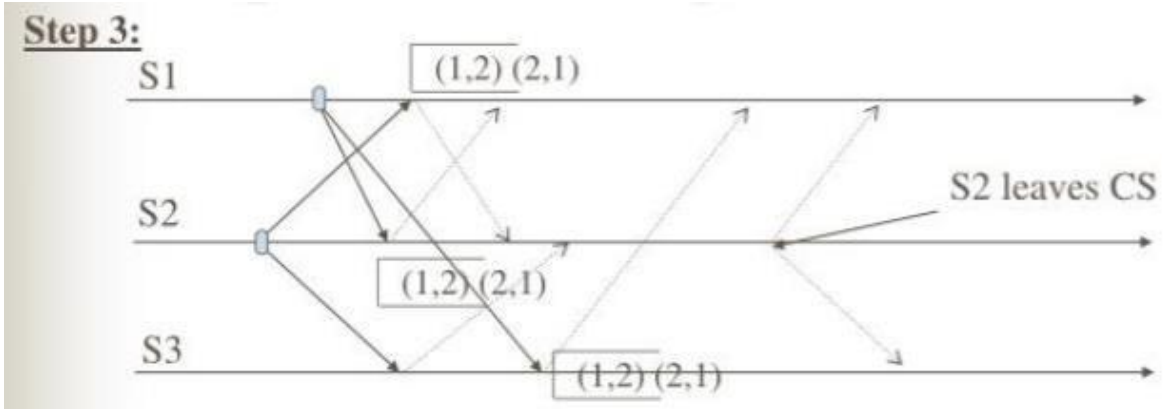


Figure: 9.5 S2 exits and sends RELEASE messages to all other sites

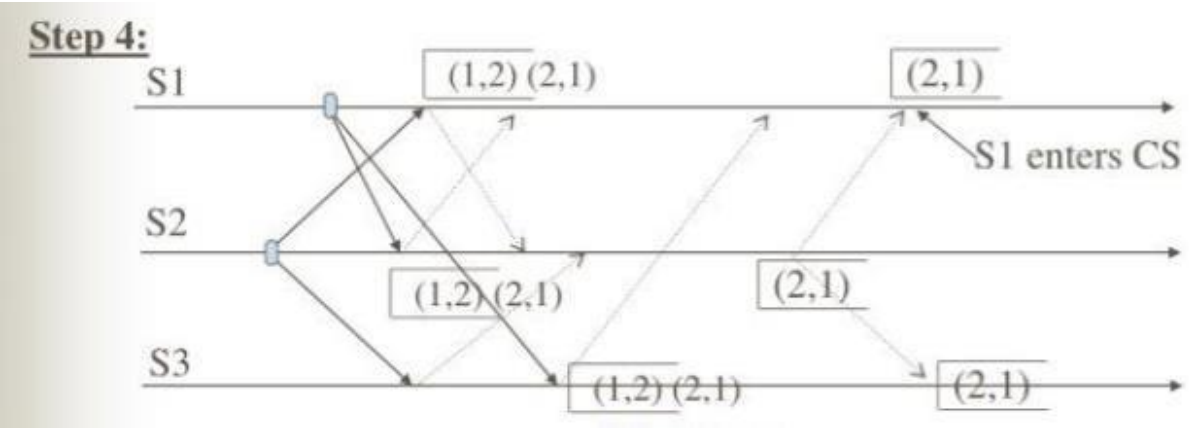


Figure: 9.6 Site S1 enters Cs

Performance

For each CS invocation

(N-1) REQUEST (N-1) REPLY

(N-1) RELEASE,

Total 3(N-1) messages, synchronization delay Sd = average delay

RICART-AGRAWALA ALGORITHM

The Ricart-Agrawala algorithm assumes the **communication channels are FIFO**.

- The algorithm uses two types of messages: **REQUEST** and **REPLY.**
- A process sends a REQUEST message to all other processes to request their permission to enter the critical section.
- A process sends a REPLY message to a process to give its permission to that process.
- **Processes use Lamport-style logical clocks to assign a timestamp** to critical section requests. Timestamps are used to decide the priority of requests in case of conflict – if a process pi that is waiting to execute the critical section, receives a REQUEST message from process pj, then if the priority of pj's request is lower, pi defers the REPLY to pj and sends a REPLY message to pj only after executing the CS for it spending request.
- Otherwise, pi sends a REPLY message to pj immediately, provided it is currently not executing the CS. Thus, if several processes are requesting execution of the CS, the highest priority request succeeds in collecting all the needed REPLY messages and gets to execute the CS.

Each process pi maintains the **Request-Deferred array**, RDi, the size of which is the same as the number of processes in the system. Initially, ∀i ∀j: RDi[j]=0. Whenever pi defer the request sent by pj, it sets RDi[j]=1 and after it has sent a REPLY message to pj, it sets RDi[j]=0. **Note:** Deferred – Postponed the request / waiting.

ALGORITHM

1. Requesting the critical section:

(a) When a site Si wants to enter the CS, it broadcasts a time stamped REQUEST message to all other sites.

(b) When site Sj receives a REQUEST message from site Si, it sends a REPLY message to Site Si if site Sj is neither requesting nor executing the CS, or if the site Sj is requestingAnd Si's request's timestamp is smaller than site Sj's own request's timestamp. otherwise, the reply is deferred and Sj sets RDj[i]=1

2. Executing the critical section:

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

3. Releasing the critical section:

(d) When site Si exits the CS, it sends all the deferred REPLY messages: ∀j if RDi[j]=1, then send a REPLY message to Sj and set RDi[j]=0.

When a site receives a message, it updates its clock using the timestamp in the message. Also, when a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request. In this algorithm, a site's REPLY messages are blocked only by sites which are requesting the CS with higher priority (i.e., smaller timestamp).Thus, when a site sends out differed REPLY messages, site with the next highest priority request receives the last needed REPLY message and enters the CS. Execution of the CS requests in this algorithm is always in the order of their timestamps.

An Example

Figures 9.7 to 9.10 illustrate the operation of Ricart-Agrawala algorithm. In Figure 9.7, sites S1 and S2 are making requests for the CS and send out REQUEST messages to other sites.The timestamps of the requests are (2, 1) and (1, 2), respectively. In Figure 9.8, S2 hasreceived REPLY messages from all other sites and consequently, it enters the CS.

In Figure 9.9, S2 exits the CS and sends a REPLY message to site S1.
In Figure 9.10, site S1 has received REPLY from all other sites and enters the CS next.

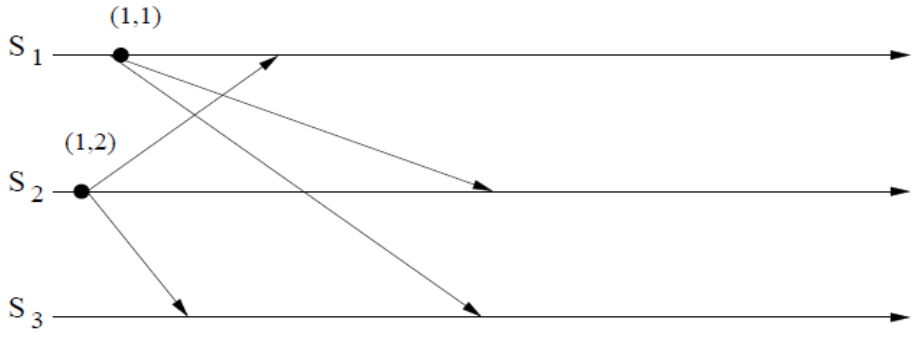


Figure 9.7: Sites S_1 and S_2 are making request for the CS

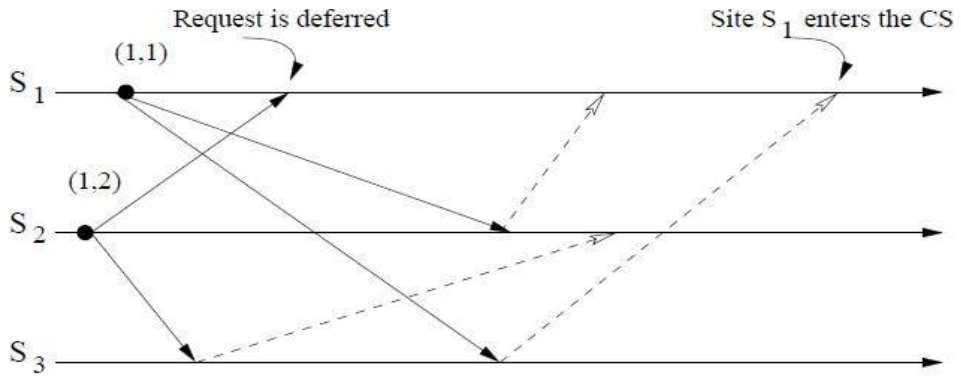


Figure 9.8: Site S_1 enters the CS

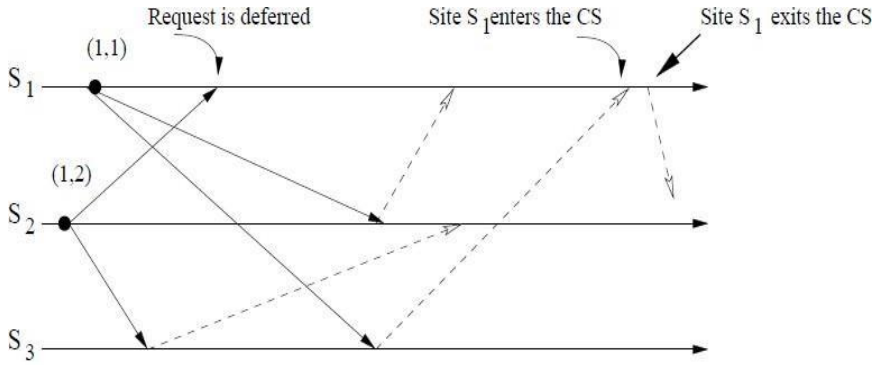


Figure 9.9: Site S_1 exits the CS and sends a REPLY message to S_2's deferred request

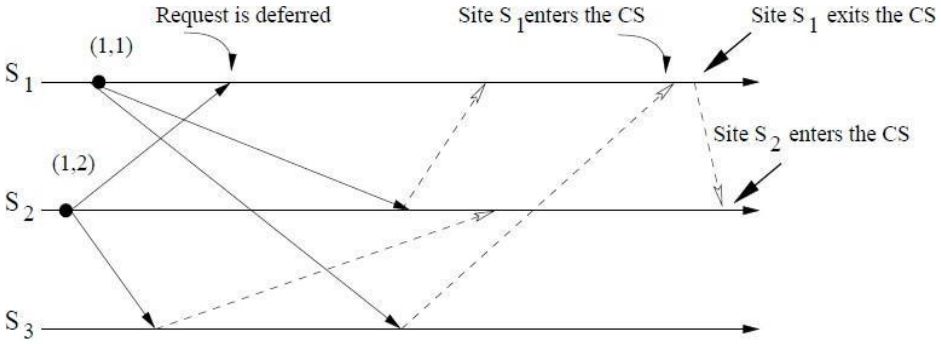


Figure 9.10: Site S_2 enters the CS

Performance

For each CS execution, Ricart-Agrawala algorithm requires (N − 1) REQUEST messages and (N−1) REPLY messages. Thus, it requires **2(N−1) messages per CS execution**. **Synchronization delay in the algorithm is T.**

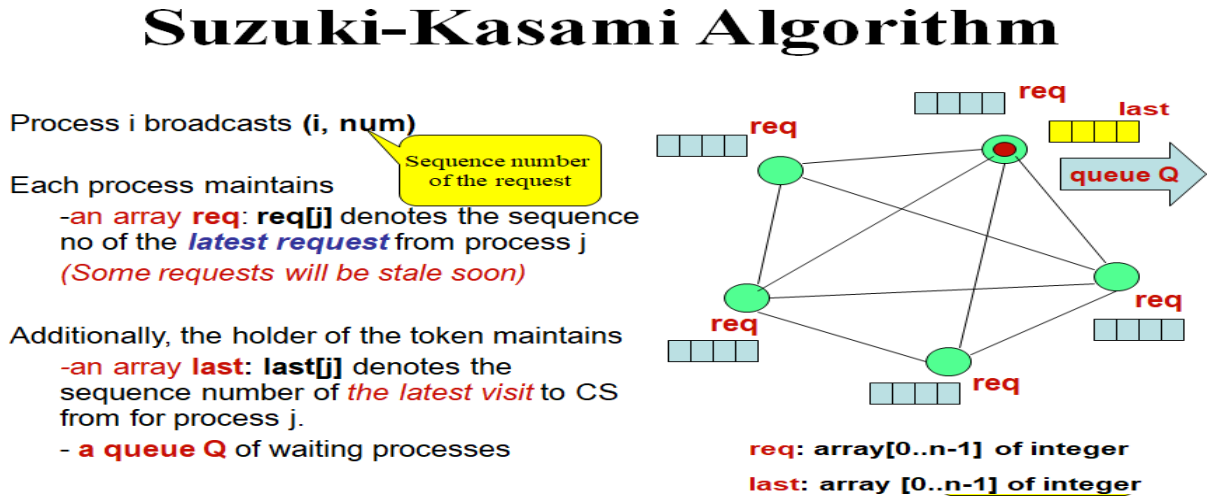
TOKEN-BASED ALGORITHMS

- One token, shared among all sites.
- Site can enter its CS if it holds token.

- The major difference is the way the token is searched.
- Use sequence numbers instead of timestamps.
  - Used to distinguish requests from same site.
  - Kept independently for each site.
- The proof of mutual exclusion is trivial.
- The proof of other issues (deadlock and starvation) may be less so.

SUZUKI-KASAMI'S BROADCAST ALGORITHM

- **Suzuki–Kasami's algorithm** is a token-based algorithm for achieving mutual exclusion in distributed systems.
- 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 request for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.



In Suzuki-Kasami's algorithm if a site that wants to enter the CS, does not have the token, it broadcasts a REQUEST message for the token to all other sites. A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

The basic idea underlying this algorithm may sound rather simple, however, there are the following two design issues must be efficiently addressed:

1. **How to distinguishing an outdated REQUEST message from a current REQUEST message:**

Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied. If a site cannot determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it. This will not violate the correctness; however, this may seriously degrade the performance by wasting messages and increasing the delay at sites that are genuinely requesting the token.

Therefore, appropriate mechanisms should implemented to determine if a token request message is outdated.

2. **How to determine which site has an outstanding request for the CS:** After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them. The problem is complicated because when a site Si receives a token request message from a site Sj, site Sj may have an outstanding request for the CS. However, after the corresponding request for the CS has been satisfied at Sj, an issue is how to inform site Si (and all other sites) efficiently about it. Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner: A REQUEST message of site Sjhas the form REQUEST(j, n) where n (n=1, 2,
...) is a sequence number which indicates that site Sjis requesting its nth CS execution.

A site Si keeps an array of integers RNi[1..N] where RNi[j] denotes the largest sequence number received in a REQUEST message so far from site Sj. When site Si receives a REQUEST(j, n)message, it sets RNi[j]:=max(RNi[j], n). Thus, when a site Si receives a REQUEST (j, n) message, the request is outdated if RNi[j]>n. Sites with outstanding requests for the CS are determined in the following manner: The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site Sj executed most recently. After executing its CS, a site Si updates LN[i]:=RNi[i] to indicate that its request corresponding to sequence number RNi[i] has been executed.Token array LN[1..N] permits a site to determine if a site has an outstanding request for the CS. Note that at site Si if RNi[j]=LN[j]+1, then site Sj is currently requesting token. After executing the CS, a site checks this condition for all the j' s to determine all the sites which are requesting the token and places their id's in queue Q if these id's are not already present in the Q. Finally, the site sends the token to the site whose id is at the head of the Q.

ALGORITHM

1. **Requesting the critical section**
   - **(a)** If requesting site Si does not have the token, then it increments its sequence number, RNi[i], and sends a REQUEST(i, sn) message to all other sites. ('sn' is the updated value of RNi[i].)
   - **(b)** When a site Sj receives this message, it sets RNj[i] to max(RNj[i], sn). If Sj has the idle token, then it sends the token to Si if RNj[i]=LN[i]+1.

2. **Executing the critical section**
   - **(c)** Site Si executes the CS after it has received the token.

3. **Releasing the critical section** Having finished the execution of the CS, site Si takes thefollowing actions:
   - **(d)** It sets LN[i] element of the token array equal to RNi[i].
   - **(e)** For every site Sj whose id is not in the token queue, it appends its id to the token queue if RNi[j]=LN[j]+1.

(f) If the token queue is nonempty after the above update, Si deletes the top site id from the token queue and sends the token to the site indicated by the id. Thus, after executing the CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS). Note that Suzuki-Kasami's algorithm is not symmetric because a site retains the token even if it does not have a request for the CS, which is contrary to the spirit of Ricart and Agrawala's definition of symmetric algorithm: *"no site possesses the right to access its CS when it has not been requested"*.

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 Si reach other sites in finite time. Since one of these sites will have token in finite time, site Si'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 Si will get the token and execute the CS in finite time. 2

Performance

Beauty of Suzuki-Kasami algorithm lies in its simplicity and efficiency. No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request. If a site does not hold the token when it makes a request, the algorithm requires N messages to obtain the token. Synchronization delay in this algorithm is 0 or T.

DEAD LOCK DETECTION

INTRODUCTION

A deadlock is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.

We can consider two types of deadlock:

- Communication deadlock occurs when process A is trying to send a message to process B, which is trying to send a message to process C which is trying to send a message to A.
- A resource deadlock occurs when processes are trying to get exclusive access to devices, files, locks, servers, or other resources. We will not differentiate between these types of deadlock since we can consider communication channels to be resources without loss of generality.

"A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set."

Deadlock deals with various components **like deadlock prevention, deadlock avoidance other than deadlock detection**.

- **Deadlock prevention** is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre-empting a process that hold the needed resource.
- In the **deadlock avoidance** approach to distributed system, a resource is granted to a process if the resulting global system is safe.
- **Deadlock detection** requires an examination of the status of the process-resources interaction for the presence of a deadlock condition. To resolve the deadlock, we have to abort a deadlocked process.

Necessary Conditions for Deadlock

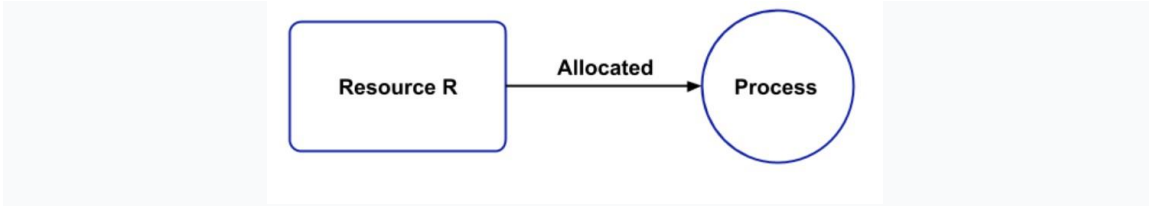
A **deadlock** is a situation in which more than one process is blocked because it is holding a resource and also requires some resource that is acquired by some other process. Therefore, none of the processes gets executed.

The four necessary conditions for a deadlock to arise are as follows.

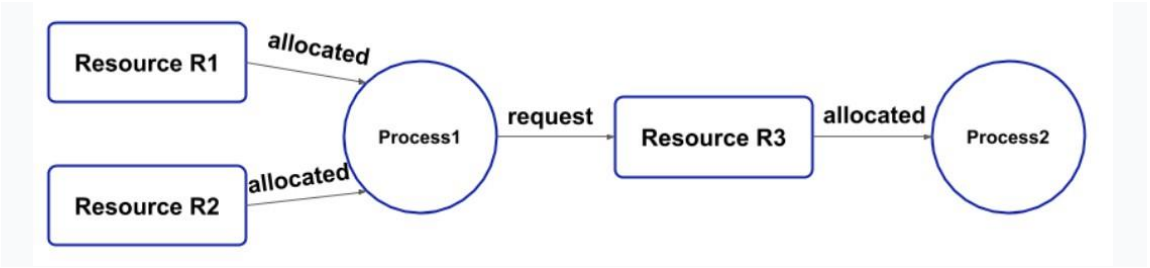
- **Mutual Exclusion**: Only one process can use a resource at any given time i.e. the resources are non-sharable.
- **Hold and wait**: A process is holding at least one resource at a time and is waiting to acquire other resources held by some other process.
- **No preemption**: The resource can be released by a process voluntarily i.e. after execution of the process.
  Circular Wait: A set of processes are waiting for each other in a circular fashion. For example, let's say there are a set of processes {P0, P1, P2, P3} such that P0 depends on P1, P1 depends onP2, P2 depends on P3 and P3 depends on P0. This creates a circular relation between all these processes and they have to wait forever to be executed.

Now let's see them one by one in detail.

- **Mutual Exclusion:** A resource can be held by only one process at a time. In other words, if a process P1 is using some resource R at a particular instant of time, then some other process P2 can't hold or use the same resource R at that particular instant of time. The process P2 can make a request for that resource R but it can't use that resource simultaneously with process P1.

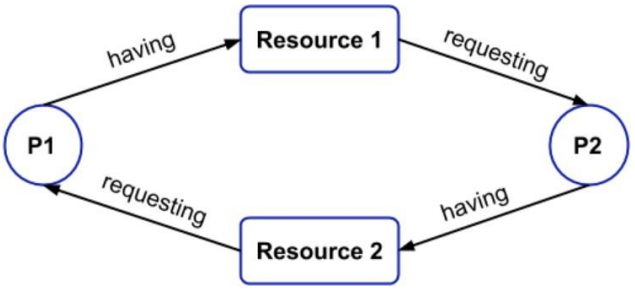


- **Hold and Wait:** A process can hold a number of resources at a time and at the same time, it can request for other resources that are being held by some other process. For example, a process P1 can hold two resources R1 and R2 and at the same time, it can request some resource R3 that is currently held by process P2.



- **No preemption:** A resource can't be preempted from the process by another process, forcefully. For example, if a process P1 is using some resource R, then some other process P2 can't forcefully take that resource. If it is so, then what's the need for various scheduling algorithm.

  The process P2 can request for the resource R and can wait for that resource to be freed by the process P1.
- **Circular Wait:** Circular wait is a condition when the first process is waiting for the resource held by the second process, the second process is waiting for the resource held by the third process, and so on. At last, the last process is waiting for the resource held by the first process. So, every process is waiting for each other to release the resource and no one is releasing their own resource. Everyone is waiting here for getting the resource. This is called a circular wait.



Deadlock will happen if all the above four conditions happen simultaneously.

SYSTEM MODEL

- A distributed system consists of a set of processors that are connected by a communication network.
- The communication delay is finite but unpredictable.
- A distributed program is composed of a set of n asynchronous processes p1, p2, . . . , pi, . . . , pn 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.
- 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.

- The system can be modelled as a directed graph in which vertices represent the processes and edge represent unidirectional communication channels.

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 (WFG)

- In distributed systems, the state of the system can be modelled by **directed graph, called a *wait for graph* (WFG)**.
- In a WFG, nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

Figure 10.1 shows a WFG, where process P11 of site 1 has an edge to process P21 of site 1 and P32 of site 2 is waiting for a resource which is currently held by process P21. At thesame time process P32 is waiting on process P33 to release a resource. If P21 is waiting on process P11, then processes P11, P32 and P21 form a cycle and all the four processes are involved in a deadlock depending upon the request model.

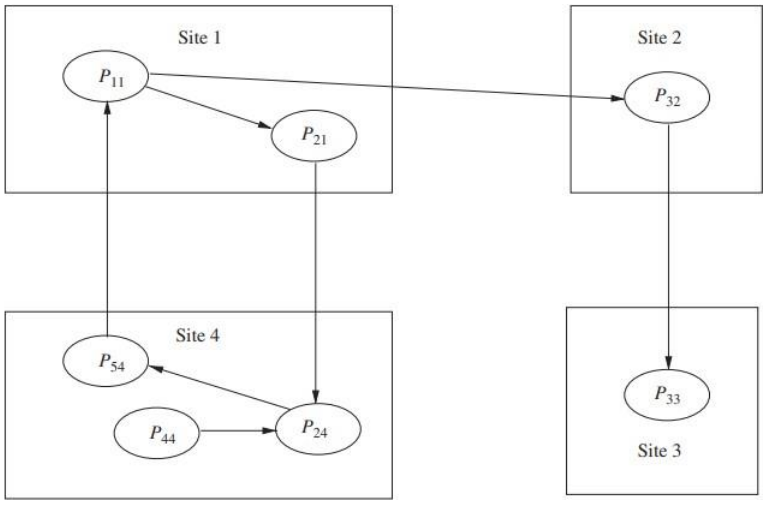


Figure 10.1 Example of a WFG.

PRELIMINARIES: DEADLOCK HANDLING STRATEGIES

There are three strategies for handling deadlocks,

- **Deadlock Prevention.**
- **Deadlock Avoidance.**
- **Deadlock Detection.**

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. Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by pre-empting a process which holds the needed resource. This approach is highly inefficient and impractical in distributed systems. In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system). However, due to several problems, deadlock avoidance is impractical in distributed systems.

Issues in Deadlock Detection

Deadlock handling using the approach of deadlock detection entails addressing two basic issues:

1. **Detection of existing deadlocks**
2. **Resolution of detected deadlocks.**

1. Detection of Deadlocks

- Detection of deadlocks involves addressing two issues: maintenance of the WFG and searching of the WFG for the presence of cycles (or knots). Since 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 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:

(i) Progress (No undetected deadlocks): The algorithm must detect all existing deadlocks in finite time. Once a deadlock has occurred, the deadlock detection activity should continuously progress until the deadlock is detected. 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.

(ii) Safety (No false deadlocks): The algorithm should not report deadlocks which do not exist (called *phantom or false* deadlocks). In distributed systems where there is no global memory and there is no global clock, it is difficult to design a correct deadlock detection algorithm because sites may obtain out of date and inconsistent WFG of the system. As a result, sites may detect a cycle which never existed but whose differentsegments existed in the system at different times. This is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

2. 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.

Note that several deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph. Therefore, when a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system. If this information is not cleaned in timely manner, it may result in detection of phantom deadlocks.

Untimely and inappropriate cleaning of broken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

MODELS OF DEADLOCKS

- Distributed systems allow many kinds of resource requests. A process might requirea single resource or a combination of resources for its execution. This section introduces a hierarchy of request models starting with very restricted forms to the ones with no restrictions whatsoever. This hierarchy shall be used to classify deadlock detection algorithms based on the complexity of the resource requests they permit.

The Single Resource Model

The single resource model is the simplest resource model in a distributed system, here a process can have at most one outstanding request for only one unit of a resource. Since 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. In a later section, an algorithm to detect deadlock in the single resource model is presented.

The AND Model

In the AND model, a process can request for more than one resource simultaneously andthe request 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. Consider the example WFG described in the Figure 10.1. Process P11 has two outstanding resource requests. In case of the AND model, P11shall become active from idle state only after both the resources are granted. There is a cycle P11->P21->P24->P54->P11 which corresponds to a deadlock situation.

In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked. Consider processP44 in Figure 10.1. It is not a part of any cycle but is still deadlocked as it is dependent onP24 which is deadlocked. Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

The OR Model

In the 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. The requested resources may exist at different locations. If all requests in

the WFG are OR requests, then the nodes are called OR nodes. Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.

To make it more clear, consider Figure 10.1. If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied. After P32 finishes execution and releases its resources, process P11 can continue with its processing.

In the OR model, the presence of a knot indicates a deadlock. In a WFG, a vertex v is in a knot if for all u:: u is reachable from v : v is reachable from u. No paths originating from a knot shall have dead ends.

A deadlock in the OR model can be intuitively defined as follows: A process Pi is blocked if ithas 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 *idle* to *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. Intuitively, a set of processes S is deadlocked if all the processes in S are permanently blocked. To formally state that a set of processes is deadlocked, the following conditions hold true:

1. Each of the process is the set S is blocked,
2. The dependent set for each process in S is a subset of S, and
3. No grant message is in transit between any two processes in set S.

We now show that a set of processes S shall remain permanently blocked in the OR model if the above conditions are met. A blocked process P is the set S becomes *active* only after receiving a grant message from a process in its dependent set, which is a subset of S. Note that no grant message can be expected from any process in S because they are all blocked. Also, the third condition states that no grant messages in transit between any two processesin set S. So, all the processes in set S are permanently blocked.

Hence, deadlock detection in the OR model is equivalent to finding knots in the graph. Note that, there can be a process deadlocked which is not a part of a knot. Consider the Figure 10.1

where P44 can be deadlocked even though it is not in a knot. So, in an OR model, a blocked process P is deadlocked if it is either in a knot or it can only reach processes on a knot.

The AND-OR Model

A generalization of the previous two models (OR model and AND model) is the AND- OR model. In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request. For example, in the AND-OR model, a request for multiple resources can be of the form x *and* (y *or* z). The requested resources may exist at different locations. To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG. Since a deadlock is a stable property (i.e., once it exists, it does not go away by itself), this property can be exploited and a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.

However, this is a very inefficient strategy. Efficient algorithms to detect deadlocks in AND-OR model are discussed in Herman

The $\binom{p}{q}$ Model

Another form of the AND-OR model is the $\binom{p}{q}$ model (called the P-out-of-Q model) which allows a request to obtain any k available resources from a pool of n resources. Both the models are the same in expressive power. However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request.

Every request in the $\binom{p}{q}$ model can be expressed in the AND-OR model and vice-versa. Note that AND requests for p resources can be stated as $\binom{p}{p}$ and OR requests for p resources can be stated as $\binom{p}{1}$.

Unrestricted Model

In the 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 way of looking at the deadlock problem helps in separation of concerns: Concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication). Hence, these algorithms can be used to detect other stable properties as they deal with this general model. But, these algorithms are of more theoretical value for distributed systems since no further assumptions are made about the underlying distributed systems computations which leads to a great deal of overhead.

KNAPP'S CLASSIFICATION OF DISTRIBUTED DEADLOCK DETECTION

Algorithms

- Distributed deadlock detection algorithms can be divided into four classes : path- pushing, edge-chasing, diffusion computation, and global state detection.

Path-Pushing Algorithms

In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG. The basic idea is to build a global WFG for each site of the distributed system. In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighboring sites. After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.

Edge-Chasing Algorithms

In an edge-chasing algorithm, the presence of a cycle in a distributed graph structureis be verified by propagating special messages called probes, along the edges of the graph. These probe messages are different than the request and reply messages. The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.

Whenever a process that is executing receives a probe message, it simply discards this message and continues. Only blocked processes propagate probe messages along their outgoing edges. An interesting variation of this method can be found in Mitchell [36], whereprobes are sent upon request and in the opposite direction of the edges.

Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short. Examples of such algorithms include Chandy et al., Choudhary et al., Kshemkalyani-Singhal , and Sinha-Natarajan algorithms.

Diffusing Computations Based Algorithms

In *diffusion computation* based distributed deadlock detection algorithms; deadlock detection computation is diffused through the WFG of the system. These algorithms make use of echo algorithms to detect deadlocks. This computation is superimposed on theunderlying distributed computation. If this computation terminates, the initiator declares a deadlock. The main feature of the superimposed computation is that the global WFG is implicitly reflected in the structure of the computation. The actual WFG is never builtexplicitly.

To detect a deadlock, a process sends out query messages along all the outgoing edges inthe WFG. These queries are successively propagated (i.e., diffused) through the edges of the WFG.

Queries are discarded by a running process and are echoed back by blocked processesin the following way: When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent (to its successors in the WFG). For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message. The initiatorof a deadlock detection detects a deadlock when it receives reply for every query it had sent out. Examples of these types of deadlock detection algorithms are Chandy-Misra-Haas algorithm for OR model and Chandy-Herman algorithm .

Global State Detection Based Algorithms

Global state detection based deadlock detection algorithms exploit the following facts:

(i) A consistent snapshot of a distributed system can be obtained without freezing the underlyingcomputation and (ii) a consistent snapshot may not represent the system state at any moment in time, but if a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock. Examples of these types of algorithms include Bracha- Toueg , Wang et al., and Kshemkalyani-Singhal algorithms.

MITCHELL AND MERRITT'S ALGORITHM FOR THE SINGLE-RESOURCE MODEL

Mitchell and Merritt's algorithm belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG. When a probe initiated by a process comes back to it, the process declares deadlock. The algorithm has many good features like:

1. Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock. This algorithm can be improvised by including priorities and the lowest priority process in a cycle detects deadlock and aborts.
2. In this algorithm process which is detected in deadlock is aborted spontaneously, even though under this assumption phantom deadlocks cannot be excluded. It can be shown, however, that only genuine deadlocks will be detected in the absence of spontaneous aborts.

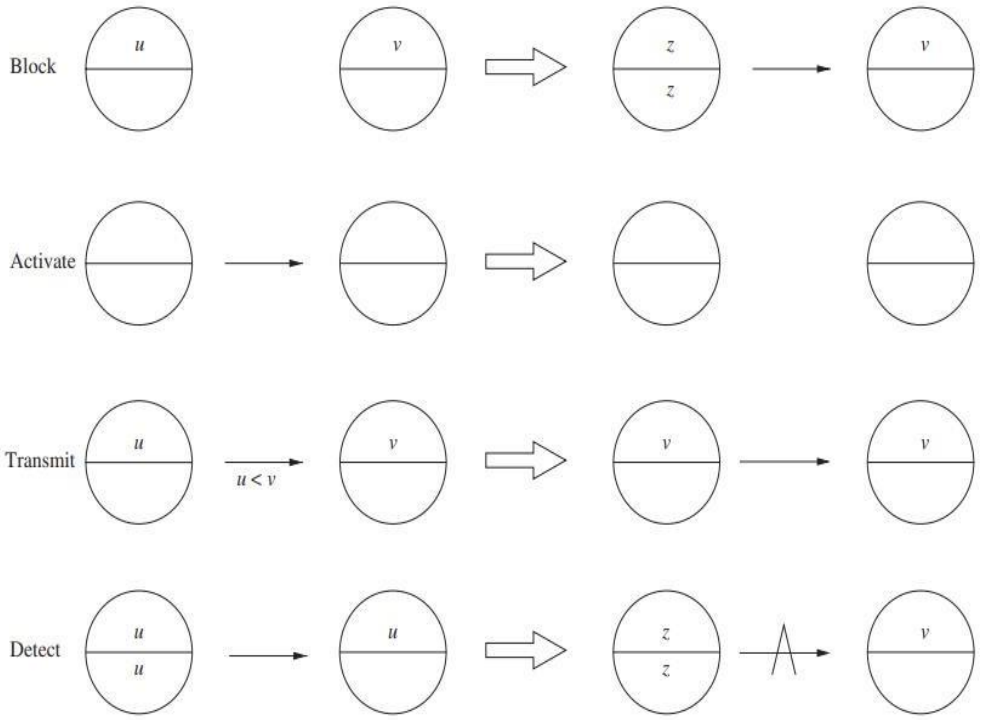


Figure 10.2 The four possible state transitions [22].

Each node of the WFG has two local variables, called labels: a private label, which is unique to the node at all times, though it is not constant, and a public label, which can be read byother processes and which may not be unique. Each process is represented as u/v where uand u are the public and private labels, respectively. Initially, private and public labels areequal for each process.

A global WFG is maintained and it defines the entire state of the system.The Algorithm is defined by the four state transitions shown in Figure 10.2, where z = inc(u, v), and inc(u, v) yields a unique label greater than both u and v labels that are not shown do not change.Block creates an edge in the WFG. Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for. Activate denotes that a process has acquired the resource from the process it was waiting for. Transmit

propagates larger labels in the opposite direction of the edges by sending a probe message. Whenever a process receives a probe which is less then its public label, then it simply ignores that probe. Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
Mitchell and Merritt showed that every deadlock is detected. Next, we show that in the absence of spontaneous aborts, only genuine deadlocks are detected. As there are no spontaneous aborts, we have following invariant:
For all processes u/v: u <= v

Proof. Initially u = u for all processes. The only requests that change u or v are

1) Block: u and v are set such that u = v.
(2) Transmit: u is increased. Hence, the invariant.

From the previous invariant, we have the following lemmas.

Lemma 12. *For any process u/v, if u > u, then u was set by a Transmit step.*

Theorem 13. *If a deadlock is detected, a cycle of blocked nodes exists.*

Proof. A deadlock is detected if the following edge p →p' exists:

We will prove the following claims:
(1) u has been propagated from p to p' via a sequence of Transmits.
(2) P has been continuously blocked since it transmitted u.
(3) All intermediate nodes in the transmit path of (l), including p', have been continuously blocked since they transmitted u.

From the above claims, the proof for the theorem follows as discussed below:

From the invariant and the uniqueness of private label u of p' : u < v. By Lemma 4.1, u was set by a Transmit step. From the semantics of Transmit, there is some p" with private label u and public label w. If w = u, then p" = p, and it is a success. Otherwise, if w <u, we repeat the argument. Since there are only processes, one of them is p. If p is active then it indicates that it has transmitted u else it is blocked if it detects deadlock. Hence upon blocking it incremented its private label. But then private and public labels cannot be equal. Consider a process which has been active since it transmitted u. Clearly, its predecessor is also active, as Transmits migrate in opposite direction. By repeating this argument, we can show p has been active since it transmitted u. The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted. This algorithm has two phases. The first phase is almost identical to the algorithm. In the second phase the smallest priority is propagated around the circle, the propagation stops when one process recognizes the propagated priority as its own.

Message Complexity

Now we calculate the complexity of the algorithm. If we assume that a deadlock persists long enough to be detected, the worst-case complexity of the algorithm is s (s - 1)/2 Transmit steps, where s is the number of processes in the cycle.

CHANDY-MISRA-HAAS ALGORITHM FOR THE AND MODEL

We now discuss Chandy-Misra-Haas's distributed deadlock detection algorithm for AND model that is based on edge-chasing. The 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 Pi and it is being sent by the home site of process Pj to the home site of process Pk. 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 Pj is said to be *dependent* on another process Pk if there exists a sequence of processes Pj,Pi1, Pi2, ..., Pim, Pk such that each process except Pk in the sequence is blocked and each process, except the Pj, holds a resource for which the previous process in the sequence is waiting. Process Pj is said to be *locally dependent* upon process Pk if Pj is dependent upon Pk and both the processes are on the same site.

Data Structures

Each process Pi maintains a boolean array, dependent i, where dependent i(j) is true only if Pi knows that Pj 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.

Therefore, a probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.

Performance Analysis

In the algorithm, one probe message (per deadlock detection initiation) is sent on every edge of the WFG which that two sites. Thus, the algorithm exchanges at most $m(n - 1)/2$ messages to detect a deadlock that involves m processes and that spans over n sites. The size of messages is fixed and is very small (only 3 integer words). Delay in detecting a deadlock is O(n).

CHANDY-MISRA-HAAS ALGORITHM FOR THE OR MODEL

We now discuss Chandy-Misra-Haas distributed deadlock detection algorithm for OR model that is based on the approach of diffusion-computation. 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) and reply(i, j, k), denoting that they belong to a diffusion computation initiated by a process Pi and are being sent from process Pj to process Pk.

Basic Idea

A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set (i.e., processes from which it is waiting to receive a message). If an active process receives a query or reply message, it discards it. When a blocked process Pk receivesa query(i, j,k) message, it takes the following actions:

1. If this is the first query message received by Pk for the deadlock detection initiated by Pi (called the *engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable numk(i) to the number of query messages sent.

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.

Algorithm 10.2 Chandy–Misra–Haas algorithm for the OR model [6].

2. If this is not the engaging query, then Pk returns a reply message to it immediately provided Pk has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query. Process Pk maintains a boolean variable waitk(i) that denotes the fact that it has been continuously blocked since it received the last engaging query from process Pi. When a blocked process Pk receives a reply(i, j, k) message, it decrements numk(i) only if waitk(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 had sent out for this engaging query. The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.

For ease of presentation, we assumed that only one diffusion computation is initiated for a process. In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process gets blocked), but, at any time only one diffusion computation is current for any process. However, messages for outdated diffusion computations may still be in transit. The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

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; Check pointing and Rollback Recovery: Introduction – Background and Definitions – Issues in Failure Recovery – Checkpoint-based Recovery – Coordinated Check pointing Algorithm - - Algorithm for Asynchronous Check pointing and Recovery

CONSENSUS AND AGREEMENT ALGORITHMS

Introduction

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.

Assumptions

- **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.

→ In the fail-stop model, a process may crash in the middle of the step.

→ In the Byzantine failure model, it may send a message to only a subset of the destination set before crashing.

→ The choice of the failure model determines the feasibility and complexity of solving consensus.

- **Synchronous/ Asynchronous**
- **communication**
- **Network connectivity**
- **Sender identification**
- **Channel reliability**

- **Authenticated vs. non-authenticated messages**
- **Agreement variable**

Problem Specifications

Failure mode	Synchronous system (message-passing and shared memory)	Asynchronous system (message-passing and shared memory)
No failure	agreement attainable; common knowledge also attainable	agreement attainable; concurrent common knowledge attainable
Crash failure	agreement attainable $f < n$ processes $\Omega(f+1)$ rounds	agreement not attainable
Byzantine failure	agreement attainable $f \leq \lfloor (n-1)/3 \rfloor$ Byzantine processes $\Omega(f+1)$ rounds	agreement not attainable

Table: Overview of results on agreement. f denotes number of failure-prone processes. n is the total number of processes.

- **Byzantine Agreement** (single source has an initial value)
  Agreement: All non-faulty processes must agree on the same value.

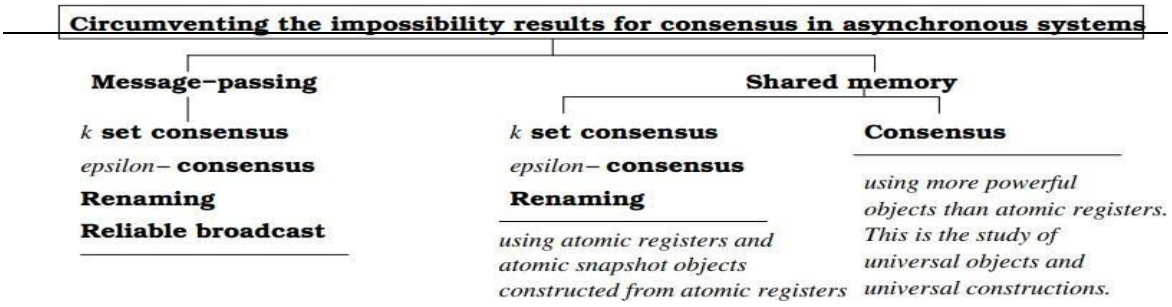
  Validity: If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.
  Termination: Each non-faulty process must eventually decide on a value.
- **Consensus Problem** (all processes have an initial value)
  Agreement: 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.
- **Interactive Consistency** (all processes have an initial value)
  Agreement: All non-faulty processes must agree on the same array of values A[v1 . . . vn].
  Validity: If process i is non-faulty and its initial value is vi, then all non- faulty processes agree on vi as the ith 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

- In a failure-free system, consensus can be attained in a straight forward manner.

Solvable Variants	Failure model and overhead	Definition
Reliable broadcast	crash failures, $n > f$ (MP)	Validity, Agreement, Integrity conditions
k-set consensus	crash failures. $f < k < n$. (MP and SM)	size of the set of values agreed upon must be less than k
ϵ-agreement	crash failures $n \geq 5f + 1$ (MP)	values agreed upon are within ϵ of each other
Renaming	up to f fail-stop processes, $n \geq 2f + 1$ (MP) Crash failures $f \leq n - 1$ (SM)	select a unique name from a set of names

Table: Some solvable variants of the agreement problem in asynchronous system. The overhead bounds are for the given algorithms, and not necessarily tight bounds for the problem.



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 application specific function.
- 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 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)

- Up to f (< n) crash failures possible.
- In f + 1 rounds, at least one round has no failures.
- Now justify: agreement, validity, termination conditions are satisfied.
- Complexity: $O(f + 1)n^2$ messages
- f + 1 is lower bound on number of rounds

Algorithm

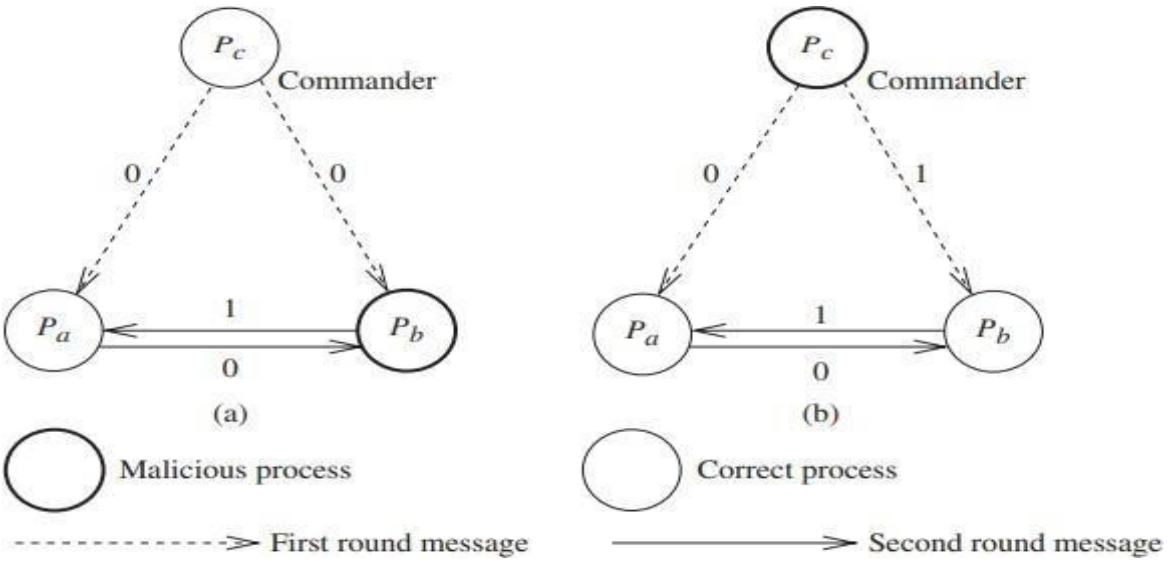
(global constants)
integer: f ; // maximum number of crash failures tolerated
(local variables)
integer: x ←− local value;
(1) Process Pi (1 ≤ i ≤ n) executes the Consensus algorithm for up to f crash failures:
(1a) for round from 1 to f + 1 do
(1b) if the current value of x has not been broadcast then
(1c) broadcast(x);
(1d) yj ←− value (if any) received from process j in this round;
(1e) x ←− min(x, yj);
(1f) output x as the consensus value.

- The **agreement condition** is satisfied because in the f +1 rounds, there must be at least one round in which no process failed.
- 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.

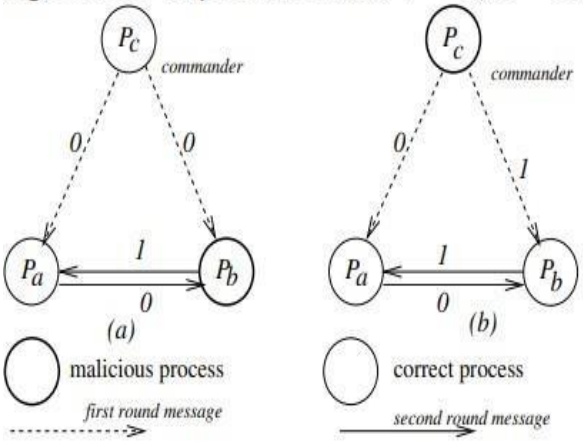
CONSENSUS ALGORITHMS FOR BYZANTINE FAILURES (SYNCHRONOUS SYSTEM)

Upper bound on Byzantine processes

- In a system of **n** processes, the Byzantine agreement problem can be solved in a synchronous system only if the number of Byzantine processes **f** is such that **f ≤ (n−1/3).**

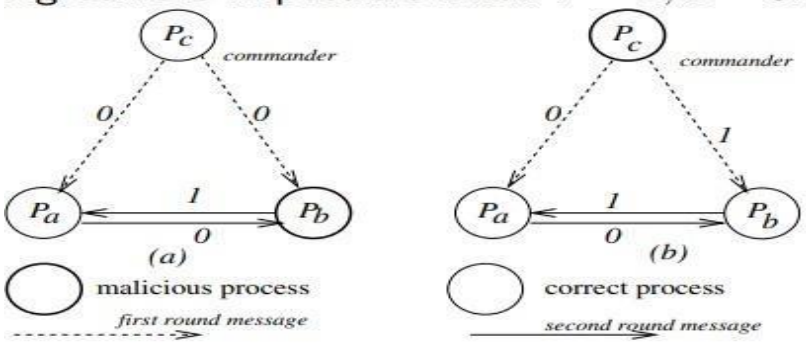


Agreement impossible when $f = 1, n = 3$.

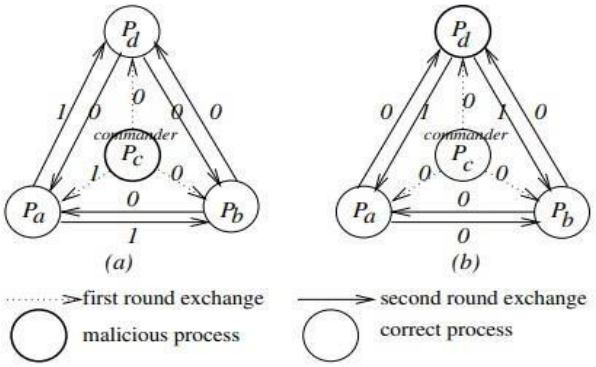


- Taking simple majority decision does not help because loyal commander P_a cannot distinguish between the possible scenarios (a) and (b);
- hence does not know which action to take.
- Proof using induction that problem solvable if $f \leq \lfloor \frac{n-1}{3} \rfloor$. See text.

Agreement impossible when $f = 1, n = 3$.



Consensus Solvable when $f = 1, n = 4$



- There is no ambiguity at any loyal commander, when taking majority decision
- Majority decision is over 2nd round messages, and 1st round message received directly from commander-in-chief process.

BYZANTINE AGREEMENT TREE ALGORITHM: EXPONENTIAL (SYNCHRONOUS SYSTEM)

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
(message type)
$OM(v, Dests, List, faulty)$, where
v is a boolean,
$Dests$ is a set of destination process i.d.s to which the message is sent,
$List$ is a list of process i.d.s 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 j, 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
$OM(v_j, Dests - \{i\}, concat(\langle i \rangle, L), (f'-1))$
to destinations not in $concat(\langle i \rangle, L)$
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 $majority_{k \notin List, k \neq i}(v_j, v_k)$ 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.

BZANTINE GENERALS (ITERATIVE FORMULATION), SYNC, MSG-PASSING

- In the recursive version of the algorithm, each message has the following parameters:
  - → a consensus estimate value (**v**);
  - → a set of destinations **(Dests)**;

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
tree of boolean:

- level 0 root is v^L_{init}, where $L = \langle\rangle$;
- level h $(f \geq h > 0)$ nodes: for each v^L_j at level $h - l = sizeof(L)$, its $n-2-sizeof(L)$ descendants at level h are $v_k^{concat(\langle j \rangle, L)}$, $\forall k$ 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 the 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 the oral message (OM):
(2a) **for** $rnd = 0$ **to** f **do**
(2b) **for** each message OM that arrives in this round, **do**
(2c) **receive** $OM(v, Dests, L = \langle P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty)$ from P_{k_1};
// $faulty + rnd = f$; $|Dests| + sizeof(L) = n$
(2d) $v^{tail(L)}_{head(L)} \longleftarrow v$; // $sizeof(L) + faulty = f + 1$. fill in estimate.
(2e) **send** $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty - 1)$
to $Dests - \{i\}$ **if** $rnd < f$;
(2f) **for** $level = f - 1$ **down to** 0 **do**
(2g) **for** each of the $1 \cdot (n-2) \cdot \ldots (n - (level+1))$ nodes v^L_x in level $level$, **do**
(2h) $v^L_x(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v^L_x, v_y^{concat(\langle x \rangle, L)})$;

→ a list of nodes traversed by the message, from most recent to least recent **(List);** and

→ The number of Byzantine processes that the algorithm still needs to tolerate **(faulty).**

→ The list L = <Pi ,Pk1,……., Pkf+1−faulty > represents the sequence of processes (subscripts) in the knowledge expression Ki(Kk1(Kk2……Kkf+1−faulty (v0)…..))

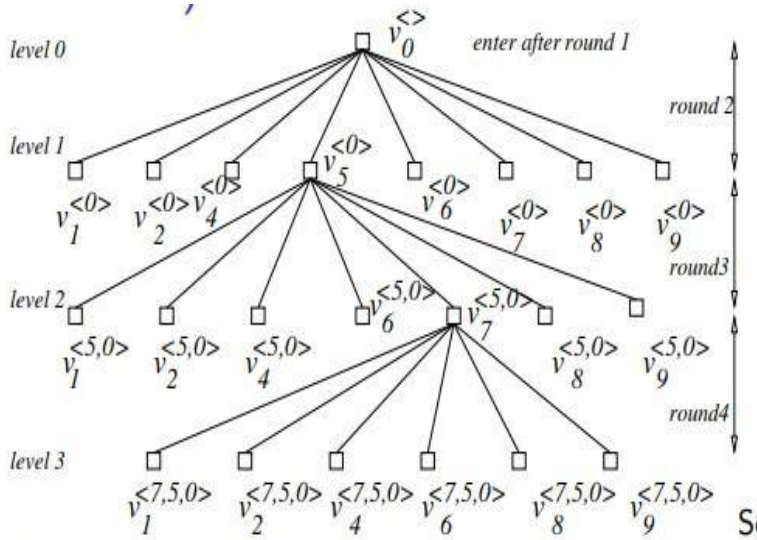
→ The commander invokes the algorithm with parameter faulty set to f, the maximum number of malicious processes to be tolerated.

→ The algorithm uses f +1 synchronous rounds.

→ Each message (having this parameter faulty = k) received by a process invokes several other instances of the algorithm with parameter faulty = k − 1.

→ The terminating case of the recursion is when the parameter faulty is 0.

TREE DATA STRUCTURE FOR AGREEMENT PROBLEM (BYZANTINE GENERALS)



Some branches of the tree at P_3. In this example, $n = 10, f = 3$, commander is P_0.

- (round 1) P_0 sends its value to all other processes using *Oral_Msg*(3), including to P_3.
- (round 2) P_3 sends 8 messages to others (excl. P_0 and P_3) using *Oral_Msg*(2). P_3 also receives 8 messages.
- (round 3) P_3 sends $8 \times 7 = 56$ messages to all others using *Oral_Msg*(1); P_3 also receives 56 messages.
- (round 4) P_3 sends $56 \times 6 = 336$ messages to all others using *Oral_Msg*(0); P_3 also receives 336 messages. The received values are used as estimates of the majority function at this level of recursion.

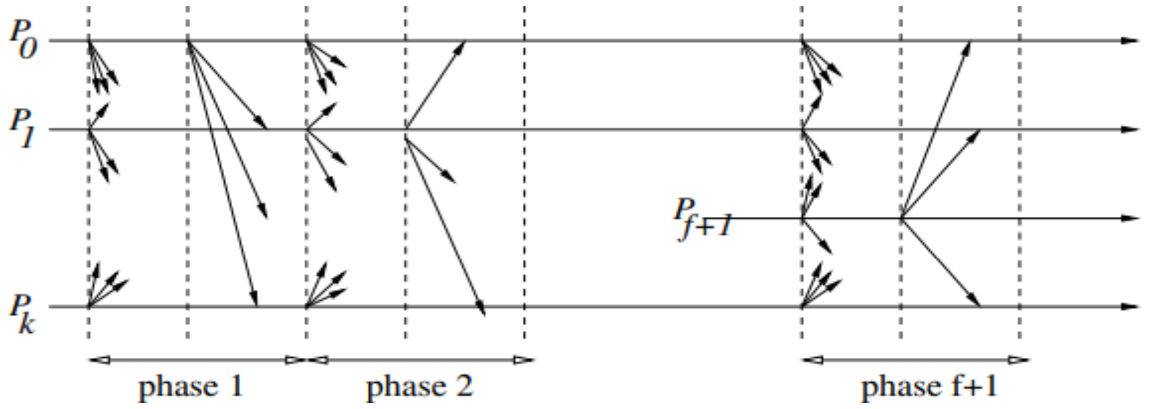
EXPONENTIAL ALGORITHM: AN EXAMPLE

Consider the tree at a lieutenant node P3, for n = 10 processes P0 through P9 and f = 3 processes. The commander is P0. Only one branch of the tree is shown for simplicity.

PHASE-KING ALGORITHM FOR CONSENSUS: POLYNOMIAL (SYNCHRONOUS SYSTEM)

- The phase-king algorithm proposed by Berman and Garay solves the consensus problem under the same model, requiring f +1 phases, and a polynomial number of messages but can tolerate only f < [n/4], malicious processes.
- The algorithm is so called because it operates in f +1 phases, each with two rounds, and a unique process plays an asymmetrical role as a leader in each round.
- The phase-king algorithm assumes a binary decision variable.

- Each phase has a unique"phase king" derived, say, from PID.
- Each phase has two rounds:
  - → in 1st round, each process sends its estimate to all other processes.
  - → 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.



The Phase King Algorithm: Code

```
(variables)
boolean: v ⟵ initial value;
integer: f ⟵ maximum number of malicious processes, f < ⌈n/4⌉;

(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 ⟵ the value among the v_j that occurs > n/2 times (default if no maj.);
(1f)       mult ⟵ 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 ⟵ majority;
(1m)       else v ⟵ 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 < [n/4]** malicious processes.

RECOVERY & CONSENSUS

CHECKPOINTING AND ROLLBACK RECOVERY-INTRODUCTION

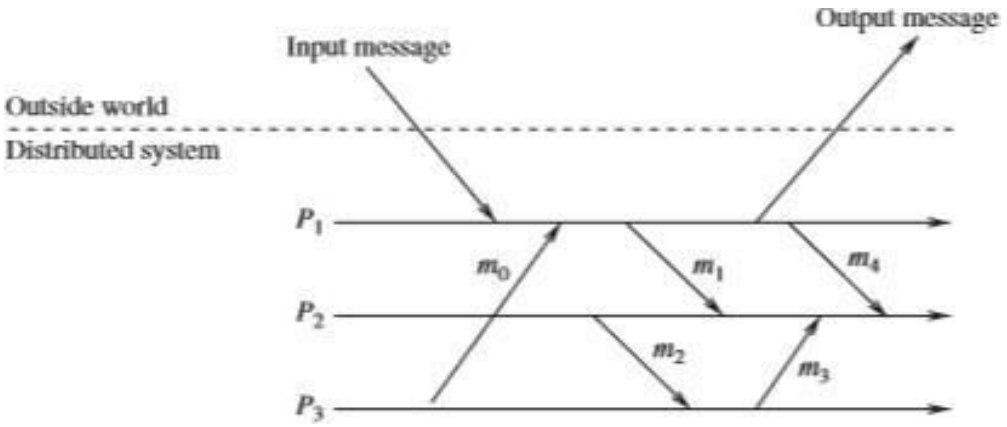
- Distributed systems are not fault-tolerant.
- To add reliability and high availability to distributed systems,many techniques have been developed
- The techniques include:
  → **Transactions,**
  → **Group communication,** and
  → **Rollback recovery.**
- **Rollback recovery** treats a distributed system application as a collection of processes that communicate over a network.
- Distributed system application achieves **fault tolerance** by periodically saving the state of a process during the failure-free execution, enabling it to restart from a saved state upon a failure to reduce the amount of lost work. The saved state is called a **checkpoint**.
- The procedure of restarting from a previously checkpointed state is called **rollback recovery**.
- Rollback recovery is complicated because messages induce inter-process dependencies during failure-free operation.
- Upon failure of one or more processes in a system, inter-process dependencies may force some of the processes that did not fail to roll back. It is called **rollback propagation**.
- **Rollback propagation** occurs :
  → The sender of a message **m** rolls back to a state that precedes the sending of m.
  → The receiver of **m** must also roll back to a state that precedes m's receipt;
  otherwise,
  → The states of the two processes would be inconsistent.
- This phenomenon of cascaded rollback is called the **domino effect**.
- **Independent or Uncoordinated check pointing**
  → In a distributed system, if each participating process takes its checkpoints independently, then the system is susceptible to the domino effect. This approach is called independent or uncoordinated checkpointing.

- **Coordinated check pointing**
  → In coordinated checkpointing processes coordinate their checkpoints to form a system-wide consistent state.
- **Communication-induced check pointing**
  → Communication-induced checkpointing forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes.
- The two approaches used in **rollback recovery** are:
  → **Checkpoint-based rollback recovery**
  → **Log-based rollback recovery**
- **Checkpoint-based rollback recovery** relies only on checkpoints to achieve fault-tolerance.
- **Log-based rollback recovery** combines checkpointing with logging of nondeterministic events.
  → **Log-based rollback recovery** relies on the **piecewise deterministic (PWD)** assumption, which postulates that all non- deterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's determinant.

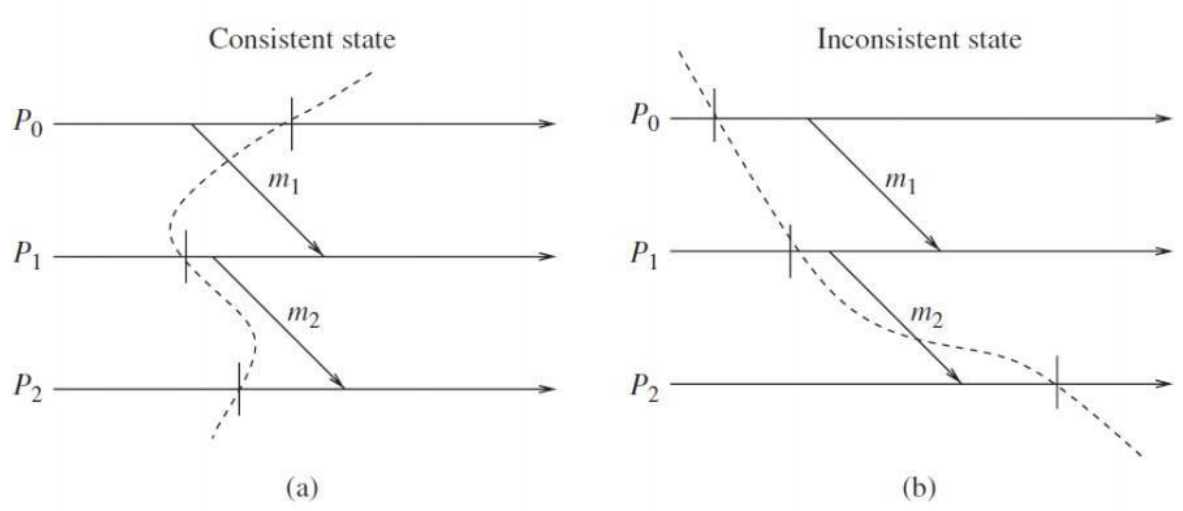
BACKGROUND AND DEFINITIONS

System Model

- A distributed system consists of a fixed number of processes, **P1, P2 …PN,**
  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.
- **Rollback-recovery protocols** enhance the reliability of the inter-process communication.
- Some protocols assume that the communication subsystem delivers messages reliably, in **first-in-first-out (FIFO) order**, while other protocols assume that the communication subsystem can **lose, duplicate**, or **reorder messages**.



- **Local checkpoint**
  - All processes save their local states at certain instants of time. This saved state is known as a **local checkpoint**.
  - A **local checkpoint** is a snapshot of the state of the process at a given instance.
  - The event of recording the state of a process is called **local checkpointing**.
  - **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.
  - **Ci,k** -- **k^{th}** local checkpoint at process Pi.
  - **Ci,0** – A process Pi takes a checkpoint **Ci,0** before it starts execution.
  - A local checkpoint is shown in the process-line by the symbol "|".
- **Consistent states**
  - **A global state of a distributed system**
    → 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**
    → a set of local checkpoints, one from each process.
  - **A consistent global checkpoint**
    → 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.



- **Example:**

In the **consistent state**,

- Message **m1** to have been sent but not yet received. The state is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event.

In the **inconsistent state**,

- Process **P2** is shown to have received **m2** but the state of process **P1** does not reflect having sent it.

- **Interactions with the outside world**
- A distributed application often interacts with the outside world to receive input data or deliver the outcome of a computation.
- A special process that interacts with the rest of the system through message passing called **"outside world process" (OWP).**
- The outside world should also to implement a consistent behavior of the system despite failures.
- **Approach** : **(Output Commit Problem)**

  → 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. It is called the **output commit problem**.
- 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**
- **In-transit message**

→ Messages that have been sent but not yet received.

→ The global state **{C1,8 ,C2,9, C3,8, C4,8}** shows that message **m1** has been sent but not yet received.

→ Message **m2** is also an in-transit message.

- **Lost messages**

→ Messages whose 'send' is done but 'receive' is undone due to rollback.

→ 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.

→ Message **m1** is a lost message.

- **Delayed messages**

→ Messages whose 'receive' is not recorded because the receiving process was either down or the message arrived after rollback.

→ Messages **m2** and **m5** are delayed messages.

- **Orphan messages**

→ Messages with 'receive' recorded but message 'send' not recorded.

→ Do not arise, if processes roll back to a consistent global state.

→ Orphan messages do not arise if processes roll back to a consistent global state.

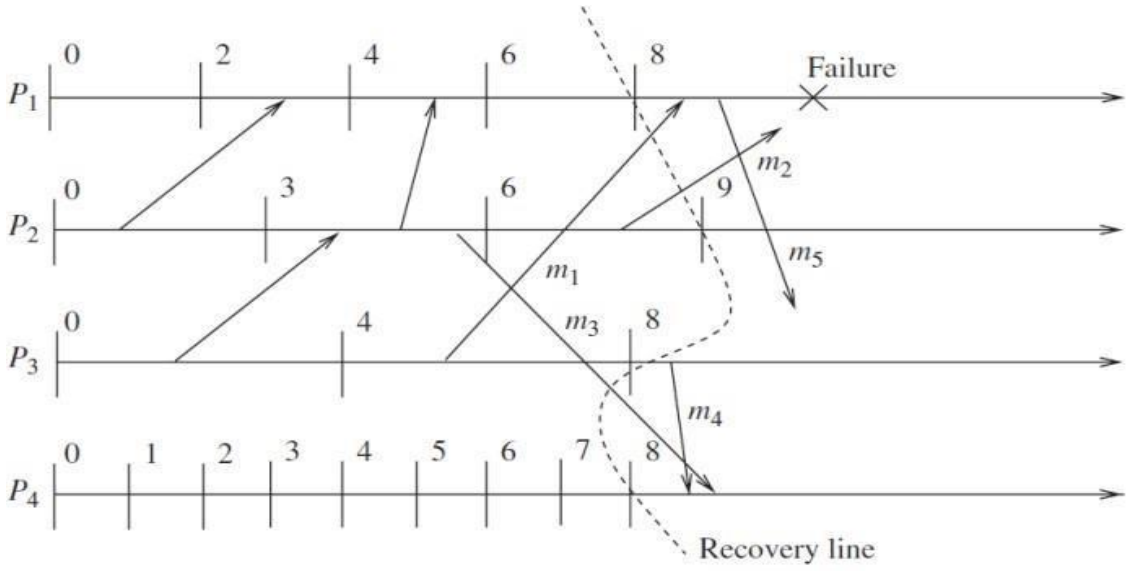
- **Duplicate messages**

→ Arise due to message logging and replaying during process recovery.

→ Message **m4** was sent and received before the rollback.

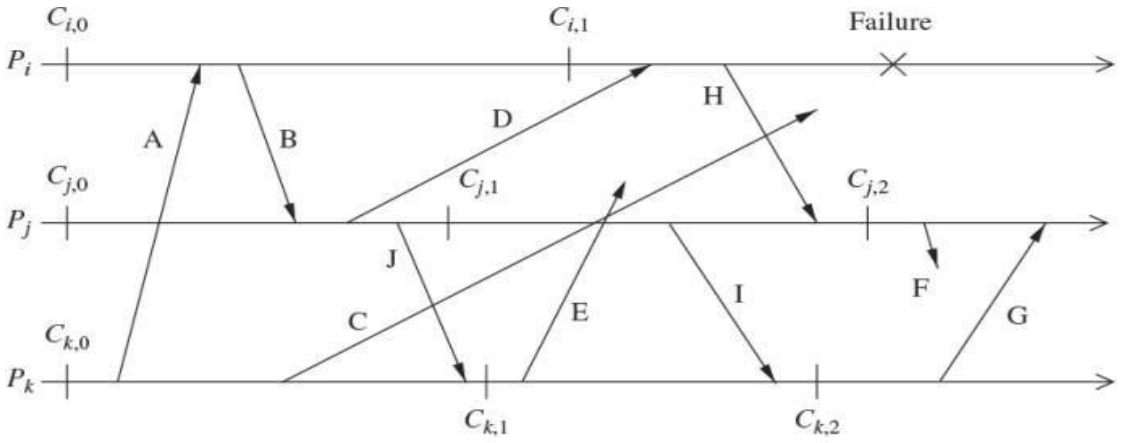
→ Due to the rollback of process **P4** to **C4,8** and process **P3** to **C3,8**, both send and receipt of message m4 are undone.

→ 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

Checkpoints : {Ci,0, C1,1}, **{Cj,0 ,Cj,1, Cj,2}**, and **{Ck,0 ,Ck,1, Ck,2} Messages : A - J**
The restored global consistent state : { Ci,1 , Cj,1, Ck,1 }



- Suppose process **Pi** fails at the instance, process **Pi's** state is restored to a valid state by rolling it back to its most recent checkpoint **Ci,1**.
- The rollback of process **Pi** to checkpoint **Ci,1** created an orphan message **H**.
- **Orphan message I** is created due to the rollback of process **Pj** to checkpoint **Cj,1**
- Messages **C, D, E**, and **F** are problematic :
  → Message **C**: **a delayed message**

→ Message **D**: **a lost message** since the send event for **D** is recorded in the restored state for **Pj**, but the receive event has been undone at process **Pi**.

→ **Lost messages** can be handled by having processes keep a message log of all the sent messages

→ Messages **E, F**: delayed orphan messages. After resuming execution from their checkpoints, processes will generate both of these messages.

CHECKPOINT-BASED RECOVERY

- In checkpoint-based recovery, the state of each process and the communication channel is checkpointed frequently so that, upon a failure, the system can be restored to a globally consistent set of checkpoints.
- It does not rely on the PWD assumption, and so does not need to detect, log, or replay non-deterministic events.
- Does not guarantee that prefailure execution can be deterministically regenerated after a rollback.
- Checkpoint-based rollback-recovery techniques can be classified into three types:
  - → **Uncoordinated check pointing,**
  - → **Coordinated check pointing,**
  - → **Communication-induced**

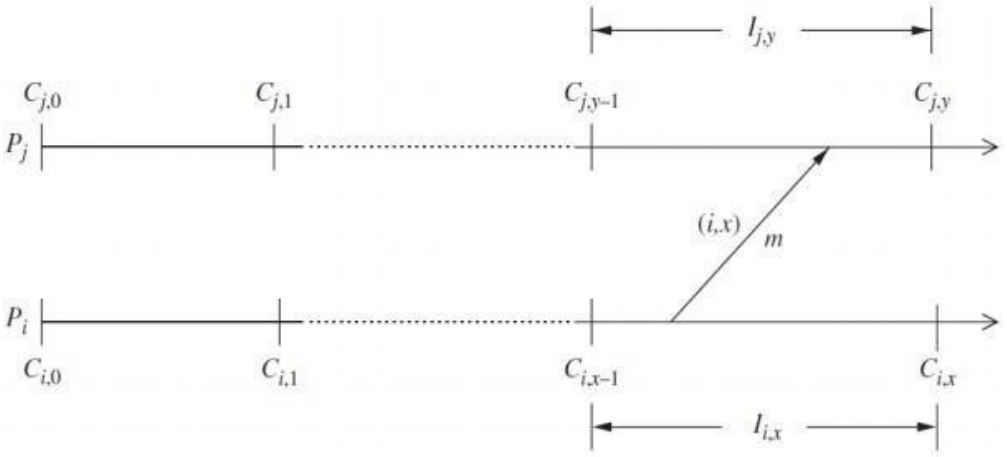
checkpointing **Uncoordinated check pointing**

- Each process has autonomy in deciding when to take checkpoints.
- Eliminates the synchronization overhead as there is no need for coordination between processes.
- **Advantages**
  - → The lower runtime overhead during normal execution.
- **Disadvantages**
  - → Domino effect during a recovery.
  - → Recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints.
  - → Each process maintains multiple checkpoints and periodically invoke a garbage collection algorithm.
  - → Not suitable for application with frequent output commits.
- The processes **record the dependencies among their checkpoints** caused by message exchange during failure-free operation.
- **Direct dependency tracking technique**
  - → Assume each process **Pi** starts its execution with an initial

checkpoint **Ci,0**

→ **Ii,x** : checkpoint interval,
Interval between **Ci,x-1** and **Ci,x**

→ When **Pj** receives a message **m** during **Ij,y**, it records the dependency from **Ii,x** to **Ij,y**, which is later saved onto stable storage when **Pj** takes **Cj,y**



- **Steps:**
  - → 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.
  - → The initiator then calculates the recovery line based on the global dependency information.
  - → Then the initiator 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.

- **Coordinated checkpointing**
- In **coordinated checkpointing**, processes orchestrate their checkpointing activities so that all local checkpoints form a consistent global state.
- Simplifies recovery.
- Not susceptible to the domino effect.

- **Disadvantages :**
  → Large latency is involved in committing output,
  → Delays and overhead are involved every time a new global checkpoint is taken.
- The **techniques used for coordinated checkpointing** are :
  1. Blocking coordinated checkpointing.
  2. Non-blocking checkpoint coordination.
  3. min-process non-blocking checkpointing

1. Blocking coordinated checkpointing

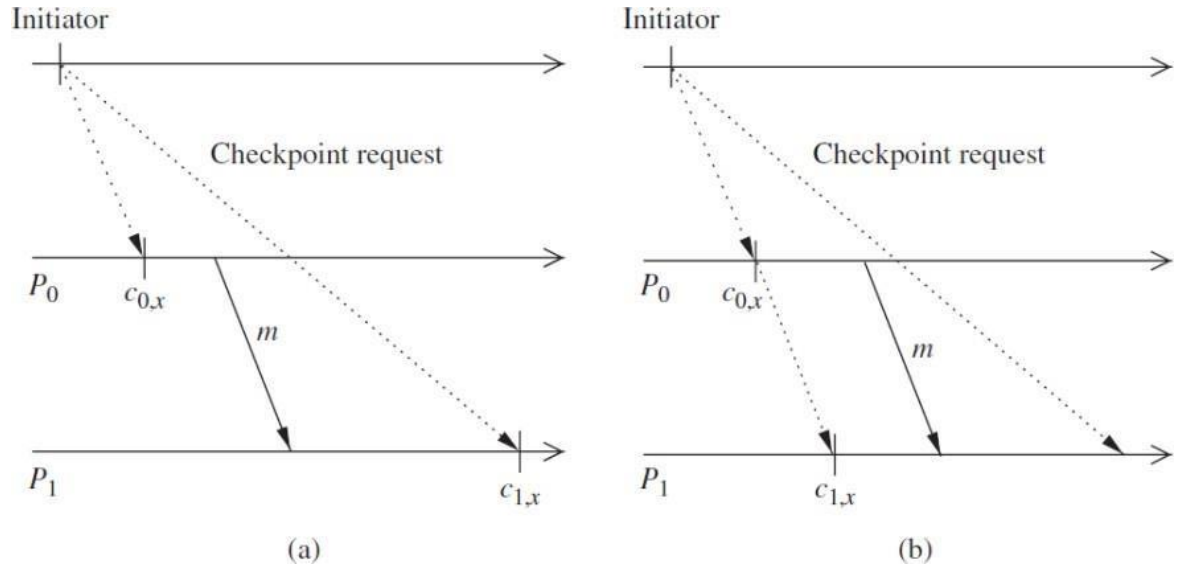
- **Block** communications while the checkpointing protocol executes.
- After a process takes a local checkpoint, it remains **blocked** until the entire checkpointing activity is complete.
- The coordinator takes a checkpoint and broadcasts a **request message** to all processes, asking them to take a checkpoint.
- When a process receives this message, it stops its execution, flushes all the communication channels, takes a tentative checkpoint, and sends an acknowledgment message back to the coordinator.
- When the **coordinator receives acknowledgments** from all processes, it broadcasts a **commit message**.
- After receiving the commit message, a process removes the old permanent checkpoint and atomically makes the tentative checkpoint permanent and then resumes its execution and exchange of messages with other processes.

2. Non-blocking checkpoint coordination

- 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
- $C\mathbf{1}$, shows the receipt of message **m** from $P\mathbf{0}$, while checkpoint $C\mathbf{0},x$ does not show **m** being sent from $P\mathbf{0}$.

- **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.



3. 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.
- The algorithm consists of two phases:
- **Phase 1 :**
  - 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.

- **Phase 2 :**
  - All processes identified in the first phase take a checkpoint.
  - The result is a consistent checkpoint that involves only the participating processes.
- After a process takes a checkpoint, it cannot send any message until the second phase terminates successfully. It is known as **z-dependency**.
- **Definition : z-dependency**
  - If a process **Pp** sends a message to process **Pq** during its **ith** checkpoint interval and process **Pq** receives the message during its **jth** checkpoint interval, then P**q Z-depends** on **Pp** during **Pp**'s **ith** checkpoint interval and **Pq's jth** checkpoint interval, denoted by **Pp→i Pq.**
    j

Communication –induced checkpointing

- Avoids the domino effect.
- Eliminates the useless checkpoints.
- In communication-induced checkpointing, processes take two types of checkpoints, namely,
  → **Autonomous and**
  → **Forced checkpoints.**
- The checkpoints that a process takes independently are called **local checkpoints**.
- A process is forced to take checkpoints are called **forced checkpoints**.
- Communication-induced checkpointing 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.
- Two types of communication-induced checkpointing :
  1. **Model-based checkpointing**
  2. **Index-based checkpointing**.
- In **model-based checkpointing**, the system maintains checkpoints and communication structures that prevent the domino effect.
- In **index-based checkpointing**, the system uses an indexing scheme for the local and forced checkpoints, such that the checkpoints of the same index at all processes form a consistent state.

KOO-TOUEG COORDINATED CHECKPOINTING ALGORITHM

- A **coordinated checkpointing** and recovery technique takes a consistent set of checkpointing and avoids domino effect and livelock problems during the recovery.
- Includes two parts:
  → **The checkpointing algorithm**
  → **The recovery algorithm**
- **Checkpointing Algorithm**
  → **Assumptions:**
    - FIFO channel,
    - End-to-end protocols,
    - Communication failures do not partition the network,
    - Single process initiation,
    - No process fails during the execution of the algorithm.
- Two kinds of checkpoints:
    - **Permanent checkpoint**
    - **Tentative checkpoint**

  → **Permanent checkpoint:** local checkpoint, part of a consistent global checkpoint

  → **Tentative checkpoint:** temporary checkpoint, become permanent checkpoint when the algorithm terminates successfully.
- The algorithm consist of two phases :
  → **First Phase :**
    - An initiating process **P_i** takes a tentative checkpoint

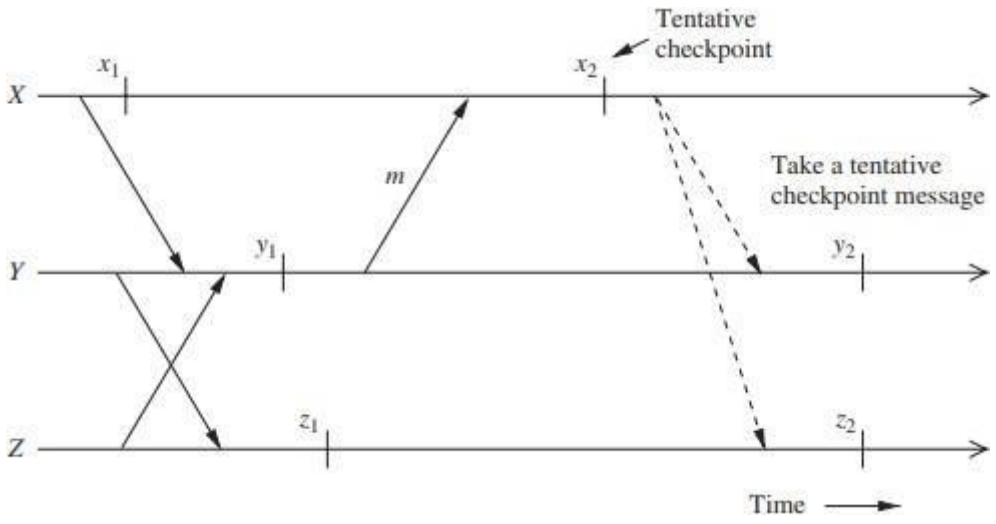
and requests all other processes to take tentative checkpoints.

- Each process informs **Pi** whether it succeeded in taking a tentative checkpoint.
- A process says "no" to a request if it fails to take a tentative checkpoint.
- If **Pi** learns that all the processes have successfully taken tentative checkpoints, **Pi** decides that all tentative checkpoints should be made permanent; otherwise, **Pi** decides that all the tentative checkpoints should be discarded.

→ **Second Phase**

- **Pi** informs all the processes of the decision it reached at the end of the first phase.
- A process, on receiving the message from **Pi,** will act accordingly.

- **Correctness:**
  - Either all or none of the processes take permanent checkpoint.
  - No process sends message after taking permanent checkpoint
- **Optimization**:
  - May be not all of the processes need to take checkpoints.
- **Example :**



- The set **{ x1, y1, z1}** is a consistent set of checkpoints.
- If process **X** decides to initiate the checkpointing algorithm after receiving message **m**. It takes a tentative checkpoint **x2** and sends

"**take tentative checkpoint**" messages to processes **Y** and **Z**, causing **Y** and **Z** to take checkpoints **y2** and **z2**, respectively.

- { **x1, y2, z2}** forms a consistent set of checkpoints.
- { **x1, y2, z1}** also forms a consistent set of checkpoints.
- There is no need for process Z to take checkpoint **z2** because Z has not sent any message since its last checkpoint.

- **Rollback Recovery Algorithm**

- The rollback recovery algorithm restore the system state to a consistent state after a failure with assumptions: single initiator, checkpoint and rollback recovery algorithms are not invoked concurrently.
- The algorithm consist of two phases :
  - → **First Phase :**
    - The initiating process sends a message to all other processes and asks for the preferences – restarting to the previous checkpoints.
    - All need to agree about either do or not.
  - → **Second Phase:**
    - The initiating process send the final decision to all processes, all the processes act accordingly after receiving the final decision.
- **Correctness:**
  - Resume from a consistent state.
- **Optimization**:
  - May not to recover all, since some of the processes did not change anything.

JUANG-VENKATESAN ALGORITHM FOR ASYNCHRONOUS CHECKPOINTING AND RECOVERY

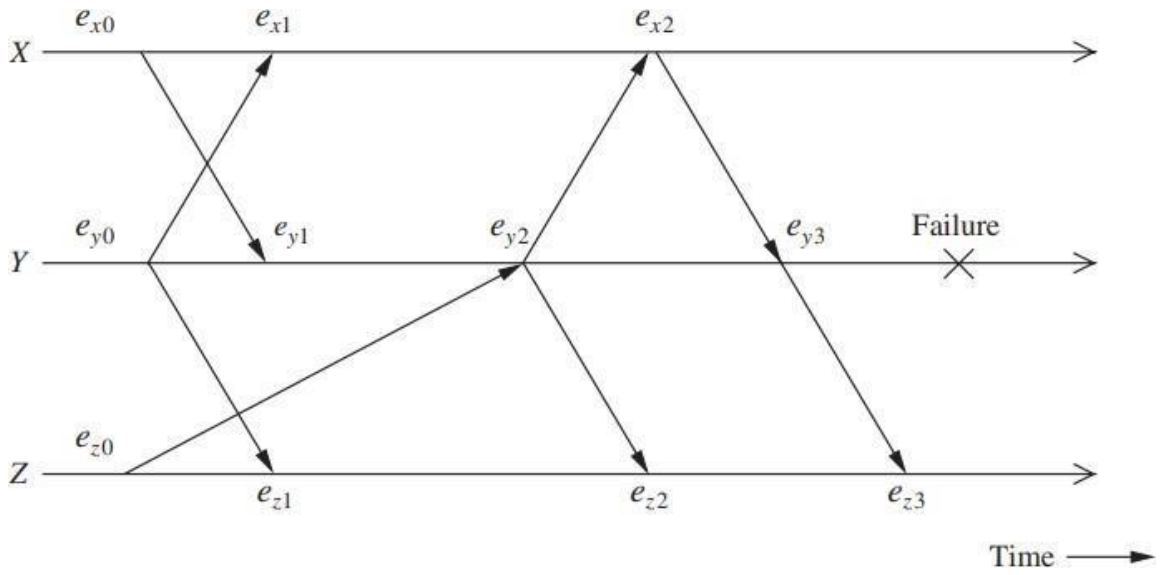
- **System Model and Assumptions**

→ Communication channels are reliable,
→ Delivery messages in FIFO order,
→ Infinite buffers,
→ Message transmission delay is arbitrary but finite.
→ The processors directly connected to a processor via communication channels are called its neighbors.

- **Underlying computation/application is event-driven:**

→ A processor **P** waits until a message **m** is received,
→ Process **P** is at state **s**,
→ receives message **m**,
→ processes the message,
→ changes its state from s to s', and
→ sends zero or more messages to some of its neighbors.

- So the **triplet (s, m, msgs_sent)** represents the state of P
- The events at a processor are identified by unique monotonically increasing numbers, ex0, ex1, ex2, …….



- 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.

- **Asynchronous checkpointing**
- 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.
- **The recovery algorithm**

- **Notations:**
  → $RCVDi \leftarrow j(CkPti)$ **:** number of messages received by pi from pj, from the beginning of computation to checkpoint $CkPti$.

→ $SENTi \rightarrow j(CkPti)$**:** number of messages sent by pi to pj, from the beginning of computation to checkpoint $CkPti$.

- **Idea:**
  → From the set of checkpoints, find a set of consistent checkpoints.
  → Doing that based on the number of messages sent and received.
- **Algorithm:**

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 $RCVD_{i \leftarrow j}(CkPt_i) > c$ {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}

- **Description of the Algorithm:**
  → When a processor restarts after a failure, it broadcasts a ROLLBACK message that it has failed.
  → The recovery algorithm at a processor is initiated when it restarts after a failure.
  → Because of the broadcast of ROLLBACK messages, the recovery algorithm is initiated at all processors.
  → The rollback starts at the failed processor and slowly diffuses into the entire system through ROLLBACK messages.
  → Note that the procedure has |N| iterations.
    - During the kth iteration (k 1), a processor pi does

the following:

- **Example:**

→ 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}.

→ The recovery algorithm is also 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 ROLLBACK(Y , 2) to X and ROLLBACK(Y , 1) to Z;
  - X sends ROLLBACK(X, 2) to Y and ROLLBACK(X, 0) to Z; and
  - Z sends ROLLBACK(Z, 0) to X and ROLLBACK(Z, 1) to Y .
  - Since $RCVD_{X \leftarrow Y}(CkPt_X) = 3 > 2$ (2 is the value received in the ROLLBACK(Y , 2) message from Y), X will set $CkPt_X$ to e_{x2} satisfying $RCVD_{X \leftarrow Y}(e_{x2}) = 2 \leq 2$.

(i) Based on the state **CkPti** it was rolled back in the $(k-1)^{th}$ iteration, it computes $\mathbf{SENT_{i \rightarrow j}(CkPt_i)}$ for each neighbor p_j and sends this value in a ROLLBACK message to that neighbor; and

(ii) p_i waits for and processes ROLLBACK messages that it receives from its neighbors in k^{th} iteration and determines a new recovery point **CkPti** for p_i based on information in these messages.

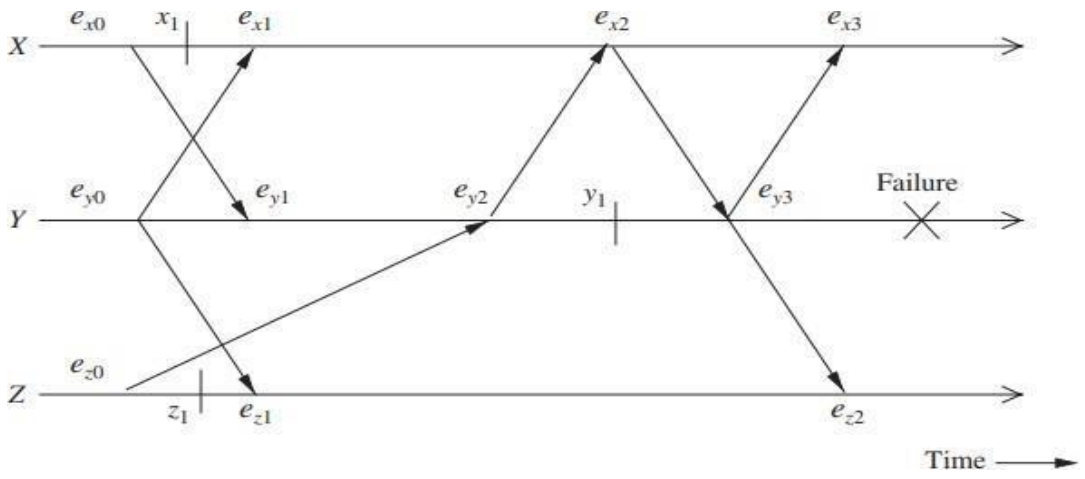
(iii) At the end of each iteration, at least one processor will rollback to its final recovery point, unless the current recovery points are already consistent.

  - Since $RCVD_{Z \leftarrow Y}(CkPt_Z) = 2 > 1$, Z will set $CkPt_Z$ to e_{z1} satisfying $RCVD_{Z \leftarrow Y}(e_{z1}) = 1 \leq 1$.

  - At Y, $RCVD_{Y \leftarrow X}(CkPt_Y) = 1 < 2$ and $RCVD_{Y \leftarrow Z}(CkPt_Y) = 1 = SENT_{Z \leftarrow Y}(CkPt_Z)$.
  - Hence, Y need not roll back further.

→ In the second iteration,

- Y sends ROLLBACK(Y , 2) to X and ROLLBACK(Y , 1) to Z;
- Z sends ROLLBACK(Z, 1) to Y and ROLLBACK(Z, 0) to X;
- X sends ROLLBACK(X, 0) to Z and ROLLBACK(X, 1) to Y .



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 means storing and accessing the data and programs on remote servers that are hosted on the internet instead of the computer's hard drive or local server. Cloud computing is also referred to as Internet-based computing,

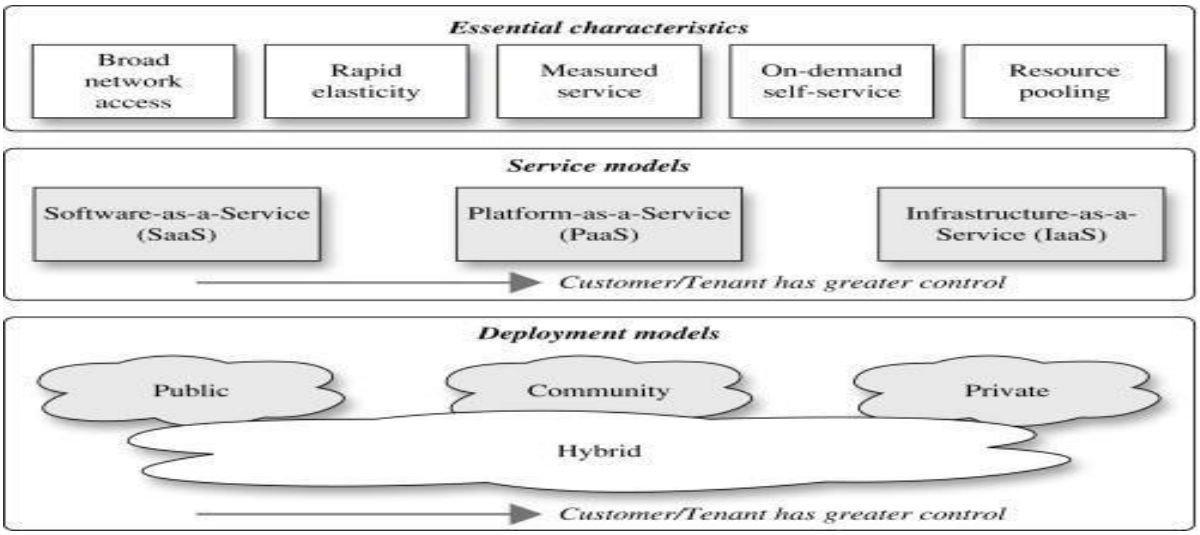
It is a technology where the resource is provided as a service through the Internet to the user. The data which is stored can be files, images, documents, or any other storable document.

Characteristics of CLOUD

- **Scalability:** With Cloud hosting, it is easy to grow and shrink the number and size of servers based on the need. This is done by either increasing or decreasing the resources in the cloud. This ability to alter plans due to fluctuations in business size and needs is a superb benefit of cloud computing, especially when experiencing a sudden growth in demand.
- **Instant:** Whatever you want is instantly available in the cloud.
- **Save Money:** An advantage of cloud computing is the reduction in hardware costs. Instead of purchasing in-house equipment, hardware needs are left to the vendor. For companies that are growing rapidly, new hardware can be large, expensive, and inconvenient. Cloud computing alleviates these issues because resources can be acquired quickly and easily. Even better, the cost of repairing or replacing equipment is passed to the vendors. Along with purchase costs, off-site hardware cuts internal power costs and saves space. Large data centers can take up precious office space and produce a large amount of heat. Moving to cloud applications or storage can help maximize space and significantly cut energy expenditures.
- **Reliability:** Rather than being hosted on one single instance of a physical server, hosting is delivered on a virtual partition that draws its resource, such as disk space, from an extensive network of underlying physical servers. If one server goes offline it will have no effect on availability, as the virtual servers will continue to pull resources from the remaining network of servers.
- **Physical Security:** The underlying physical servers are still housed within data centers and so benefit from the security measures that those facilities implement to prevent people from accessing or disrupting them on-site.
- **Outsource Management:** When you are managing the business, Someone else manages your computing infrastructure. You do not need to worry about management as well as degradation.

CLOUD DEPLOYMENT MODELS

Cloud Deployment Model functions as a virtual computing environment with a deployment architecture that varies depending on the amount of data you want to store and who has access to the infrastructure.



Types of Cloud Computing Deployment Models

The cloud deployment model identifies the specific type of cloud environment based on ownership, scale, and access, as well as the cloud's nature and purpose. The location of the servers utilizing and who controls them are defined by a cloud deployment model.

Different types of cloud computing deployment models are described below.

- **Public Cloud**
- **Private Cloud**
- **Hybrid Cloud**
- **Community Cloud**
- **Multi-Cloud**

Public Cloud

The public cloud makes it possible for anybody to access systems and services. The public cloud may be less secure as it is open to everyone. The infrastructure in this cloud model is owned by the entity that delivers the cloud services, not by the consumer.

It is a type of cloud hosting that allows customers and users to easily access systems and services.

This form of cloud computing is an excellent example of cloud hosting, in which service providers supply services to a variety of customers. In this arrangement, storage backup and retrieval services are given for free, as a subscription, or on a per-user basis.

For example, Google App Engine etc.

Public Cloud

Advantages of the Public Cloud Model

- **Minimal Investment:** Because it is a pay-per-use service, there is no considerable upfront fee, making it excellent for enterprises that require immediate access to resources.
- **No setup cost:** The entire infrastructure is fully subsidized by the cloud service providers, thus there is no need to set up any hardware.
- **Infrastructure Management is not required:** Using the public cloud does not necessitate infrastructure management.
- **No maintenance:** The maintenance work is done by the service provider (not users).
- **Dynamic Scalability:** To fulfill your company's needs, on-demand resources are accessible.

Disadvantages of the Public Cloud Model

- **Less secure:** Public cloud is less secure as resources are public so there is no guarantee of high-level security.
- **Low customization:** It is accessed by many public so it can't be customized according to personal requirements.

Private Cloud

The private cloud deployment model is the exact opposite of the public cloud deployment model. It's a one-on-one environment for a single user (customer).

It is also called the "internal cloud" & it refers to the ability to access systems and services within a given border or organization. The cloud platform is implemented in a cloud-based secure environment that is protected by powerful firewalls and under the supervision of an organization's IT department. The private cloud gives greater flexibility of control over cloud resources.

Advantages of the Private Cloud Model

- **Better Control:** You are the sole owner of the property. You gain complete command over service integration, IT operations, policies, and user behavior.
- **Data Security and Privacy:** It's suitable for storing corporate information to which only authorized staff have access. By segmenting resources within the same infrastructure, improved access and security can be achieved.

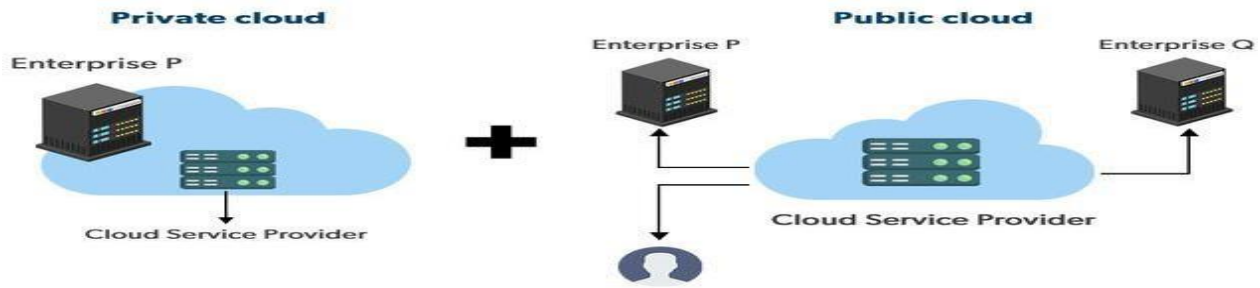
- **Supports Legacy Systems:** This approach is designed to work with old systems that are unable to access the public cloud.
- **Customization:** Unlike a public cloud deployment, a private cloud allows a company to tailor its solution to meet its specific needs.

Disadvantages of the Private Cloud Model

- **Less scalable:** Private clouds are scaled within a certain range as there is less number of clients.
- **Costly:** Private clouds are more costly as they provide personalized facilities.

Hybrid Cloud

By bridging the public and private worlds with a layer of proprietary software, hybrid cloud computing gives the best of both worlds. With a hybrid solution, you may host the app in a safe environment while taking advantage of the public cloud's cost savings. Organizations can move data and applications between different clouds using a combination of two or more cloud deployment methods, depending on their needs.



Advantages of the Hybrid Cloud Model

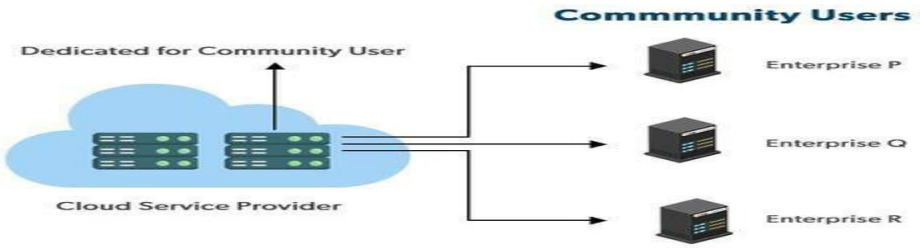
- **Flexibility and control:** Businesses with more flexibility can design personalized solutions that meet their particular needs.
- **Cost:** Because public clouds provide scalability, you'll only be responsible for paying for the extra capacity if you require it.
- **Security:** Because data is properly separated, the chances of data theft by attackers are considerably reduced.

Disadvantages of the Hybrid Cloud Model

- **Difficult to manage:** Hybrid clouds are difficult to manage as it is a combination of both public and private cloud. So, it is complex.
- **Slow data transmission:** Data transmission in the hybrid cloud takes place through the public cloud so latency occurs.

Community Cloud

It allows systems and services to be accessible by a group of organizations. It is a distributed system that is created by integrating the services of different clouds to address the specific needs of a community, industry, or business. The infrastructure of the community could be shared between the organization which has shared concerns or tasks. It is generally managed by a third party or by the combination of one or more organizations in the community.



Advantages of the Community Cloud Model

- **Cost Effective:** It is cost-effective because the cloud is shared by multiple organizations or communities.
- **Security:** Community cloud provides better security.
- **Shared resources:** It allows you to share resources, infrastructure, etc. with multiple organizations.
- **Collaboration and data sharing:** It is suitable for both collaboration and data sharing.

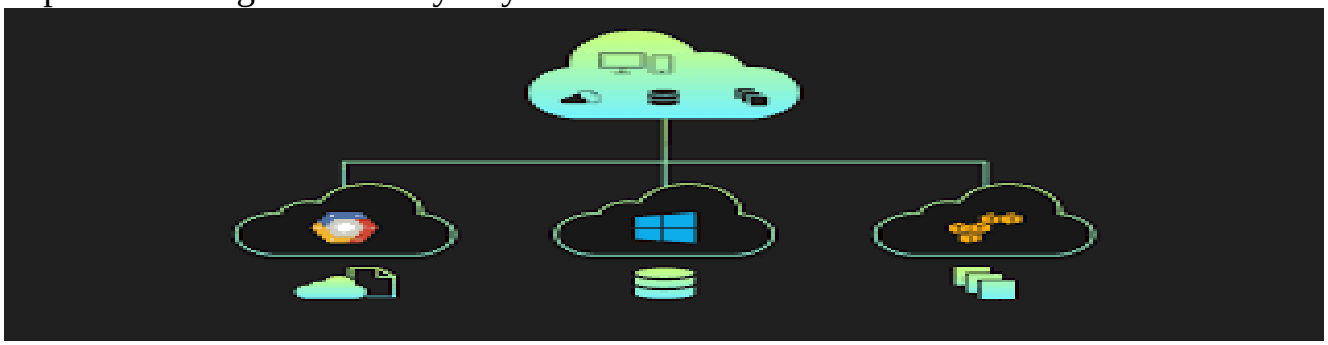
Disadvantages of the Community Cloud Model

- **Limited Scalability:** Community cloud is relatively less scalable as many organizations share the same resources according to their collaborative interests.
- **Rigid in customization:** As the data and resources are shared among different organizations according to their mutual interests.

Multi-Cloud

It's similar to the hybrid cloud deployment approach, which combines public and private cloud resources.

Public cloud providers provide numerous tools to improve the reliability of their services, mishaps still occur. It's quite rare that two distinct clouds would have an incident at the same moment. As a result, multi-cloud deployment improves the high availability of your services even more.



Advantages of the Multi-Cloud Model

- You can mix and match the best features of each cloud provider's services to suit the demands of your apps, workloads, and business by choosing different cloud providers.
- **Reduced Latency:** To reduce latency and improve user experience, you can choose cloud regions and zones that are close to your clients.
- **High availability of service:** It's quite rare that two distinct clouds would have an incident at the same moment. So, the multi-cloud deployment improves the high availability of your services.

Disadvantages of the Multi-Cloud Model

- **Complex:** The combination of many clouds makes the system complex and bottlenecks may occur.

Security issue: Due to the complex structure, there may be loopholes to which a hacker can take advantage hence, makes the Cloud service models in cloud computing refer to the standardized framework functioning for delivering computing resources and services on the internet. It defines the structure and component framework defining the way services are offered, managed, and loud Computing can be defined as the practice of using a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server or a personal computer. Companies offering such kinds of cloud computing services are called *cloud providers* and typically charge for cloud computing services based on usage. Grids and clusters are the foundations for cloud computing.

CLOUD SERVICE MODELS

Most cloud computing services fall into five broad categories:

1. Software as a service (SaaS)
2. Platform as a service (PaaS)
3. Infrastructure as a service (IaaS)
4. Anything/Everything as a service (XaaS)
5. Function as a Service (FaaS)

These are sometimes called the **cloud computing stack** because they are built on top of one another. Knowing what they are and how they are different, makes it easier to accomplish your goals. These abstraction layers can also be viewed as a **layered architecture** where services of a higher layer can be composed of services of the underlying layer i.e, SaaS can provide Infrastructure.

Software as a Service(SaaS)

Software-as-a-Service (SaaS) is a way of delivering services and applications over the Internet. Instead of installing and maintaining software, we simply access it via the Internet, freeing ourselves from the complex software and hardware management. It removes the need to install and run applications on our own computers or in the data center eliminating the expenses of hardware as well as software maintenance.

SaaS provides a complete software solution that you purchase on a **pay-as-you-go** basis from a cloud service provider. Most SaaS applications can be run directly from a web browser without any downloads or installations required. The SaaS applications are sometimes called **Web-based software, on-demand software, or hosted software.**

Advantages of SaaS

1. **Cost-Effective:** Pay only for what you use.
2. **Reduced time:** Users can run most SaaS apps directly from their web browser without needing to download and install any software. This reduces the time spent in installation and configuration and can reduce the issues that can get in the way of the software deployment.
3. **Accessibility:** We can Access app data from anywhere.

4. **Automatic updates:** Rather than purchasing new software, customers rely on a SaaS provider to automatically perform the updates.
5. **Scalability:** It allows the users to access the services and features on-demand.

The various companies providing *Software as a service* are Cloud Analytics, Salesforce.com, Cloud Switch, Microsoft Office 365, Big Commerce, Eloqua, dropBox, and Cloud Tran.

Disadvantages of Saas :

1. **Limited customization**: SaaS solutions are typically not as customizable as on-premises software, meaning that users may have to work within the constraints of the SaaS provider's platform and may not be able to tailor the software to their specific needs.
2. **Dependence on internet connectivity**: SaaS solutions are typically cloud-based, which means that they require a stable internet connection to function properly. This can be problematic for users in areas with poor connectivity or for those who need to access the software in offline environments.
3. **Security concerns:** SaaS providers are responsible for maintaining the security of the data stored on their servers, but there is still a risk of data breaches or other security incidents.
4. **Limited control over data:** SaaS providers may have access to a user's data, which can be a concern for organizations that need to maintain strict control over their data for regulatory or other reasons.

Platform as a Service

PaaS is a category of cloud computing that provides a platform and environment to allow developers to build applications and services over the internet.

PaaS services are hosted in the cloud and accessed by users simply via their web browser.

A PaaS provider hosts the hardware and software on its own infrastructure. As a result, PaaS frees users from having to install in-house hardware and software to develop or run a new application. Thus, the development and deployment of the application take place **independent of the hardware**.

The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment. To make it simple, take the example of an annual day function, you will have two options either to create a venue or to rent a venue but the function is the same.

Advantages of PaaS:

1. **Simple and convenient for users:** It provides much of the infrastructure and other IT services, which users can access anywhere via a web browser.
2. **Cost-Effective:** It charges for the services provided on a per-use basis thus eliminating the expenses one may have for on-premises hardware and software.

3. **Efficiently managing the lifecycle:** It is designed to support the complete web application lifecycle: building, testing, deploying, managing, and updating.
4. **Efficiency:** It allows for higher-level programming with reduced complexity thus, the overall development of the application can be more effective.

The various companies providing *Platform as a service* are Amazon Web services Elastic Beanstalk, Salesforce, Windows Azure, Google App Engine, cloud Bees and IBM smart cloud.

Disadvantages of Paas:

1. **Limited control over infrastructure:** PaaS providers typically manage the underlying infrastructure and take care of maintenance and updates, but this can also mean that users have less control over the environment and may not be able to make certain customizations.
2. **Dependence on the provider**: Users are dependent on the PaaS provider for the availability, scalability, and reliability of the platform, which can be a risk if the provider experiences outages or other issues.
3. **Limited flexibility:** PaaS solutions may not be able to accommodate certain types of workloads or applications, which can limit the value of the solution for certain organizations.

Infrastructure as a Service

Infrastructure as a service (IaaS) is a service model that delivers computer infrastructure on an outsourced basis to support various operations. Typically IaaS is a service where infrastructure is provided as outsourcing to enterprises such as networking equipment, devices, database, and web servers. It is also known as **Hardware as a Service (HaaS).** IaaS customers pay on a per-user basis, typically by the hour, week, or month. Some providers also charge customers based on the amount of virtual machine space they use. It simply provides the underlying operating systems, security, networking, and servers for developing such applications, and services, and deploying development tools, databases, etc.

Advantages of IaaS:

1. **Cost-Effective:** Eliminates capital expense and reduces ongoing cost and IaaS customers pay on a per-user basis, typically by the hour, week, or month.
2. **Website hosting:** Running websites using IaaS can be less expensive than traditional web hosting.
3. **Security:** The IaaS Cloud Provider may provide better security than your existing software.
4. **Maintenance:** There is no need to manage the underlying data center or the introduction of new releases of the development or underlying software. This is all handled by the IaaS Cloud Provider.

The various companies providing *Infrastructure as a service* are Amazon web services, Bluestack, IBM, Openstack, Rackspace, and Vmware.

Disadvantages of laaS :

1. **Limited control over infrastructure:** IaaS providers typically manage the underlying infrastructure and take care of maintenance and updates, but this can also mean that users have less control over the environment and may not be able to make certain customizations.
2. **Security concerns**: Users are responsible for securing their own data and applications, which can be a significant undertaking.
3. **Limited access:** Cloud computing may not be accessible in certain regions and countries due to legal policies.

Anything as a Service

It is also known as Everything as a Service. Most of the cloud service providers nowadays offer anything as a service that is a compilation of all of the above services including some additional services.

Advantages of XaaS:

1. **Scalability:** XaaS solutions can be easily scaled up or down to meet the changing needs of an organization.
2. **Flexibility:** XaaS solutions can be used to provide a wide range of services, such as storage, databases, networking, and software, which can be customized to meet the specific needs of an organization.
3. **Cost-effectiveness**: XaaS solutions can be more cost-effective than traditional on-premises solutions, as organizations only pay for the services.

Disadvantages of XaaS:

1. **Dependence on the provider:** Users are dependent on the XaaS provider for the availability, scalability, and reliability of the service, which can be a risk if the provider experiences outages or other issues.
2. **Limited flexibility**: XaaS solutions may not be able to accommodate certain types of workloads or applications, which can limit the value of the solution for certain organizations.
3. **Limited integration:** XaaS solutions may not be able to integrate with existing systems and data sources, which can limit the value of the solution for certain organizations.

Function as a Service :

FaaS is a type of cloud computing service. It provides a platform for its users or customers to develop, compute, run and deploy the code or entire application as functions.

It allows the user to entirely develop the code and update it at any time without worrying about the maintenance of the underlying infrastructure. The developed code can be executed with response to the specific event. It is also **as same as PaaS**.

FaaS is an event-driven execution model. It is implemented in the serverless container. When the application is developed completely, the user will now trigger the event to execute the code. Now, the triggered event makes response and activates the servers to execute it.

The servers are nothing but the Linux servers or any other servers which is managed by the vendor completely. Customer does not have clue about any servers

which is why they do not need to maintain the server hence it is **serverless architecture.**

Both PaaS and FaaS are providing the same functionality but there is still some differentiation in terms of Scalability and Cost.

FaaS, provides auto-scaling up and scaling down depending upon the demand. PaaS also provides scalability but here users have to configure the scaling parameter depending upon the demand.

In FaaS, users only have to pay for the number of execution time happened. In PaaS, users have to pay for the amount based on pay-as-you-go price regardless of how much or less they use.

Advantages of FaaS :

- **Highly Scalable:** Auto scaling is done by the provider depending upon the demand.
- **Cost-Effective:** Pay only for the number of events executed.
- **Code Simplification:** FaaS allows the users to upload the entire application all at once. It allows you to write code for independent functions or similar to those functions.
- Maintenance of code is enough and no need to worry about the servers.
- Functions can be written in any programming language.
- Less control over the system.

The various companies providing Function as a Service are Amazon Web Services – Firecracker, Google – Kubernetes, Oracle – Fn, Apache OpenWhisk – IBM, OpenFaaS,

Disadvantages of FaaS :

1. **Cold start latency**: Since FaaS functions are event-triggered, the first request to a new function may experience increased latency as the function container is created and initialized.
2. **Limited control over infrastructure:** FaaS providers typically manage the underlying infrastructure and take care of maintenance and updates, but this can also mean that users have less control over the environment and may not be able to make certain customizations.
3. **Security concerns:** Users are responsible for securing their own data and applications, which can be a significant undertaking.
4. **Limited scalability**: FaaS functions may not be able to handle high traffic or large number of requests.

Cloud computing is the resources of data and storage in real-time. It has been proven to be revolutionary in the IT industry with the market valuation growing at a rapid rate. Cloud development has proved to be beneficial not only for huge public and private enterprises but small-scale businesses as well as it helps to cut costs. It is estimated that more than 94% of businesses will increase their spending on the cloud by more than 45%. This also has resulted in more and high-paying jobs if you are a cloud developer.

Cloud technology was flourishing before the pandemic, but there has been a sudden spike in cloud deployment and usage during the lockdown. The tremendous growth can be linked to the fact that classes have been shifted online, virtual office meetings are happening on video calling platforms, conferences are taking place

virtually as well as on-demand streaming apps have a huge audience. All this is made possible by us of cloud computing only.

So, cloud is an important part of our life today, even if we are an enterprise, student, developer, or anyone else and are heavily dependent on it. But with this dependence, it is also important for us to look at the issues and challenges that arise with cloud computing. These are the most common challenges that are faced when dealing with cloud computing.

CHALLENGES OF CLOUD

1. Data Security and Privacy

Data security is a major concern when working with Cloud environments. It is one of the major challenges in cloud computing as users have to take accountability for their data, and not all Cloud providers can assure 100% data privacy. Lack of visibility and control tools, no identity access management, data misuse, and Cloud misconfiguration are the common causes behind Cloud privacy leaks. There are also concerns with insecure APIs, malicious insiders, and oversights or neglect in Cloud data management.

Solution: Configure network hardware and install the latest software updates to prevent security vulnerabilities. Using firewalls, antivirus, and increasing bandwidth for Cloud data availability are some ways to prevent data security risks.

2. Multi-Cloud Environments

Common cloud computing issues and challenges with multi-cloud environments are - configuration errors, lack of security patches, data governance, and no granularity. It is difficult to track the security requirements of multi-clouds and apply data management policies across various boards.

Solution: Using a multi-cloud data management solution is a good start for enterprises. Not all tools will offer specific security functionalities, and multi-cloud environments grow highly sophisticated and complex. Open-source products like Terraform provide a great deal of control over multi-cloud architectures.

3. Performance Challenges

The performance of Cloud computing solutions depends on the vendors who offer these services to clients, and if a Cloud vendor goes down, the business gets affected too. It is one of the major challenges associated with cloud computing.

Solution: Sign up with Cloud Service Providers who have real-time SaaS monitoring policies.

The **Cloud Solution Architect Certification** training addresses all Cloud performance issues and teaches learners how to mitigate them.

4. Interoperability and Flexibility

Interoperability is a challenge when you try to move applications between two or multiple Cloud ecosystems. It is one of the challenges faced in cloud computing. Some common issues faced are:

- Rebuilding application stacks to match the target cloud environment's specifications
- Handling data encryption during migration
- Setting up networks in the target cloud for operations

- Managing apps and services in the target cloud ecosystem

Solution: Setting Cloud interoperability and portability standards in organizations before getting to work on projects can help solve this problem. The use of multi-layer authentication and authorization tools is also encouraged for account verifications in public, private, and hybrid cloud ecosystems.

5. High Dependence on Network

Lack of sufficient internet bandwidth is a common problem when transferring large volumes of information to and from Cloud data servers. It is one of the various challenges in cloud computing. Data is highly vulnerable, and there is a risk of sudden outages. Enterprises that want to lower hardware costs without sacrificing performance need to ensure there is high bandwidth, which will help prevent business losses from sudden outages.

Solution: Pay more for higher bandwidth and focus on improving operational efficiency to address network dependencies.

6. Lack of Knowledge and Expertise

Organizations are finding it tough to find and hire the right Cloud talent, which is another common challenge in cloud computing. There is a shortage of professionals with the required qualifications in the industry. Workloads are increasing, and the number of tools launched in the market is increasing. Enterprises need good expertise in order to use these tools and find out which ones are ideal for them.

Solution: Hire Cloud professionals with specializations in DevOps and automation

7. Reliability and Availability

High unavailability of Cloud services and a lack of reliability are two major concerns in these ecosystems. Organizations are forced to seek additional computing resources in order to keep up with changing business requirements. If a Cloud vendor gets hacked or affected, the data of organizations using their services gets compromised. It is another one of the many cloud security risks and challenges faced by the industry.

Solution: Implementing the NIST Framework standards in Cloud environments can greatly improve both aspects.

8. Password Security

Account managers use the same passwords to manage all their Cloud accounts. Password management is a critical problem, and it is often found that users resort to using reused and weak passwords.

Solution: Use a strong password management solution to secure all your accounts. To further improve security, use Multifactor Authentication (MFA) in addition to a password manager. Good cloud-based password managers alert users of security risks and leaks.

9. Cost Management

Even though Cloud Service Providers (CSPs) offer a pay-as-you-go subscription for services, the costs can add up. Hidden costs appear in the form of underutilized resources in enterprises.

Solution: Auditing systems regularly and implementing resource utilization monitoring tools are some ways organizations can fix this. It's one of the most effective ways to manage budgets and deal with major challenges in cloud computing.

10. Lack of expertise

Cloud computing is a highly competitive field, and there are many professionals who lack the required skills and knowledge to work in the industry. There is also a huge gap in supply and demand for certified individuals and many job vacancies.

Solution: Companies should retrain their existing IT staff and help them in upskilling their careers by investing in Cloud training programs.

11. Control or Governance

Good IT governance ensures that the right tools are used, and assets get implemented according to procedures and agreed-to policies. Lack of governance is a common problem, and companies use tools that do not align with their vision. IT teams don't get total control of compliance, risk management, and data quality checks, and there are many uncertainties faced when migrating to the Cloud from traditional infrastructures.

Solution: Traditional IT processes should be adopted in ways to accommodate Cloud migrations.

12. Compliance

Cloud Service Providers (CSP) are not up-to-date when it comes to having the best data compliance policies. Whenever a user transfers data from internal servers to the Cloud, they run into compliance issues with state laws and regulations.

Solution: The General Data Protection Regulation (GDPR) Act is expected to expedite compliance issues in the future for CSPs.

13. Multiple Cloud Management

Enterprises depend on multiple cloud environments due to scaling up and provisioning resources. One of the hybrid cloud security challenges is that most companies follow a hybrid cloud strategy, and many resort to multi-cloud. The problem is that infrastructures grow increasingly complex and difficult to manage when multiple cloud providers get added, especially due to technological cloud computing challenges and differences.

Solution: Creating strong data management and privacy policies is a starting point when it comes to managing multi-cloud environments effectively.

14. Migration

Migration of data to the Cloud takes time, and not all organizations are prepared for it. Some report increased downtimes during the process, face security issues, or have problems with data formatting and conversions. Cloud migration

projects can get expensive and are harder than anticipated.

Solution: Organizations will have to employ in-house professionals to handle their Cloud data migration and increase their investments. Experts must analyze cloud computing issues and solutions before investing in the latest platforms and services offered by CSPs.

15. Hybrid-Cloud Complexity

Hybrid-cloud complexity refers to cloud computing challenges arising from mixed computing, storage, and services, and multi-cloud security causes various challenges. It comprises private cloud services, public Clouds, and on-premises infrastructures, for example, products like Microsoft Azure and Amazon Web Services - which are orchestrated on various platforms.

Solution: Using centralized Cloud management solutions, increasing automation, and hardening security are good ways to mitigate hybrid-cloud complexity.

DRIVING FACTORS OF CLOUD

BusinessDriver

It is the interface or resource, and a process that is used for the growth and success of the business. Every business has its own driver to which they decide as per the circumstances. Business drivers are the key inputs that drive a business operationally and financially. Businesses have been motivated to adopt such business drivers to achieve organizational goals.

Example

Some common examples of business drivers are the quantity and price of the products sold, units of production, number of enterprises, salespeople, etc.

BusinessDriversin CloudComputing :

Business drivers have motivated organizations to adopt cloud computing to meet and support the requirements of these drivers. They have also motivated organizations to become providers of the cloud environment. There are three types of Business Drivers as follows.

1. Capacity Planning
2. Cost Reduction
3. Organizational Agility

CapacityPlanning

Capacity planning is the process in which an organization estimates the production capacity needed for its products to cope with the ever-changing demands in the market. This involves estimating the storage, infrastructure, hardware and software, availability of resources, etc. for over a future period of time. There are three major consideration's incapacity planning as follows.

1. Level of Demand
2. Cost of Production
3. Availability of Funds

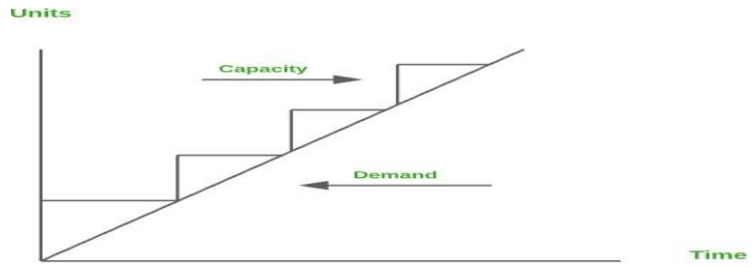
OtherStrategy

Taking these considerations into account let us look at the different capacity

planning strategies that exist. Let's discuss it one by one.

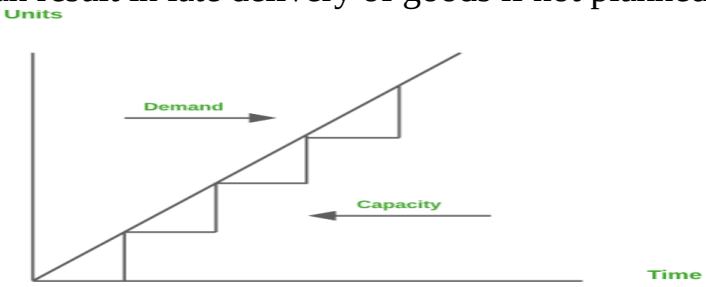
LeadStrategy

This is a strategy where the capacity is added beforehand in reference to a future increase in demand. This strategy keeps the customers intact and prevents competitors from luring them back in.



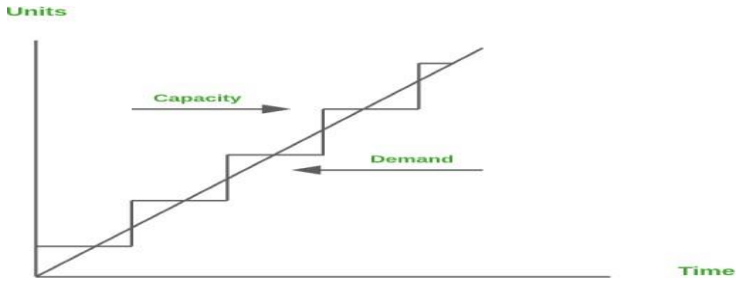
Lag Strategy

This strategy is where the capacity is added only when it is required, that is, only when the demand is observed and not based on anticipation. This strategy is more conservative, as it reduces the risk of wastage but at the same time, it can result in late delivery of goods if not planned outright.



Match Strategy

This strategy is where small amounts of capacity are added gradually in required intervals of time, keeping in mind the demand and the market potential of the product. This strategy is said to improve performance in heterogeneous environments and hybrid clouds.



Cost Reduction

Cost reduction is the process by which organizations reduce unnecessary costs in order to increase their profits in the business. There is a direct alignment between the cost and the growth of the company, which is why cost reduction is an important factor in the organization's productivity. The maximum usage requirements should be kept in mind when dealing with the performance of the organization.

Cost factor

Two costs should be taken into account as follows.

- The cost of acquiring new infrastructure.
- The cost of its ongoing ownership.

Tools and Techniques for cost reduction

There are the following tools and techniques that are used to reduce costs as follows.

- Budgetary Control
- Standard Costing
- Simplification and Variety Reduction
- Planning and Control of Finance
- Cost-Benefit Analysis
- Value Analysis

Organizational Agility

Organization agility is the process by which an organization will adapt and evolve to sudden changes caused by internal and external factors. It measures how quickly an organization will get back on its feet, in the face of problems. Agility requires stability, and for an organization to reach organizational agility, it should build a stable foundation. In the IT field, one should respond to business change by scaling its IT resources. If infrastructure seems to be the problem, changing the business needs and prioritizing as per the circumstances should be the solution.

PrincipleofOrganizationalAgility

The five principles of Organizational Agility are as follows.

1. Frame your problems properly
2. Limit Change
3. Simplify Change
4. Subtract before you Add
5. Verify Outcomes

VIRTUALIZATION

Virtualization is a technique how to separate a service from the underlying physical delivery of that service. It is the process of creating a virtual version of something like computer hardware. It was initially developed during the mainframe era.

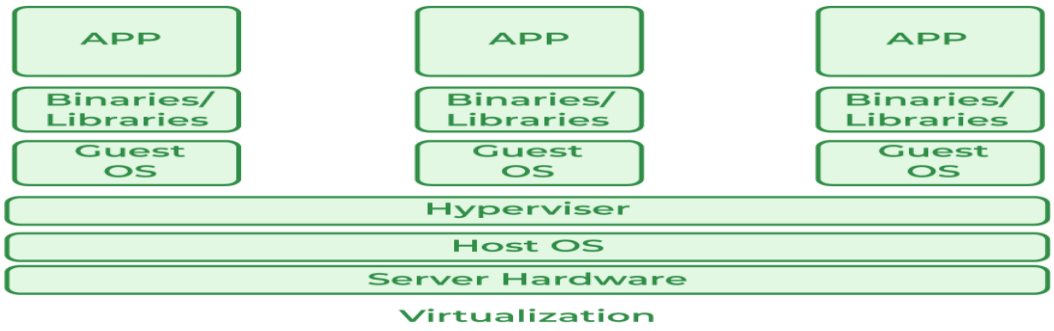
It involves using specialized software to create a virtual or software-created version of a computing resource rather than the actual version of the same resource. With the help of Virtualization, multiple operating systems and applications can run on the same machine and its same hardware at the same time, increasing the utilization and flexibility of hardware.

In other words, one of the main cost-effective, hardware-reducing, and energy-saving techniques used by cloud providers is Virtualization.

Virtualization allows sharing of a single physical instance of a resource or an application among multiple customers and organizations at one time. It does this by assigning a logical name to physical storage and providing a pointer to that physical resource on demand.

The term virtualization is often synonymous with hardware virtualization, which plays a fundamental role in efficiently delivering

Infrastructure-as-a-Service (IaaS) solutions for cloud computing. Moreover, virtualization technologies provide a virtual environment for not only executing applications but also for storage, memory, and networking.



- Host Machine: The machine on which the virtual machine is going to be built is known as Host Machine.
- Guest Machine: The virtual machine is referred to as a Guest Machine.

Work of Virtualization in Cloud Computing

Virtualization has a prominent impact on Cloud Computing. In the case of cloud computing, users store data in the cloud, but with the help of Virtualization, users have the extra benefit of sharing the infrastructure. Cloud Vendors take care of the required physical resources, but these cloud providers charge a huge amount for these services which impacts every user or organization. Virtualization helps Users or Organisations in maintaining those services which are required by a company through external (third-party) people, which helps in reducing costs to the company. This is the way through which Virtualization works in Cloud Computing.

Benefits of Virtualization

- More flexible and efficient allocation of resources.
- Enhance development productivity.
- It lowers the cost of IT infrastructure.
- Remote access and rapid scalability.
- High availability and disaster recovery.
- Pay peruse of the IT infrastructure on demand.
- Enables running multiple operating systems.

Drawback of Virtualization

- **High Initial Investment:** Clouds have a very high initial investment, but it is also true that it will help in reducing the cost of companies.
- **Learning New Infrastructure:** As the companies shifted from Servers to Cloud, it requires highly skilled staff who have skills to work with the cloud easily, and for this, you have to hire new staff or provide training to current staff.
- **Risk of Data:** Hosting data on third-party resources can lead to putting the data at risk, it has the chance of getting attacked by any hacker or cracker very easily.

For more benefits and drawbacks, you can refer to the Pros and Cons of Virtualization.

Characteristics of Virtualization

- **Increased Security:** The ability to control the execution of a guest program in a completely transparent manner opens new possibilities for delivering a secure, controlled execution environment. All the operations

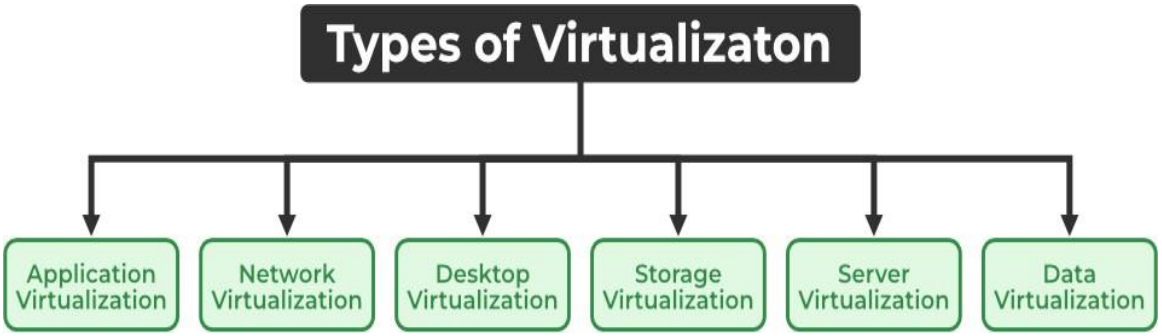
of the guest programs are generally performed against the virtual machine, which then translates and applies them to the host programs.

- **Managed Execution:** In particular, sharing, aggregation, emulation, and isolation are the most relevant features.
- **Sharing:** Virtualization allows the creation of a separate computing environment within the same host.
- **Aggregation:** It is possible to share physical resources among several guests, but virtualization also allows aggregation, which is the opposite process.

For more characteristics, you can refer to Characteristics of Virtualization.

Types of Virtualization

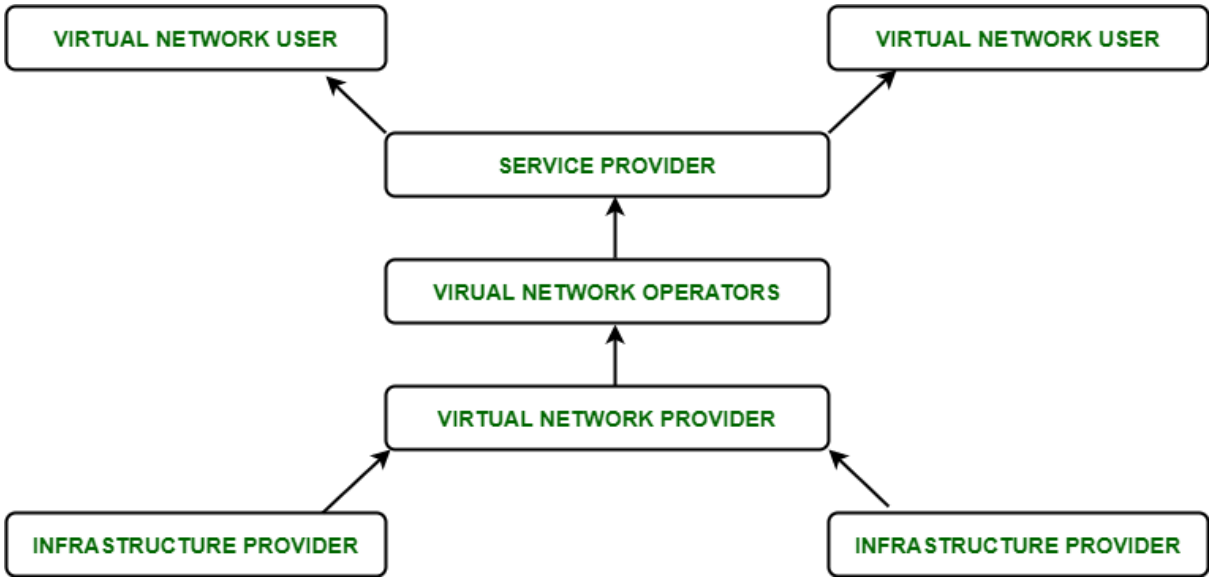
1. Application Virtualization
2. Network Virtualization
3. Desktop Virtualization
4. Storage Virtualization
5. Server Virtualization
6. Data virtualization



Types of Virtualization

1. Application Virtualization: Application virtualization helps a user to have remote access to an application from a server. The server stores all personal information and other characteristics of the application but can still run on a local workstation through the internet. An example of this would be a user who needs to run two different versions of the same software. Technologies that use application virtualization are hosted applications and packaged applications.

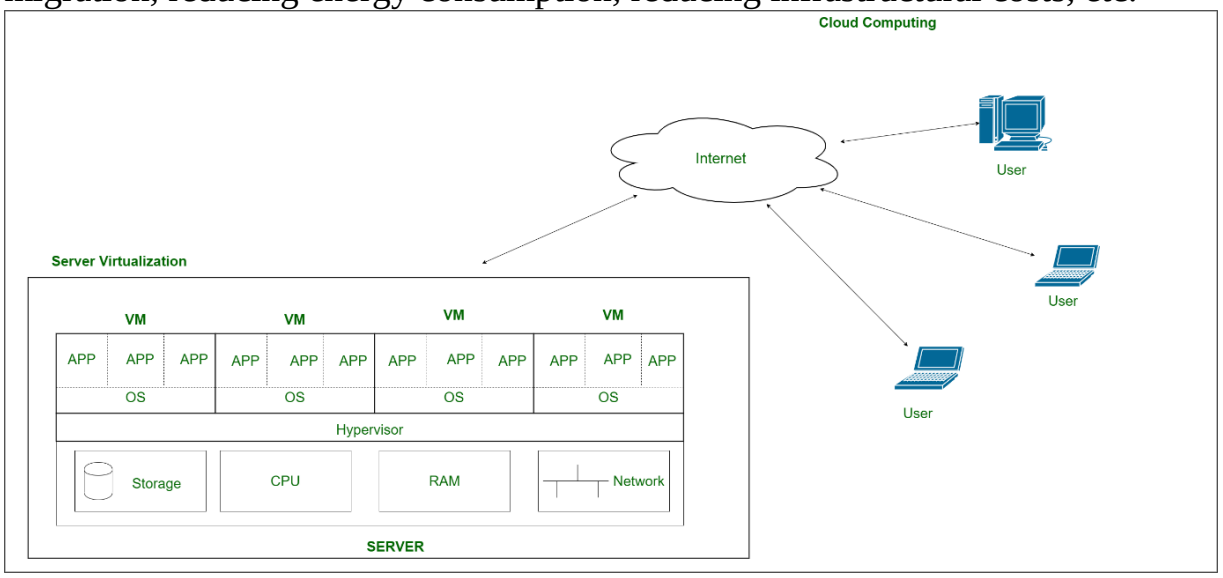
2. Network Virtualization: The ability to run multiple virtual networks with each having a separate control and data plan. It co-exists together on top of one physical network. It can be managed by individual parties that are potentially confidential to each other. Network virtualization provides a facility to create and provision virtual networks, logical switches, routers, firewalls, load balancers, Virtual Private Networks (VPN), and workload security within days or even weeks.



3. Desktop Virtualization: Desktop virtualization allows the users' OS to be remotely stored on a server in the data center. It allows the user to access their desktop virtually, from any location by a different machine. Users who want specific operating systems other than Windows Server will need to have a virtual desktop. The main benefits of desktop virtualization are user mobility, portability, and easy management of software installation, updates, and patches.

4. Storage Virtualization: Storage virtualization is an array of servers that are managed by a virtual storage system. The servers aren't aware of exactly where their data is stored and instead function more like worker bees in a hive. It makes managing storage from multiple sources be managed and utilized as a single repository. storage virtualization software maintains smooth operations, consistent performance, and a continuous suite of advanced functions despite changes, breaks down, and differences in the underlying equipment.

5. Server Virtualization: This is a kind of virtualization in which the masking of server resources takes place. Here, the central server (physical server) is divided into multiple different virtual servers by changing the identity number, and processors. So, each system can operate its operating systems in an isolated manner. Where each sub-server knows the identity of the central server. It causes an increase in performance and reduces the operating cost by the deployment of main server resources into a sub-server resource. It's beneficial in virtual migration, reducing energy consumption, reducing infrastructural costs, etc.



Server Virtualization

6. Data Virtualization: This is the kind of virtualization in which the data is collected from various sources and managed at a single place without knowing more about the technical information like how data is collected, stored & formatted then arranged that data logically so that its virtual view can be accessed by its interested people and stakeholders, and users through the various cloud services remotely. Many big giant companies are providing their services like Oracle, IBM, At scale, Cdata, etc.

Uses of Virtualization

- Data-integration
- Business-integration
- Service-oriented architecture data-services
- Searching organizational data

LOAD BALANCING

Load balancing is an essential technique used in cloud computing to optimize resource utilization and ensure that no single resource is overburdened with traffic. It is a process of distributing workloads across multiple computing resources, such as servers, virtual machines, or containers, to achieve better performance, availability, and scalability.

1. In cloud computing, load balancing can be implemented at various levels, including the network layer, application layer, and database layer. The most common load balancing techniques used in cloud computing are:
2. Network Load Balancing: This technique is used to balance the network traffic across multiple servers or instances. It is implemented at the network layer and ensures that the incoming traffic is distributed evenly across the available servers.
3. Application Load Balancing: This technique is used to balance the workload across multiple instances of an application. It is implemented at the application layer and ensures that each instance receives an equal share of the incoming requests.
4. Database Load Balancing: This technique is used to balance the workload across multiple database servers. It is implemented at the database layer and ensures that the incoming queries are distributed evenly across the available database servers.

Load balancing helps to improve the overall performance and reliability of cloud-based applications by ensuring that resources are used efficiently and that there is no single point of failure. It also helps to scale applications on demand and provides high availability and fault tolerance to handle spikes in traffic or server failures.

Sure, here are some advantages and disadvantages of load balancing in cloud computing:

Advantages:

1. Improved Performance: Load balancing helps to distribute the workload across multiple resources, which reduces the load on each resource and improves the overall performance of the system.
2. High Availability: Load balancing ensures that there is no single point of failure in the system, which provides high availability and fault tolerance to handle server failures.

3. Scalability: Load balancing makes it easier to scale resources up or down as needed, which helps to handle spikes in traffic or changes in demand.
4. Efficient Resource Utilization: Load balancing ensures that resources are used efficiently, which reduces wastage and helps to optimize costs.

Disadvantages:

1. Complexity: Implementing load balancing in cloud computing can be complex, especially when dealing with large-scale systems. It requires careful planning and configuration to ensure that it works effectively.
2. Cost: Implementing load balancing can add to the overall cost of cloud computing, especially when using specialized hardware or software.
3. Single Point of Failure: While load balancing helps to reduce the risk of a single point of failure, it can also become a single point of failure if not implemented correctly.
4. Security: Load balancing can introduce security risks if not implemented correctly, such as allowing unauthorized access or exposing sensitive data.

Overall, the benefits of load balancing in cloud computing outweigh the disadvantages, as it helps to improve performance, availability, scalability, and resource utilization. However, it is important to carefully plan and implement load balancing to ensure that it works effectively and does not introduce additional risks.

Cloud load balancing is defined as the method of splitting workloads and computing properties in a cloud computing. It enables enterprise to manage workload demands or application demands by distributing resources among numerous computers, networks or servers. Cloud load balancing includes holding the circulation of workload traffic and demands that exist over the Internet. As the traffic on the internet growing rapidly, which is about 100% annually of the present traffic. Hence, the workload on the server growing so fast which leads to the overloading of servers mainly for popular web server. There are two elementary solutions to overcome the problem of overloading on the servers-

- First is a single-server solution in which the server is upgraded to a higher performance server. However, the new server may also be overloaded soon, demanding another upgrade. Moreover, the upgrading process is arduous and expensive.
- Second is a multiple-server solution in which a scalable service system on a cluster of servers is built. That's why it is more cost effective as well as more scalable to build a server cluster system for network services.

Load balancing is beneficial with almost any type of service, like HTTP, SMTP, DNS, FTP, and POP/IMAP. It also rises reliability through redundancy. The balancing service is provided by a dedicated hardware device or program. Cloud-based servers farms can attain more precise scalability and availability using server load balancing. **Load balancing solutions can be categorized into two types –**

1. **Software-based load balancers:** Software-based load balancers run on standard hardware (desktop, PCs) and standard operating systems.
2. **Hardware-based load balancer:** Hardware-based load balancers are dedicated boxes which include Application Specific Integrated Circuits (ASICs) adapted for a particular use. ASICs allows high speed promoting of network traffic and are frequently used for transport-level load balancing because hardware-based load balancing is faster in comparison

to software solution.

Major Examples of Load Balancers –

1. **Direct Routing Requesting Dispatching Technique:** This approach of request dispatching is like to the one implemented in IBM's Net Dispatcher. A real server and load balancer share the virtual IP address. In this, load balancer takes an interface constructed with the virtual IP address that accepts request packets and it directly routes the packet to the selected servers.
2. **Dispatcher-Based Load Balancing Cluster:** A dispatcher does smart load balancing by utilizing server availability, workload, capability and other user-defined criteria to regulate where to send a TCP/IP request. The dispatcher module of a load balancer can split HTTP requests among various nodes in a cluster. The dispatcher splits the load among many servers in a cluster so the services of various nodes seem like a virtual service on an only IP address; consumers interrelate as if it were a solo server, without having an information about the back-end infrastructure.
3. **Linux Virtual Load Balancer:** It is an opensource enhanced load balancing solution used to build extremely scalable and extremely available network services such as HTTP, POP3, FTP, SMTP, media and caching and Voice Over Internet Protocol (VoIP). It is simple and powerful product made for load balancing and fail-over. The load balancer itself is the primary entry point of server cluster systems and can execute Internet Protocol Virtual Server (IPVS), which implements transport-layer load balancing in the Linux kernel also known as Layer-4 switching.

CLOUD ELASTICITY:

Elasticity refers to the ability of a cloud to automatically expand or compress the infrastructural resources on a sudden up and down in the requirement so that the workload can be managed efficiently. This elasticity helps to minimize infrastructural costs. This is not applicable for all kinds of environments, it is helpful to address only those scenarios where the resource requirements fluctuate up and down suddenly for a specific time interval. It is not quite practical to use where persistent resource infrastructure is required to handle the heavy workload. The versatility is vital for mission basic or business basic applications where any split the difference in the exhibition may prompts enormous business misfortune. Thus, flexibility comes into picture where extra assets are provisioned for such application to meet the presentation prerequisites.

It works such a way that when number of client access expands, applications are naturally provisioned the extra figuring, stockpiling and organization assets like central processor, Memory, Stockpiling or transfer speed what's more, when fewer clients are there it will naturally diminish those as per prerequisite.

The Flexibility in cloud is a well-known highlight related with scale-out arrangements (level scaling), which takes into consideration assets to be powerfully added or eliminated when required.

It is for the most part connected with public cloud assets which is generally highlighted in pay-per-use or pay-more only as costs arise administrations.

The Flexibility is the capacity to develop or contract framework assets (like process, capacity or organization) powerfully on a case by case basis to adjust to responsibility changes in the applications in an autonomic way.

It makes make most extreme asset use which bring about reserve funds in foundation costs in general.

Relies upon the climate, flexibility is applied on assets in the framework that isn't restricted to equipment, programming, network, QoS and different arrangements.

The versatility is totally relying upon the climate as now and again it might become negative characteristic where execution of certain applications probably ensured execution.

It is most commonly used in pay-per-use, public cloud services. Where IT managers are willing to pay only for the duration to which they consumed the resources.

Example:

Consider an online shopping site whose transaction workload increases during festive season like Christmas. So for this specific period of time, the resources need a spike up. In order to handle this kind of situation, we can go for a Cloud-Elasticity service rather than Cloud Scalability. As soon as the season goes out, the deployed resources can then be requested for withdrawal.

CLOUD SCALABILITY:

Cloud scalability is used to handle the growing workload where good performance is also needed to work efficiently with software or applications. Scalability is commonly used where the persistent deployment of resources is required to handle the workload statically.

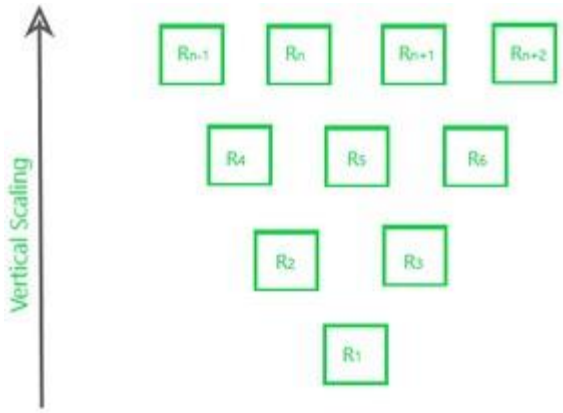
Example:

Consider you are the owner of a company whose database size was small in earlier days but as time passed your business does grow and the size of your database also increases, so in this case you just need to request your cloud service vendor to scale up your database capacity to handle a heavy workload.

It is totally different from what you have read above in Cloud Elasticity. Scalability is used to fulfill the static needs while elasticity is used to fulfill the dynamic need of the organization. Scalability is a similar kind of service provided by the cloud where the customers have to pay-per-use. So, in conclusion, we can say that Scalability is useful where the workload remains high and increases statically.

Types of Scalability:

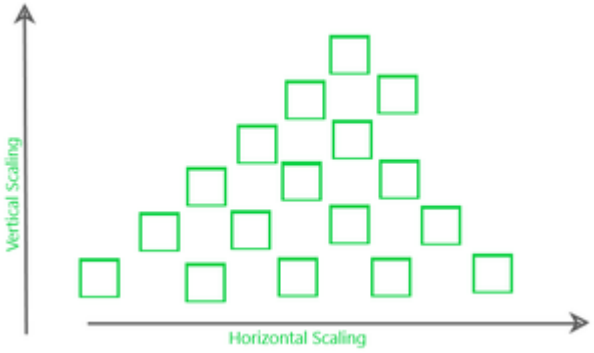
1. Vertical Scalability (Scale-up) – In this type of scalability, we increase the power of existing resources in the working environment in an upward direction.



4.

2. Horizontal Scalability: In this kind of scaling, the resources are added in a horizontal row.

3. Diagonal Scalability – It is a mixture of both Horizontal and Vertical scalability where the resources are added both vertically and horizontally.



5.

6. **Difference Between Cloud Elasticity and Scalability :**

	Cloud Elasticity	**Cloud Scalability**
1	Elasticity is used just to meet the sudden up and down in the workload for a small period of time.	Scalability is used to meet the static increase in the workload.
2	Elasticity is used to meet dynamic changes, where the resources need can increase or decrease.	Scalability is always used to address the increase in workload in an organization.
3	Elasticity is commonly used by small companies whose workload and demand increases only for a specific period of time.	Scalability is used by giant companies whose customer circle persistently grows in order to do the operations efficiently.
4	It is a short term planning and adopted just to deal with an unexpected	Scalability is a long term planning and adopted just to deal with an expected

	Cloud Elasticity	Cloud Scalability
	increase in demand or seasonal demands.	Increase in demand.

REPLICATION

Cloud Replication refers to the process of replicating data from on-premises storage to the cloud, or from one cloud instance to another.

Why do we require replication?

The first and foremost thing is that it makes our system more stable because of node replication. It is good to have replicas of a node in a network due to following reasons:

- If a node stops working, the distributed network will still work fine due to its replicas which will be there. Thus it increases the fault tolerance of the system.
- It also helps in load sharing where loads on a server are shared among different replicas.
- It enhances the availability of the data. If the replicas are created and data is stored near to the consumers, it would be easier and faster to fetch data.

Types of Replication

- Active Replication
- Passive Replication

Active Replication:

- The request of the client goes to all the replicas.
- It is to be made sure that every replica receives the client request in the same order else the system will get inconsistent.
- There is no need for coordination because each copy processes the same request in the same sequence.
- All replicas respond to the client's request.

Advantages:

- It is really simple. The codes in active replication are the same throughout.
- It is transparent.
- Even if a node fails, it will be easily handled by replicas of that node.

Disadvantages:

- It increases resource consumption. The greater the number of replicas, the greater the memory needed.

- It increases the time complexity. If some change is done on one replica it should also be done in all others.

Passive Replication:

- The client request goes to the primary replica, also called the main replica.
- There are more replicas that act as backup for the primary replica.
- Primary replica informs all other backup replicas about any modification done.
- The response is returned to the client by a primary replica.
- Periodically primary replica sends some signal to backup replicas to let them know that it is working perfectly fine.
- In case of failure of a primary replica, a backup replica becomes the primary replica.

Advantages:

- The resource consumption is less as backup servers only come into play when the primary server fails.
- The time complexity of this is also less as there's no need for updating in all the nodes replicas, unlike active replication.

Disadvantages:

- If some failure occurs, the response time is delayed.

MONITORING

Cloud monitoring is the process of reviewing and managing the operational workflow and processes within a cloud infrastructure or asset. It's generally implemented through automated monitoring software that gives central access and control over the cloud infrastructure.

Admins can review the operational status and health of cloud servers and components.

Concerns arise based on the type of cloud structure you have, and your strategy for using it. If you're using a public cloud service, you tend to have limited control and visibility for managing and monitoring the infrastructure. A private cloud, which most large organizations use, provides the internal IT department more control and flexibility, with added consumption benefits.

Regardless of the type of cloud structure your company uses, monitoring is critical to performance and security.

How Cloud Monitoring Works

The cloud has many moving parts, and it's important to ensure everything works together seamlessly to optimize performance. Cloud monitoring primarily includes functions such as:

- **Website monitoring**: Tracking the processes, traffic, availability and resource utilization of cloud-hosted websites
- **Virtual machine monitoring:** Monitoring the virtualization infrastructure and individual virtual machines
- **Database monitoring**: Monitoring processes, queries, availability, and consumption of cloud database resources
- **Virtual network monitoring**: Monitoring virtual network resources, devices, connections, and performance
- **Cloud storage monitoring**: Monitoring storage resources and their processes provisioned to virtual machines, services, databases, and applications

Cloud Monitoring Capabilities

Cloud monitoring makes it easier to identify patterns and discover potential security risks in the infrastructure. Some key capabilities of cloud monitoring include:

- Ability to monitor large volumes of cloud data across many distributed locations
- Gain visibility into application, user, and file behavior to identify potential attacks or compromises
- Continuous monitoring to ensure new and modified files are scanned in real time
- Auditing and reporting capabilities to manage security compliance
- Integrating monitoring tools with a range of cloud service providers

Monitoring Private, Public, and Hybrid Clouds

Cloud monitoring is easier if you operate in a private cloud for reasons we mentioned earlier (control and visibility), as you have access to the systems and software stack. Though monitoring can be more difficult in public or hybrid clouds, application performance monitoring tools (APM) give you visibility into performance behaviors.

A hybrid cloud environment presents unique challenges because data resides in both the private and public cloud. Limitations due to security and compliance can create issues for users accessing data. Admins can solve performance issues by determining what data to store in which cloud as well as what data to asynchronously update. Database synchronization can be a challenge as well, but sharding—partitioning data into smaller, faster and more easily managed parts—helps reduce issues.

Though private cloud gives you more control, you still need to monitor workloads to ensure optimum performance. Without a clear picture of workload and network performance, you can’t justify configuration or architectural changes or

quantify the effectiveness of quality of service implementations or other technologies.

APM tools are helpful in private cloud environments as well, as they work hand-in-hand with existing data monitoring and management and can track performance.

CLOUD COMPUTING SERVICES AND PLATFORM

CLOUD COMPUTING SERVICES

- **Compute services**
- **Storage services**
- **Application services**

COMPUTE SERVICES

What are compute services?

Compute services are also known as Infrastructure-as-a-Service (IaaS). Compute platforms, such as AWS Compute, supply a virtual server instance and storage and APIs that let users migrate workloads to a virtual machine. Users have allocated compute power and can start, stop, access, and configure their computer resources as desired.

How to choose between different AWS Compute Services

Choosing the best AWS infrastructure depends on your application requirements, lifecycle, code size, demand, and computing needs. Take a look at these three examples:

1. If you want to deploy a selection of on-demand instances offering a wide array of different performance benefits within your AWS environment, you would use Amazon Elastic Compute Cloud (EC2).
2. If you want to run Docker-enabled applications packaged as containers across a cluster of EC2 instances, you could use Amazon Elastic Container Service (Amazon ECS).
3. If you want to run your own code using only milliseconds of compute resource in response to event-driven triggers in serverless environment, you could use AWS Lambda.

What are the benefits of AWS compute services?

- AWS Compute services offer the broadest and deepest functionality for compute. Key benefits of using AWS Compute include:
- Right compute for your workloads
- Amazon EC2 (Amazon Elastic Compute Cloud) offers granular control for managing application infrastructure with the choice of processors, storage, and networking. Amazon Elastic Container Services (Amazon ECS) offer choice and flexibility to run containers.

- Built-in security
- AWS offers significantly more security, compliance, and governance services, and key features than the next largest cloud provider. The AWS Nitro System has security built in at the chip level to continuously monitor, protect, and verify the instance hardware.
- Cost optimization
- With AWS compute you pay only for the instance or resource you need, for as long as you use it, without requiring long-term contracts or complex licensing.
- Flexibility
    - AWS provides multiple ways to build, deploy, and get applications to market quickly. For example, Amazon Light sail is an easy-to-use service that offers you everything you need to build an application or website.
    - To determine which AWS Compute service is best suited to grow your business, don't hesitate to Get in Touch with our team of experts or sign-up for a Free AWS Account today.
    - In AWS Compute services, virtual machines are called instances. AWS EC2 provides various instance types with different configurations of CPU, memory, storage, and networking resources so a user can tailor their compute resources to the needs of their application.

There are five types of instances:

General purpose instances

General purpose instances provide a balance of compute, memory and networking resources, and can be used for a variety of diverse workloads. These instances are ideal for applications that use these resources in equal proportions such as web servers and code repositories.

Compute optimized instances

Compute optimized instances are used to run high-performance compute applications that require fast network performance, extensive availability, and high input/output (I/O) operations per second. Scientific and financial modeling and simulation, big data, enterprise data warehousing, and business intelligence are examples of this type of application.

Accelerated computing instances

Accelerated computing instances use hardware accelerators, or co-processors, to perform functions, such as floating point number calculations, graphics processing, or data pattern matching, more efficiently than is possible in software running on CPUs.

Memory optimized instances

Memory optimized instances use high-speed, solid-state drive infrastructure to provide ultra-fast access to data and deliver high performance. These instances are ideal for applications that require more memory and less CPU power, such as open-source databases and real-time big data analytics.

Storage optimized instances

Storage optimized instances are designed for workloads that require high, sequential read and write access to very large data sets on local storage. They are optimized to deliver tens of thousands of low-latency, random I/O operations per second (IOPS) to applications.

What is a container?

Before software is released, it must be tested, packaged, and installed. Software deployment refers to the process of preparing an application for running on a computer system or a device.

Docker is a tool used by developers for deploying software. It provides a standard way to package an application's code and run it on any system. It combines software code and its dependencies inside a container. Containers (or Docker Images) can then run on any platform via a docker engine. Amazon Elastic Container Service (ECS) is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances. This ensures quick, reliable, and consistent deployments, regardless of the environment.

A hospital booking application: an example of Docker

For example, a hospital wants to make an appointment booking application. The end users may use the app on Android, iOS, Windows machine, MacBook, or via the hospital's website. If the code were deployed separately on each platform, it would be challenging to maintain. Instead, Docker could be used to create a single universal container of the booking application. This container can run everywhere, including on computing platforms like AWS.

What is serverless computing with AWS cloud?

Serverless computing refers to the development of applications with externally managed, underlying server infrastructure. Serverless services, like AWS Lambda, come with automatic scaling, built-in high availability, and a pay-for-value billing model.

Serverless computing is a way to describe the services, practices, and strategies that enable software development companies to innovate and respond faster to change. Teams can release applications quickly, get feedback, and improve their software by eliminating operational overheads.

For example, a tech start-up creates an application to search and filter university courses. To launch, the company can go serverless, and focus on refining

user experience and systems. By using fully managed hardware infrastructure, it can invest in marketing instead.

What is elastic load balancing of compute resources?

Load balancing is the process of evenly distributing computing resources and workload in a cloud computing environment. This is done to reduce lag and maintain processing time, even when the application is in high demand. Load balancers can intelligently distribute client requests across multiple application servers that are running in a cloud environment.

Elastic Load Balancing enables users to maximize application performance and reliability. It can automatically distribute incoming application traffic across multiple targets, such as Amazon EC2 instances, containers, IP addresses, AWS lambda functions, and virtual servers. It can handle the varying load of application traffic, reduce cost, and efficiently scale the application up or down to match demand

E-commerce: an example of elastic load balancing

For example, an online e-commerce store runs an application for sorting the best deals of the day. As a compute-intensive application, it uses cloud compute and load balancing to manage demand. This automatically uses additional processing resources on weekends, Christmas, and other seasonal peaks when demand spikes. On other days, it scales down compute when demand slows. Without load balancing, the store would have to pay peak usage rates even on slow days, reducing profit margins.

STORAGE SERVICES

Cloud storage is a virtual locker where we can remotely stash any data. When we upload a file to a cloud-based server like Google Drive, OneDrive, or iCloud that file gets copied over the Internet into a data server that is **cloud-based** actual physical space where companies store files on multiple hard drives. Most companies have hundreds of the ese servers known as ‘server farms’ spanning across multiple locations. So, if our data gets somehow lost we will not lose our data because it will be backed up by another location. This is known as redundancy which keeps our data safe from being lost.

Features of Cloud Storage System

The key features of cloud computing are as follows.

- It has a greater availability of resources.
- Easy maintenance is one of the key benefits of using Cloud computing.
- Cloud computing has a Large Network Access.
- It has an automatic system.
- Security is one of the major components and using cloud computing you can secure all over the networks.

Storage Systems in the Cloud

There are 3 types of storage systems in the Cloud as follows.

- Block-Based Storage System
- File-Based Storage System
- Object-Based Storage System

Let's discuss it one by one as follows.

1. Block-Based Storage System

- Hard drives are block-based storage systems. Your operating system like Windows or Linux actually sees a hard disk drive. So, it sees a drive on which you can create a volume, and then you can partition that volume and format them.
- For example, If a system has 1000 GB of volume, then we can partition it into 800 GB and 200 GB for local C and local D drives respectively.
- Remember with a block-based storage system, your computer would see a drive, and then you can create volumes and partitions.

2. File-Based Storage System

- In this, you are actually connecting through a Network Interface Card (NIC). You are going over a network, and then you can access the network-attached storage server (NAS). NAS devices are file-based storage systems.
- This storage server is another computing device that has another disk in it. It is already created a file system so that it's already formatted its partitions, and it will share its file systems over the network. Here, you can actually map the drive to its network location.
- In this, like the previous one, there is no need to partition and format the volume by the user. It's already done in file-based storage systems. So, the operating system sees a file system that is mapped to a local drive letter.

3. Object-Based Storage System

In this, a user uploads objects using a web browser and uploads an object to a container i.e., Object Storage Container. This uses the HTTP Protocols with the rest of the APIs (for example: GET, PUT, POST, SELECT, DELETE).

- For example, when you connect to any website, you need to download some images, text, or anything that the website contains. For that, it is a code HTTP GET request. If you want to review any product then you can use PUT and POST requests.
- Also, there is no hierarchy of objects in the container. Every file is on the same level in an Object-Based storage system.

Cloud Storage Architecture

Cloud Storage architecture flow is as follows :

- The Cloud Storage Architecture consists of several distributed resources, but still functions as one, either in a cloud architecture of federated or cooperative storage.
- Durable through the manufacture of copies of versions.
- Ultimately, it is usually compatible with data replication advantages.

- Companies just need to pay for the storage they actually use, normally an average of a month's consumption. This does not indicate that cloud storage is less costly, but rather that operating costs are incurred rather than capital expenses.
- Cloud storage companies can cut their energy usage by up to 70 percent, making them a greener company.
- Storage and data security is inherent in the architecture of object storage
- The additional infrastructure, effort, and expense to incorporate accessibility and security can be removed depending on the application.
- Tasks for storage management, such as the procurement of additional storage space, are offloaded to the service provider's obligation.
- It provides users with immediate access to a wide variety of tools and software housed in another organization's infrastructure through a web service interface.
- Very few backups servers are located in different locations across the globe, cloud storage may be used as a natural disaster-proof backup.
- With the WebDAV protocol, cloud storage can be mapped as a local drive

Advantages of Cloud Storage

- **Scalability –** Capacity and storage can be expanded and performance can be enhanced.
- **Flexibility –** Data can be manipulated and scaled according to the rules.
- **Simpler Data Migrations –** As it can add and remove new and old data when required and eliminates disruptive data migrations.
- **Recovery** -In the event of a hard drive failure or other hardware malfunction, you can access your files on the cloud.

Disadvantages of Cloud Storage

- Data centers require electricity and proper internet facility to operate their work, failing which system will not work properly.
- Support for cloud storage isn't the best, especially if you are using a free version of a cloud provider.
- When you use a cloud provider, your data is no longer on your physical storage.
- Cloud-based storage is dependent on having an internet connection. If you are on a slow net work you may have issues accessing your storage.

APPLICATION SERVICES

1. Big Data Analysis

One of the most important applications of cloud computing is its role in extensive data analysis. The extremely large volume of big data makes it impossible to store using traditional data management systems. Due to the unlimited storage capacity of the cloud, businesses can now store and analyze big data to gain valuable business insights.

2. **Testing and Development**

Cloud computing applications provide the easiest approach for testing and development of products. In traditional methods, such an environment would be time-consuming, expensive due to the setting up of IT resources and infrastructure,

and needed manpower. However, with cloud computing, businesses get scalable and flexible cloud services, which they can use for product development, testing, and deployment.

3. Antivirus Applications

With Cloud Computing comes cloud antivirus software which is stored in the cloud from where they monitor viruses and malware in the organization's system and fixes them. Earlier, organizations had to install antivirus software within their system and detect security threats.

4. E-commerce Application

Ecommerce applications in the cloud enable users and e-businesses to respond quickly to emerging opportunities. It offers a new approach to business leaders to make things done with minimum amount and minimal time. They use cloud environments to manage customer data, product data, and other operational systems.

5. Cloud Computing in Education

E-learning, online distance learning programs, and student information portals are some of the key changes brought about by applications of cloud computing in the education sector. In this new learning environment, there's an attractive environment for learning, teaching, experimenting provided to students, teachers, and researchers so they can connect to the cloud of their establishment and access data and information.

6. Online Data Storage

Cloud Computing allows storage and access to data like files, images, audio, and videos on the cloud storage. In this age of big data, storing huge volumes of business data locally requires more and more space and escalating costs. This is where cloud storage comes into play, where businesses can store and access data using multiple devices.

The interface provided is easy to use, convenient, and has the benefits of high speed, scalability, and integrated security.

7. Backup and Recovery

Cloud service providers offer safe storage and backup facility for data and resources on the cloud. In a traditional computing system, data backup is a complex problem, and often, in case of a disaster, data can be permanently lost. But with cloud computing, data can be easily recovered with minimal damage in case of a disaster.

CLOUD COMPUTING PLAT FORM

AmazonWebServices(AWS**)**

AWS provides different wide-ranging clouds IaaS services, which

ranges from virtual compute, storage, and networking to complete computing stacks. AWS is well known for its storage and compute on demand services, named as Elastic Compute Cloud (EC2) and Simple Storage Service (S3). In addition, EC2 and S3, a wide range of services can be leveraged to build virtual computing system including: networking support, caching system, DNS, database support, and others.

GooglAppEngine

Google AppEngine is a scalable runtime environment frequently dedicated to executing web applications. These utilize benefits of the large computing infrastructure of Google to dynamically scale as per the demand. AppEngine offers both a secure execution environment and a collection of which simplifies the development if scalable and high-performance Web applications.

These services include: in-memory caching, scalable data store, job queues, messaging, and corn tasks

MicrosoftAzure

Microsoft Azure is a Cloud operating system and a platform in which user can develop the applications in the cloud. Generally, a scalable runtime environment for web applications and distributed applications is provided. Application in Azure are organized around the fact of roles, which identify a distribution unit for applications and express the application's logic.

Azure provides a set of additional services that complement application execution such as support for storage, networking, caching, content delivery, and others.

Hadoop

Apache Hadoop is an open source framework that is appropriate for processing large data sets on commodity hardware. Hadoop is an implementation of MapReduce, an application programming model which is developed by Google. This model provides two fundamental operations for data processing: map and reduce. Yahoo! Is the sponsor of the Apache Hadoop project, and has put considerable effort in transforming the project to an enterprise-ready cloud computing platform for data processing.

Hadoop is an integral part of the Yahoo! Cloud infrastructure and it supports many business processes of the corporates. Currently, Yahoo! Manges the world's largest Hadoop cluster, which is also available to academic institutions.

Force.comandSalesforce.com

Force.com is a Cloud computing platform at which user can develop social enterprise applications. The platform is the basis of SalesForce.com – a Software-as-a-Service solution for customer relationship management. Force.com allows creating applications by composing ready-to-use blocks: a complete set of components supporting all the activities of an enterprise are available. From the design of the data layout to the definition of business rules and user interface is provided by Force.com as a support. This platform is completely hostel in the Cloud, and provides complete access to its functionalities, and those implemented in the hosted applications through Web services technologies.

DMI COLLEGE OF ENGINEERING

Palanchur, Chennai.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3551 - DISTRIBUTED

SYSTEMS

QUESTION BANK

UNIT 1 INTRODUCTION

2 MARKS:

1. What do you mean by message passing ?

In this model, data is shared by sending and receiving messages between co-operating processes, using system calls. Message passing refers to services performing a simple, one-way transfer operation between two programs.

2. Define distributed program.

Distributed program is composed of a set of "n" asynchronous processes like P1 P2 P3... Pi Pn that communicate by message passing over the communication network.

3. What do you mean by synchronous and asynchronous execution?

Synchronous execution means the first task in a program must finish processing before moving on to executing the next task. Asynchronous execution means a second task can begin executing in parallel, without waiting for an earlier task to finish.

4. List out the features of distributed systems.

Features of distributed systems are heterogeneity, openness, scalability, fault tolerance, transparency and resource sharing.

5. Write down the principles of distributed systems.

Distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

6. State the objectives of resource sharing model.

Ability to use any hardware, software or data anywhere in the system. Resource manager controls access, provides naming scheme and controls concurrency.

7. What are the significant consequences of distributed systems?

a. No global clock: The only communication is by sending messages through a network.

b. Independent failures: The programs may not be able to detect whether the network has failed or has become unusually slow.

c. Concurrency: The capacity of the system to handle shared resources can increased by adding more resources to the network.

8. Define transparency. What are its types?

A distributed system needs to hide the fact that its processes and resources are physically distributed across multiple computers.

9. What is the need of openness in distributed system?

Distributed system must be able to interact with services from other open systems, irrespective of the underlying environment. Systems should conform to well-defined interfaces and should support portability of applications.

10. List any two resources of hardware and software, which can be shared in distributed systems with example.

Hardware resource: Memory cache server and CPU servers do some computation for their clients hence their CPU is a shared resource.

Software resource: File: File servers enable multiple clients to have read/write access to the same files. Database: The content of a database can be usefully shared. There are many techniques that control the concurrent access to a database.

11. List an example of distributed system.

Distributed system examples: Internet, an intranet which is a portion of the internet managed by an organization, mobile and ubiquitous computing.

12. Enlist the design issues and challenges of distributed systems.

Design issues and challenges of distributed systems are heterogeneity openness, security, scalability, failure handling, concurrency and transparency.

13. Define access transparency.

Enables local and remote information objects to be accessed using identical operations.

14. What is replication transparency ?

It enables multiple instances of information objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs. Example: Distributed DBMS.

15. What is the goal of concurrency and failure transparency?

Enables several processes to operate concurrently using shared information objects without interference between them.

Failure transparency: Allows users and applications to complete their tasks despite the failure of other components.

16. Differentiate between buffering and caching.

Cache is made from static ram which is faster than the slower dynamic ram used for a buffer. A cache transparently stores data so that future requests for that data can be served faster. A buffer temporarily stores data while the data is the process of moving from one place to another, ie. the input device to the output device. The buffer is mostly used for input/output processes while the cache is used during reading and writing processes from the disk.

17. What is open distributed system?

Open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.

18. Give the example of relocation transparency?

When mobile users can continue to use their wireless laptops while moving from place to place without ever being disconnected.

19. Describe what is meant by a scalable system?

A system is scalable with respect to either its number of components, size or number and size of administrative domains, if it can grow in one or more of these dimensions without an unacceptable loss of performance.

20. What is the role of middleware in a distributed system?

To enhance the distribution transparency that is missing in network operating systems. In other words, middleware aims at improving the single system view that a distributed system should have.

13 MARKS:

1. Explain the difference between message passing and message sharing.

- Message passing
- Message sharing
- Emulating Message – passing systems on a shared memory systems.

2. Describe about Design issues and challenges in Distributed Computing.

- Challenges from system perspective.
- Challenges.
    - Heterogeneity
    - Openness
    - Security

- Scalability
- Failure handling
- Concurrency
- Transparency

3. Explain about the model of Distributed Computations: A distributed program

- A model of Distributed Execution
  - Casual precedence relation
  - Logical vs Physical concurrency
- Models of Communication Networks

4. What is Global State? Explain about the global state of Distributed Systems.

- Definition
- Requirements of global state
  - Garbage collection
  - Deadlock
  - Termination
  - Distributed debugging

5. Explain the applications of Distributed Computing and Challenges.

- Applications
  - Mobile systems
  - Pervasive computing
    - Intranet
  - Multimedia system
    - Web casting

UNIT 2 LOGICAL TIME AND GLOBAL STATE

2 MARKS:

1. What is meant by asynchronous programming?

Asynchronous programming provides opportunities for a program to continue running other code while waiting for a long-running task to complete.

2. What is meant by group communication in distributed system?

Group communication offers a service whereby a message is sent to a group and then this message is delivered to all members of the group. The sender is not aware of the identities of the receivers.

3. Write application of casual order.

Causal ordering is used for implementing distributed shared memory, fair resource allocation. Other applications are updating replicated data, synchronizing multimedia streams and allocating requests in a fair manner.

4. What is synchronous order?

When all the communication between pairs of processes is by using synchronous send and receives primitives, the resulting order is synchronous order.

5. Define scalar time.

Scalar time is designed by Lamport to synchronize all the events in distributed systems. Time domain is the set of non-negative integers.

6. List the issue related with implementation of logical clocks.

Addressing following issues: Data structures local to every process to represent logical time.

Protocol to update the data structures to ensure the consistency condition.

7. List the properties of scalar time.

Properties of scalar time are consistency, total ordering, event counting and system of scalar clocks is not strongly consistent.

8. What is Rendezvous?

Rendezvous is an architecture for creating multi-user applications. It provides support for managing a multi-user session, for performing fundamental input and output activities and for controlling the degree to which the multiple users either share or do not share both information and control.

9. What is clock tick?

When the counter gets to zero, an interruption is generated and is called one clock tick.

10. What is clock skew?

With n computers, all n crystals will run at slightly different rates, causing the software clocks to gradually get out of sync. This difference in time values is called dock skew.

11. What is clock drift rate.

A clock drift rate is the change in the offset between the clock and a nominal perfect reference clock per unit of time measured by the reference clock.

12. List the name of modes the NTP servers synchronize.

NTP servers synchronize with one another in one of three modes multicast,procedure-call and symmetric mode.

13. **What are the two modes of**

synchronization?

The two modes are:

1. External synchronization: For a synchronization bound D > 0, and for a source S of UTC time, IS(0)-C, (0) T. for i = 1,2 N and for all real times t in L

2 Internal synchronization: For a synchronization bound D>0.(C0)-C < D. for 1.1 1.2 N and for all real times t in 1

14. What is logical clock?

Logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process P, keeps its own logical clock L., which it uses to apply so called Lamport timestamps to events.

15. What is global state of the distributed system?

The global state of the distributed system consists of the local state of each process, together with the messages which are in transit.

16. Write the happens-before relation?

The happens before relation can be observed directly in two situations:

1. If a and b are events in the same process, and a occurs before b then a 'b is true.

2. It is the event of a message being sent by one process, and b is the event of the message being received by another process, then a b is also true.

17. What is need of physical clock?

Lamport's algorithm for logical clock synchronization gives an unambiguous event ordering the tone values assigned to events are not necessarily close to the actual times at which they occur.

In some systems like real-time systems, the actual clock time is important these systems external physical clocks are required.

18. What problem with Lamport's clocks to vector clocks solve?

With Lamport's clocks, you cannot tell whether two events are causally related or concurrent by looking at the timestamps. Just because L(a) L(b) does not mean that a->b Vector clocks allow you to compare two vector timestamps to determine whether the events are concurrent or not.

19. What is vector clock?

Vector clocks are used in a distributed system to determine whether pairs of events are causally related Using vector clocks, timestamps are generated for each event in the system, and their causal relationship is determined by comparing those timestamps.

20. How vector clock timestamps are assigned?

Vector clock timestamps are assigned as follows:

a Events: Every time an event is generated, a process increments its clock and assigns a

timestamp to the event based on its knowledge of all the clocks in the system.

b. Sending messages: When a message is sent the timestamp of the sending event es given to the message.

c. Receiving messages: When a message is receivest, the process updates its knowledge of the system clock states by taking the maximum of each component of the message timestamp and its current knowledge of the system clock states.

13 MARKS:

1. Explain Logical Time.

- Event Ordering
  - Condition of happens before
  - Logical clock condition
- Lamport Timestamp
- Vector Timestamp

2. Discuss about Physical Clock Synchronization: NTP

- Synchronization in a Synchronous System
- Cristian's Method for Synchronizing Clocks
  - Christian's Algorithm
- Berkeley Algorithm
- Network Time Protocol
  - Localized Averaging Distributed Algorithm.
    - Network time protocol
    - Features of NTP

3. Explain the difference between Scalar Time and Vector Time.

- Scalar Time
  - Basic Properties
    - Consistency property
    - Total ordering
    - Event counting
- Vector Time
  - Definition

4. Describe and Explain the Group Communication.

- One to Many Communication
  - Group Management
  - Group addressing
  - Buffered and unbuffered multicast
- Many to One Communication
- Many to Many Communication
  - Message Ordering
    - Absolute ordering
    - Consistent/ Total Ordering

- Causal ordering

5. Explain the global state and snapshot recording algorithm.

- Definition
- System Model
- Consistent Global State

UNIT 3 DISTRIBUTED MUTEX AND DEADLOCK

2 MARKS:

1.Explain the term mutual exclusion.

Asynchronous programming provides opportunities for a program running other code while waiting for a long-running task to complete.

2. What is deadlock?

Deadlock is the problem of multiprogramming system. Deadlock can be defined as the permanent blocking of a set of processes that either complete for system resources.

3. Name the two types of messages used in Ricart-Agrawala's algorithm.

Two type of messages used by Ricart-Agrawala are REQUEST and REPLY and communication channels are assumed to follow FIFO order. Site send a REQUEST message to all other site to get their permission to enter critical section. A site send a REPLY message to other site to give its permission to enter the critical section.

4. What are the conditions for deadlock?

Conditions should hold simultaneously for deadlock to occur are:

a) Mutual exclusion c) Hold and wait

b) No preemption d) Circular wait.

5. What is mutual exclusion?

Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables or a local kernel cannot be used to implement mutual exclusion.

6. **Which are the three basic approaches for implementing distributed mutual exclusion?**

There are three basic approaches for implementing distributed mutual exclusion

1. Token based approach 2. Non-token based approach

3. Quorum based approach

7. What are the requirements of mutual exclusion algorithms?

Requirements of mutual exclusion algorithms are

a. Freedom from deadlocks

b. Freedom from starvation

c Strict fairness

d. Fault tolerance

8. What are the performance metric of mutual exclusion algorithm?

Performances metric are message complexity, synchronization delay, response time and system throughput.

9. What is response time?

The time interval a request waits for its CS execution to be over after its request messages have been sent out.

10. **Which are the criteria for evaluating performance of algorithms for mutual exclusion?**

Criteria for evaluating performance of algorithms for mutual exclusion are:

a. Bandwidth consumed which is proportional to the number of messages sent in each entry and exit operation.

b. Client delay incurred by a process at each entry and exit operation.

c. Throughput of the system.

10. What is the advantage if your server side processing uses threads instead of a single process?

An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running. This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.

11. What is a phantom deadlock?

A deadlock that is detected' but is not really a deadlock is called a phantom deadlock.

12. What is wait for graph?

The state of process-resource interaction in distributed systems can be modeled by a bi-partite directed graph called a resource allocation graph. The nodes of this graph are processes

and resources of a system, and the edges of the graph depict assignments or pending requests. A pending request is represented by a request edge directed from the node of a requesting process to the node of the requested resource.

13. Explain Wait-die" method.

A non-preemptive approach. If a younger process is using the resource, then the older process waits. If an older process is holding the resource, the younger process kills itself. This forces the resource utilization graph to be directed from older to younger processes, making cycles impossible. This algorithm is known as the wait-die algorithm.

14. List the deadlock handling strategies in distributed system.

There are three strategies for handling deadlocks, viz, deadlock prevention, deadlock avoidance, and deadlock detection.

15. What do you mean by deadlock avoidance?

Deadlock avoidance depends on additional information about the long term resource needs of each process. The system must be able to decide whether granting a resource is safe or not and only make the allocation when it is safe. When a process is created, it must declare its maximum claim, i.e. the maximum number of unit resource The resource manager can grant the request if the resources are available.

16. Define deadlock detection in distributed systems.

Deadlock detection requires examination of process-resource interaction for the presence of cyclic wait.

17. What is Chandy-Misra-Haas Algorithm?

A blocked process determines if it is deadlocked by initiating a diffusion computation. processes in its dependent set. If an active process receives a query or reply message, it discards it. all the query messages it has sent out.

18. What is OR Model?

Set of Deadlocked processes, where each process waits to receive messages from other processes in the set.

19. What is AND Model?

Set of deadlocked processes, where each process waits for resource held by another process.

Use AND condition.

The condition for deadlock in a system using the AND condition is the existence of a cycle.

20. Define Deadlock Avoidance.

Decision made dynamically, before allocating a resource, the resulting global system state is checked, if it is safe state then allow for allocation.

Because of the following drawback, deadlock avoidance can be impractical in distributed system.

13 MARKS:

1. What is Lampot's Algorithm and Explain it?

- Definition
- Requesting the critical session
- Conditions for entering CS
- Releasing the CS
- Correctness
- Optimization
- Lamport evalution

2. Explain Ricart – Agarwala's Algorithm.

- Definition
- Algorithm
- Requesting the Critical Session
- Executing the Critical Session
- Releasing the Critical Session

3. Explain Token Based Algorithm

- Definition
- Suzuki – Kasami's Broadcast Algorithm.
    - Major Design Issues
    - Important Data Structures
    - Algorithm
        - Requesting CS
        - Executing CS
        - Releasing CS
        - Theorem: A requesting site enters CS in finite time
        - Performance

4. Discuss the Deadlock Detection in Distributed Systems: Introduction.

- Deadlock
- Necessary Condition
    - Mutual exclusion
    - Hold and wait
    - Circular waiting
    - No preemption

5. Explain Preliminaries: Deadlock Handling Strategies.

- Deadlock Prevention

- First Method
- Second Method
- Third Method

- Dead Avoidance
  - Disadvantage
- Deadlock Detection
  - Principle of operation
  - Resolution
  - Observation

UNIT –4 CONSENSUS AND RECOVERY

2 MARKS:

1. State the use of Rollback recovery.

Restore the system back to a consistent state after a failure

. • Achieve fault tolerance by periodically saving the state of a process during the failure-free execution.

- Treats a distributed system application as communicate over a network.

2. What is consensus in distributed system?

Each process has an initial value and all the correct processes must agree on a single value.

3. Write the purpose of using checkpoints.

Check pointing is most typically used to provide fault tolerance to applications. Check pointing techniques are useful not only for availability, but also for program debugging, process migration, and load balancing.

4. What do you mean by agreement problem in distributed system?

In the agreement problem, to achieve overall system reliability in the presence of a number of faulty processes and single process has the initial value.

5. What is the difference between agreement and consensus problem?

The difference between the agreement problem and the consensus problem is that, in the agreement problem, a single process has the initial value, whereas in the consensus problem, all processes have an initial value.

6. Define recovery.

Recovery refers to restoring a system to its normal operational state. Once a failure has occurred, it is essential that the process where the failure happened recover to a correct state. Fundamental to fault tolerance is the recovery from an error.

7. Explain two types of checkpoints.

1. Tentative: A temporary checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint algorithm.
2. Permanent: A local checkpoint at a process.

8. List drawback of synchronous check pointing.

1. Additional messages must be exchanged to coordinate check pointing

2. Synchronization delays are introduced during normal operations

3. No computational messages can be sent while the check pointing algorithm is in progress.

4. If failure rarely occurs between successive checkpoints, then the checkpoint algorithm places an unnecessary extra load on the system, which can significantly affect performance.

9. How shadow versions are helpful in recovery?

Shadow version uses a map to locate versions of the server's objects in a file called a version store. The map associates the identifiers of the server's objects with the positions of their current versions in the version store. The versions written by each transaction are shadows of the previous committed versions. The transaction status entries and intentions lists are stored separately. When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions. To complete the commit process, the new map replaces the old map.

10. Define fault and failure. What are different approaches to fault-tolerance?

Fault: Anomalous physical condition, eg design errors, manufacturing problems, damage, external disturbances.

Failure of a system occurs when the system does not perform its service in the manner specified.

11. **List the requirements of consensus algorithm to hold for**

execution.

The requirements of consensus algorithm to hold for execution are

1. Termination
2. Agreement and
3. Integrity.

12. **What are the performance aspects of agreement**

protocols?

Following metrics are used

1. Time: No of rounds needed to reach an agreement.

2. Message traffic: Number of messages exchanged to reach an agreement.

3. Storage overhead: Amount of information that needs to stored at processors

during execution of the protocol.

13. **What are the application of agreement**

algorithm?

Applications of agreement algorithms

- Fault-tolerant clock synchronization.
- Distributed systems require physical clocks to synchronized
- Physical clocks have drift problem.
- Agreement protocols may help to reach a common clock value. • Synchronizing distributed clocks:
- At any time, values of clocks of all non-faulty processes must be approximately equal.
- There is a small bound on amount by which the clock of a non-faulty process is

changed during re-synchronization.

14. State Byzantine agreement problem.

In the Byzantine agreement problem, n processors communicate with each other in order to reach an agreement on a binary value b. There are bad processors. that may collaborate with each other in order to prevent an admissible agreement. Each processor has an initial binary value. The agreement must reflect to a certain extent the majority among the initial value.

15. What is local checkpoints?

A process may take a local check point anytime during the execution. The local checkpoints of different processes are not coordinated to form a global consistent checkpoint.

16. What is forced checkpoints?

To guard against the domino effect, a communication induced checkpoint protocol piggybacks protocol-speci?c information to application messages that processes exchange. Each process examines the information and occasionally is forced to take a checkpoint according to the protocol.

17. Explain useless checkpoints.

A useless checkpoint of a process is one that will never be part of a global consistent state. Useless checkpoints are not desirable because they do not contribute to the recovery of the system from failures, but they consume resources and cause performance overhead.

18. What is checkpoint intervals?

A checkpoint interval is the sequence of events between two consecutive checkpoints in the execution of a process.

19. Define orphan messages.

Messages with receive recorded but message send not recorded are called the orphan messages.

20. **What is the basic idea behind task assignment**

approach?

Basic idea:

a. A process has already been split up into pieces called tasks:

b. The amount of computation required by each task and the are known.

c. The cost of processing each task on every node is known.

d. The IPC costs between every pair of tasks is known..

e. Precedence relationships among the taks are known.

f. Reassignment of tasks is not possible. Mention some motivations for replication.

13 MARKS:

1. Explain the Solution to Byzantine Agreement Problem.

- Impossible Scenario
- Lamport – Shostak – Pease Algorithm
    - Example

2. Explain the Consistent Set of Checkpoint.

- Definition
    - Strongly Consistent Set of Checkpoint.
    - Consistent Set of Checkpoint
    - Checkpoint Notation
- Synchronous Checkpoint and Recovery
    - Checkpointing Algorithm
        - Types
    - Synchronous Checkpointing Disadvantages
- The Rollback Recovery Algorithm
    - Phase one
    - Phase two
- Message Types

3. Discuss about the Checkpoint – Based Recovery.

- Uncoordinated Checkpointing

- Direct dependency tracking technique
- Coordinated Checkpointing
  - Blocking Checkpointing
  - Non – Blocking Checkpointing
- Communication – induced Checkpointing

4.Describe the Issues in Failure Recovery.

- Basic Concept
- Recovery
  - System Failure
  - Erroneous System State
  - Error
  - Fault

5.Explain Byzantine Agreement Problem.

- Introduction
  - The Problem
  - Validity
- Consensus Problem
  - Agreement
  - Validity
- Interactive Consistency Problem

UNIT 5 CLOUD COMPUTING

2 MARKS:

1. Explain NIST definition of cloud computing.

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

2. What is cloud service?

Cloud service is any service made available to users on demand via the Internet from a cloud computing provider's servers as opposed to being provided from a company's own on-premises servers.

3. What is public cloud?

Public cloud is built over the Internet and can be accessed by any by user who has paid for the service. Public clouds are owned by service providers and are accessible through a subscription.

4. What is private clouds?

A private cloud is built within the domain of an intranet owned by a single organization. Therefore, it is client owned and managed, and its access is limited to the owning clients and their partners.

5. Explain about virtual machines.

A Virtual Machine (VM) is a software construct that mimics the characteristics. of a physical server. VM is a software program or operating system that not only exhibits the behavior of a separate computer, but is also capable of performing tasks such as running applications and programs like a separate computer.

6. What is NIST definition of laaS?

The ability given to the infrastructure architects to deploy or run any software on the computing resources provided by the service provider. The end users are responsible for managing applications that are running on top of the service provider cloud infrastructure.

7. **Explain characteristics of**

laas.

Characteristics of laaS

1. Resources are provided as a service
2. Allows for dynamic scaling and elasticity.

3. It has a variable cost, usage based pricing model (pay per go and pay per use)

4. It has multi-tenet architecture, includes multiple users on a single piece of hardware.

5. IaaS typically has enterprise grade infrastructure.

8. List the situations where PaaS may not be the best option.

- Integration with on-premise applications.
- Flexibility at the platform level.
- Customization at the infrastructure level.
- Frequent application migration.

9. What is Amazon EC2?

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers and system administrators.

10. List the function of EC2?

EC2 functions:

1. Load variety of operating system.

2. Install custom applications

3. Manage network access permission 4. Run image using as many/few systems as we desire.

11. What is Azure?

Windows Azure is a cloud computing platform and infrastructure, created by Microsoft, for building, deploying and managing applications and services through a global network of Microsoft-managed data centers.

12. What is Azure queues?

Azure queue storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls using HTTP or HTTPS A single queue message can be up to 64 KB in size, and a queue can contain of messages, up to the total capacity limit of a storage account.

13. How virtualization employed in Azure?

Azure is a virtualized infrastructure to which a set of additional enterprise services has been layered on top, including a virtualization service called An AppFabric that creates an application hosting environment. AppFabric is a cloud-enabled version of the NET framework.

14. What is service cloud?

Service cloud refers to the service module in Salesforce.com. It includes accounts, contacts, cases, and solutions. It also encompasses features such as the publi knowledge base, web-to-case, call center, and self-service portal, as well as custome service automation.

15. **What is Google Cloud Storage?**

Google cloud storage allows world-wide storage and retrieval of any amount of data at any time. It can be used for a range of scenarios including serving website content, storing data for archival and disaster recovery, or distributing large data objects to users via direct download.

16. Define Storage Service.

Amazon S3 defines a bucket name as a series of one or more labels, separated by periods, that adhere to the following rules: The bucket name can be between 3 and 63 characters long, and can contain only lower-case characters, numbers, periods, and dashes.

17. What Scalability and Elasticity?

Scalability is the ability of a system or network to handle increased load or usage. At the same time, elasticity is the ability to automatically expand and contract resources to meet demand.

Cloud elasticity is a system's ability to manage available resources according to the current workload requirements dynamically. This is a vital feature of a system infrastructure. It comes in handy when the system is expected to experience sudden spikes of user activity and, as a result, a drastic increase in workload demand.

18. Define Load Balancing.

Load balancing can be defined as the process of task distribution among multiple computers, processes, disk, or other resources in order to get optimal resource utilization and to reduce the computation time.

Load balancing is an important means to achieve effective resource sharing and utilization.

19. Define Pros and Cons of Virtualization.

Pros:

1. Data center and energy-efficiency savings: As companies reduce the size of their hardware and server footprint, they lower their energy consumption.

2. Operational expenditure savings: Once servers are virtualized, your IT staff can greatly reduce the ongoing administration and management of manual work.

3. Reduced costs: It reduced cost of IT infrastructure. 4. Data does not leak across virtual machine.

5. Virtual machine is completely isolated from host machine and other virtual machine.

6. Simplifies resource management by pooling and sharing resources.

7. Significantly reduce downtime.

8. Improved performance of IT resources.

Cons:

1. Not all hardware or software can be virtualized.

2. Not all servers are applications are specifically designed to be virtualization-friendly.

20. Define Para-Virtualization.

Paravirtualization is a type of virtualization in which a guest operating system (OS) is recompiled, installed inside a virtual machine (VM), and operated on top of a hypervisor program running on the host OS.

• Para-virtualization refers to communication between the guest OS and the hypervisor to improve performance and efficiency.

• Para-virtualization involves modifying the OS kernel to replace non-virtualizable instructions with hyper-calls that communicate directly with the virtualization layer hypervisor.

13 MARKS:

1. Explain Cloud Deployment Models

- Public Cloud
  - Benefits
  - Risks
- Private Cloud
  - Benefits
  - Risks
- Community Cloud
- Hybrid Cloud
  - Benefits
  - Risks
- Difference between public and private Cloud

2. Explain Cloud Service Models

- Software as a Service(SaaS)
  - Characteristics
  - Benefits
- Platform as a Service(PaaS)
  - Characteristics
  - Benefits
- Infrastructure as a Service(IaaS)
  - Types
    - Physical Server
    - Dedicated Virtual Server
    - Shared Virtual Server
  - Advantage

3. Discuss about Virtualization.

- Hypervisor
- Para – Virtualization
  - Problems
- Full – Virtualization

- Host Based Virtualization
- Pros and Cons of Virtualization

4. Explain Cloud Services and Platforms: Compute Services.

- Amazon Elastic Compute Cloud
  - Launching an EC2 instance
  - Stop instances
- Windows Azure

5. Explain the Application Services of Cloud.

- Application Framework and Runtime: Google App Engine
  - Major feature of Google App Engine
  - Key feature of GAE Programming mode using Java and Python
    - Python
    - Java
  - Queuing Service: Amazon Simple Queue Service
    - Example.

Reg. No. : | | | | | | | | | | | | |

Question Paper Code : 90422

B.E./B.Tech. DEGREE EXAMINATIONS, NOVEMBER/DECEMBER 2022.

Sixth Semester

Computer Science and Engineering

CS 8603 – DISTRIBUTED SYSTEMS

(Regulations 2017)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What do you mean by Message Passing?

2. Define Distributed Program.

3. What do you mean by Synchronous and Asynchronous Execution?

4. What is meant by asynchronous programming?

5. Explain the term mutual exclusion.

6. What is Deadlock?

7. State the use of Rollback Recovery.

8. What is Consensus in distributed System?

9. What do you understand by two lines Peer-to-Peer computing?

10. Define Data Indexing.

PART B — (5 × 13 = 65 marks)

11. (a) Illustrate the difference between Message Passing and Shared Memory Process Communication Model.

Or

(b) Explain the types of group communications used in Distributed System.

12. (a) What are the four different types of ordering the messages? Explain.

Or

(b) Elucidate on the Total and Casual Order in Distributed System with a neat diagram.

13. (a) Explain Ricart Agrawala Algorithm with an example.

Or

(b) Name and explain the different types of deaklock models in Distributed system with the commonly used strategies to handle deadlocks with a neat diagram.

14. (a) Illustrate the different types of failures in distributed systems and explain how to prevent them.

Or

(b) Illustrate briefly the two kinds of checkpoints for checkpoint algorithm.

15. (a) Explain the different types of Overlay Networks with its advantages and disadvantages.

Or

(b) Critically examine the different types of Distributed Shared Memory with its advantages.

PART C — (1 × 15 = 15 marks)

16. (a) Design the procedure for causality in a synchronous execution with a suitable example.

Or

(b) Analyse Suzuki-Kasami's Broadcast Algorithm for Mutual Exclusion in Distributed system.

———————

Reg. No. : | | | | | | | | | | | | |

Question Paper Code : 70447

B.E./B.Tech. DEGREE EXAMINATIONS, NOVEMBER/DECEMBER 2023.

Sixth Semester

Computer Science and Engineering

CS 8603 — DISTRIBUTED SYSTEMS

(Common to : Artificial Intelligence and Data Science)

(Regulations 2017)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. What do you mean by Shared Memory?
2. Differentiate scalar time and vector time.
3. What do you mean by Asynchronous Execution?
4. Define synchronous programming.
5. Explain the term mutual exclusion.
6. What is Deadlock detection?
7. What do you mean by check point? What is its use?
8. What is Consensus in distributed System?
9. Define Peer-to-Peer computing. Give its advantage.
10. What is Distributed Shared Memory?

PART B — (5 × 13 = 65 marks)

11. (a) Illustrate Message Passing and Shared Memory Process Communication Model.

Or

(b) Elucidate the types of group communications used in Distributed Computing System.

12. (a) What is Asynchronous Execution with synchronous communication? Explain.

Or

(b) Illustrate Total & Casual Order in Distributed System with a neat diagram.

13. (a) Demonstrate Maekawa's Algorithm with an example.

Or

(b) Illustrate the different types of Deadlock models in Distributed System with the commonly used strategies to handle deadlocks with a neat diagram.

14. (a) Demonstrate the different types of failures in distributed Systems and elucidate in what way to prevent them.

Or

(b) Illustrate about the Agreement in synchronous systems with failures.

15. (a) Elucidate the diverse kinds of Overlay Networks with its advantages and disadvantages.

Or

(b) Enlighten the different types of Memory consistency models with its advantages.

PART C — ($1 \times 15 = 15$ marks)

16. (a) Elucidate synchronous and Asynchronous Executions with a neat schematic block diagram. Explain causality in a synchronous execution with a suitable example.

Or

(b) Illustrate Suzuki-Kasami's Broadcast Algorithm for Mutual Exclusion in Distributed System.

———————

Reg. No. : | | | | | | | | | | | | |

Question Paper Code : 20874

B.E./B.Tech. DEGREE EXAMINATIONS, NOVEMBER/DECEMBER 2023.

Fifth Semester

Computer Science and Design

CS 3551 – DISTRIBUTED COMPUTING

(Common to : Computer Science and Engineering/ Computer Science and Engineering (Artificial Intelligence and Machine Learning)/ Computer Science and Engineering (Cyber Security)/ Computer and Communication Engineering/ Artificial Intelligence and Data Science and Information Technology)

(Regulations – 2021)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. State the differences from synchronous and asynchronous communication along with the buffering strategy adapted in both.

2. List the role of shared memory systems.

3. How do you define the scalar time and the vector time?

4. If the two events e_1 and e_2 of two different processes occur at same time, independently. What kind of relationship exist between e_1 and e_2?

5. What are the various deadlock detection in distributed systems? State a common factor in detecting the deadlock.

6. How beneficial is the Chandy-Misra-Haas algorithm in the AND model and OR model?

7. What is a checkpoint in the recovery system? How is it useful?

8. State the issues in failure recovery. State the means to identify the failure at an early stage.

9. What is virtualization? Why is it required in the larger organization?

10. State the relation of compute service with storage service in a cloud environment.

PART B — (5 × 13 = 65 marks)

11. (a) Explain message passing systems and discuss on the message oriented middleware and its types. Also explain their functionality in distributed computing.

Or

(b) Analyze the concepts of heterogeneity, openness, security, scalability impact of the distributed systems. How is the standardization of them make them effective?

12. (a) Give a real time scenario where FIFO message queue is used. Write and describe the snapshot algorithms for FIFO channels.

Or

(b) Describe the physical clock synchronization and logical clock synchronization and explain the framework of a system of logical clock.

13. (a) State the Lamport's algorithm and its use and also the limitations and benefits. Compare this with any two token based algorithms.

Or

(b) Explain the system models with its preliminaries and how is it characterized in the models of deadlock.

14. (a) Explain the facts associated with agreement in failure-free systems of both synchronous and asynchronous systems.

Or

(b) What are the different check pointing based recovery systems available? Explain the coordinated check pointing algorithm with illustration.

15. (a) A company like to have an advanced collaboration services like video, chat and web conferences for their employees, but their system does not support any of the IT resources due to insufficient infrastructure. If they could leverage cloud computing technology in their system, suggest a suitable cloud type with proper justification. List the characteristic of cloud computing.

Or

(b) Explain the different cloud services and the platform that support these services. Give an example for application service adapted by any organization.

PART C — (1 × 15 = 15 marks)

16. (a) Consider an ABC IT company wanted to provide services like scientific converter, data converter, currency converter and some of the business logic as a server side component to the developer to tailor make the application. Explain the various adaptable technologies to implement in an distributed environment. State its merit and demerits.

Or

(b) An organization has planned to implement a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. State the best distributed architecture to deploy and deliver the system. Also explain the suitable system model to avoid failures and adapt a recovery system for the same in case of unavoidable failures.

———————

CSE

Reg. No. : | | | | | | | | | | | | |

Question Paper Code : 50435

B.E./B.Tech. DEGREE EXAMINATIONS, APRIL/MAY 2023.

Sixth Semester

Computer Science and Engineering

CS 8603 – DISTRIBUTED SYSTEMS

(Common to: Artificial Intelligence and Data Science)

(Regulations 2017)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Define distributed system.
2. What are the main differences between a parallel system and a distributed system?
3. Define asynchronous execution.
4. What are the two phases in obtaining a global snapshot?
5. Identify the three basic approaches for implementing distributed mutual exclusion.
6. Outline the wait-for graph (WFG).
7. What do you mean by checkpoint?
8. What are the difference between the agreement problem and the consensus problem?
9. List the advantages of unstructured overlays.
10. What is memory coherence?

PART B — (5 × 13 = 65 marks)

11. (a) Tabulate the Interaction of the software components at each processor in the distributed system.

Or

(b) Examine in brief about the two basic models of process communications synchronous and asynchronous.

12. (a) Discuss the operation of Privilege-based algorithms with a neat diagram.

Or

(b) Identify how individual local checkpoints can be combined with those from other processes to form global snapshots that are consistent.

13. (a) Recognize a distributed mutual exclusion algorithm as an illustration of the clock synchronization scheme.

Or

(b) Analyze the Correctness criteria of the deadlock detection algorithm.

14. (a) Indicate the several interesting features of the Manivannan-Singhal quasi-synchronous check pointing algorithm.

Or

(b) Compare the results and lower bounds on agreement on solving the consensus problem under different assumptions.

15. (a) Comprehend the classification of data indexing mechanisms in a P2P network.

Or

(b) List the seven advantages of distributed shared memory.

PART C — (1 × 15 = 15 marks)

16. (a) Discriminate the Lamport's bakery algorithm in detail by providing the same algorithm.

Or

(b) Infer the use of causal order in updating replicas of a data item in the communication system.

———————