## UNIT I
## INTRODUCTION

Introduction: Definition –Relation to computer system components –Motivation – Relation to parallel systems – 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.

**1. Explain in detail about distributed systems. (or) Tabulate the Interaction of the software components at each processor in the distributed system. (APR/MAY 2023)**

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. Distributed systems have been in existence since the start of the universe. From a school of fish to a flock of birds and entire ecosystems of microorganisms, there is communication among mobile intelligent agents in nature. With the widespread proliferation of the Internet and the emerging global village, the notion of distributed computing systems as a useful and widely deployed tool is becoming a reality. For computing systems, a distributed system has been characterized in one of several ways:

- A collection of computers that do not share common memory or a common physical clock, that communicate by a message passing over a communication network, and where each computer has its own memory and runs its own operating system.
- A collection of independent computers that appears to the users of the system as a single coherent computer
- A term that describes a wide range of computers, from weakly coupled systems such as wide-area networks, to strongly coupled systems such as local area networks, to very strongly coupled systems such as multiprocessor systems.

A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features:

- **No common physical clock** This is an important assumption because it introduces the element of "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.
- **shared memory** This is a key feature that requires message-passing for communication. This feature implies the absence of the common physical clock.
- **Geographical separation** The geographically wider apart that the processors are, the more representative is the system of a distributed system.
- **Autonomy and heterogeneity** The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system.

**Relation to computer system components**

- A typical distributed system is shown in Figure 1.1. Each computer has a memory-processing unit and the computers are connected by a communication network.
- Figure 1.2 shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning.
- The 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. An execution is also sometimes termed a *computation* or a *run*.
- The distributed system uses a layered architecture to break down the com- plexity of system design. The middleware is the distributed software that

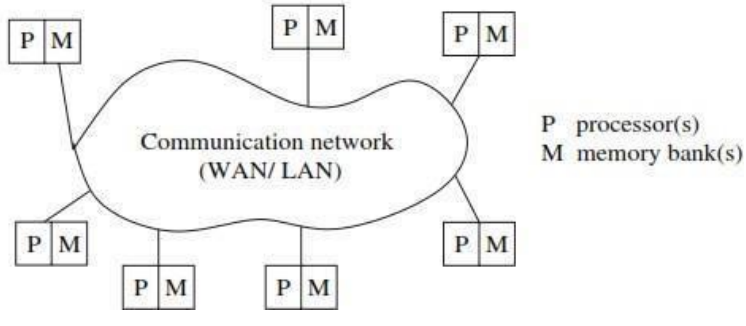



**Figure 1.1** A distributed system connects processors by a communication network.

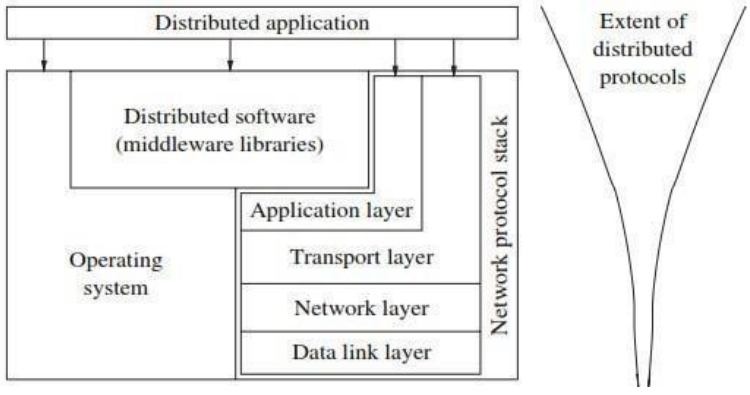



**Figure 1.2** Interaction of the software components at each processor.

drives the distributed system, while providing transparency of heterogeneity at the platform level.

Figure 1.2 schematically shows the interaction of this software with these system components at each processor.

- Here we assume that the middleware layer does not contain the traditional application layer functions of the network protocol stack, such as *http*, *mail*, *ftp*, and *telnet*.
- Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program code.
- There exist several libraries to choose from to invoke primitives for the more common func- tions – such as reliable and ordered multicasting – of the middleware layer.
- There are several standards such as Object Management Group's (OMG) common object request broker architecture (CORBA), and the remote procedure call (RPC) mechanism.
- The RPC mechanism conceptually works like a local procedure call, with the difference that the procedure code may reside on a remote machine, and the RPC software sends a message across the network to invoke the remote procedure.
- It then awaits a reply, after which the procedure call completes from the perspective of the program that invoked it.
- Currently deployed commercial versions of middleware often use CORBA, DCOM (distributed component object model), Java, and RMI (remote method invocation) technologies.

- The message-passing interface (MPI) developed in the research community is an example of an interface for various communication functions.

**2. Discuss in detail about motivation.Motivation**

The motivation for using a distributed system is some or all of the following requirements:

1. **Inherently distributed computations** In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed.
2. **Resource sharing** Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites because it is often neither practical nor cost-effective. Further, they cannot be placed at a single site because access to that site might prove to be a bottleneck.
3. **Access to geographically remote data and resources** In many scenarios, the data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated.
4. It is therefore stored at a central server which can be queried by branch offices. Similarly, special resources such as supercomputers exist only in certain locations, and to access such supercomputers, users need to log in remotely.
5. **Enhanced reliability** A distributed system has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions.
6. The reality that geographically distributed resources are not likely to crash/malfunction at the same time under normal circumstances.
7. Reliability entails several aspects: availability, i.e., the resource should be accessible at all times; integrity, i.e., the value/state of the resource should be correct.
8. **Increased performance/cost ratio** By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased.
9. Although higher throughput has not necessarily been the main objective behind using a distributed system, nevertheless, any task can be partitioned across the various computers in the distributed system.
10. Such a configuration provides a better performance/cost ratio than using special parallel machines. This is particularly true of the NOW configuration.

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

11. **Scalability** As the processors are usually connected by a wide-area net- work, adding more processors does not pose a direct bottleneck for the communication network.

12. **Modularity and incremental expandability** Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algo- rithms. Similarly, existing processors may be easily replaced by other processors.

3. **Explain iin detail about parallel multiprocessor or multicomputer systems. (or) How do you classify a parallel system and brief them? (Nov/Dec 2020/Apr/May 2021)**

**Relation to parallel multiprocessor/multicomputer systems**

The characteristics of a distributed system were identified above. A typical distributed system would look as shown in Figure 1.1. we first examine the architecture of parallel systems, and then examine some well-known taxonomies for multiprocessor/multicomputer systems.

**Characteristics of parallel systems**

A parallel system may be broadly classified as belonging to one of three types:

- A *multiprocessor system* is a parallel system in which the multiple processors have *direct access to shared memory* which forms a common address space. The architecture is shown in Figure 1.3(a). Such processors usually do not have a common clock.
- A multiprocessor system *usually* corresponds to a uniform memory access (UMA) architecture in which the access latency, i.e., waiting time, to complete an access to any memory location from any processor is the same.
- The processors are in very close physical proximity and are connected by an interconnection network.
- Interprocess communication across processors is traditionally through read and write operations on the shared memory, although the use of message-passing primitives such as those provided by

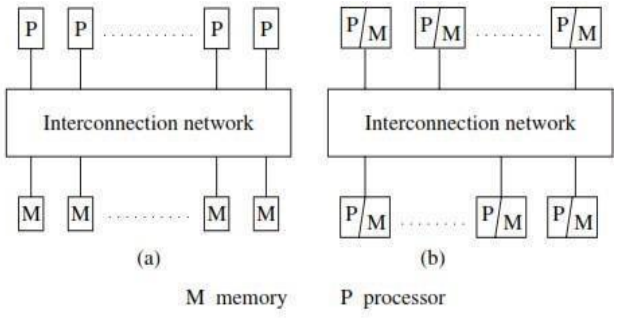



**Figure 1.3** Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.

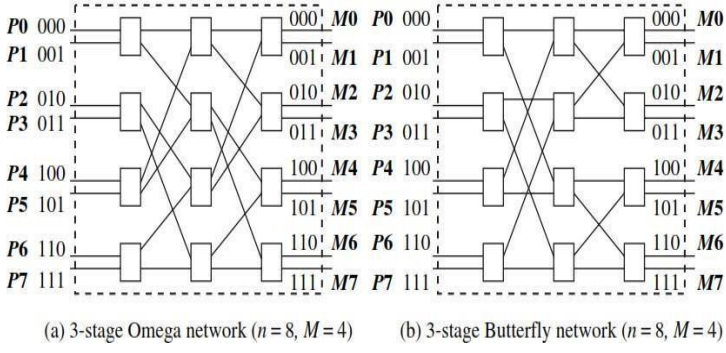



**Figure 1.4** Interconnection networks for shared memory multiprocessor systems. (a) Omega network [4] for $n = 8$ processors $P0$–$P7$ and memory banks $M0$–$M7$. (b) Butterfly network [10] for $n = 8$ processors $P0$–$P7$ and memory banks $M0$–$M7$.

the MPI, is also possible (using emulation on the shared memory).

- All the processors usually run the same operating system, and both the hardware and software are very tightly coupled.
- The processors are usually of the same type, and are housed within the same box/container with a shared memory.
- The interconnection network to access the memory may be a bus, although for greater efficiency, it is usually a *multistage switch* with a symmetric and regular design.
- Figure 1.4 shows two popular interconnection networks – the Omega

network and the Butterfly network, each of which is a multi-stage network formed of 2 ×2 switching elements.

- Each 2 × 2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire. In a single step, however, only one data unit can be sent on an output wire.
- So if the data from both the input wires is to be routed to the same output wire in a single step, there is a collision. Various techniques such as buffering or more elaborate interconnection designs can address collisions.
- Each 2 × 2 switch is represented as a rectangle in the figure. Furthermore, a *n*-input and *n*-output network uses *log n* stages and *log n* bits for addressing.
- Routing in the 2 × 2 switch at stage *k* uses only the *k*th bit, and hence can be done at clock speed in hardware.
- The multi-stage networks can be constructed recursively, and the interconnection pattern between any two stages can be expressed using an iterative or a recursive generating function.

- **Omega interconnection function**

  The Omega network which connects *n* processors to *n* memory units has $n/2 \log_2 n$ switching elements of size 2 × 2 arranged in $\log_2 n$ stages.

Between each pair of adjacent stages of the Omega network, a link exists between output *i* of a stage and the input *j* to the next stage according to the following *perfect shuffle* pattern which is a left-rotation operation on the binary representation of *i* to get *j*. The iterative generation function is as follows:

$$j = \begin{cases} 2i, & \text{for } 0 \le i \le n/2-1, \\ 2i+1-n, & \text{for } n/2 \le i \le n-1. \end{cases}$$

- Consider any stage of switches. Informally, the upper (lower) input lines for each switch come in sequential order from the upper (lower) half of the switches in the earlier stage.
- With respect to the Omega network in Figure 1.4(a), $n = 8$.
- Hence, for any stage, for the outputs $i$, where $0 \leq i \leq 3$, the output $i$ is connected to input $2i$ of the next stage.
- For $4 \leq i \leq 7$, the output $i$ of any stage is connected to input $2i + 1 - n$ of the next stage.
- **Omega routing function** The routing function from input line $i$ to output line $j$ considers only $j$ and the stage number $s$, where $s \in [0, log2n - 1]$.
- In a stage $s$ switch, if the $s + 1$th MSB (most significant bit) of $j$ is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

**Butterfly interconnection function**

Unlike the Omega network, the generation of the interconnection pattern between a pair of adjacent stages depends not only on $n$ but also on the stage number $s$.

The recursive expression is as follows.

Let there be $M = n/2$ switches per stage, and let a switch be denoted bythe tuple $(x, s)$, where $x \in [0, M - 1]$ and stage $s \in [0, log2n - 1]$. The two outgoing edges from any switch $(x, s)$ are as follows. There is an edge from switch $(x, s)$ to switch $(y, s + 1)$ if (i) $x = y$ or (ii) $x$ XOR $y$ has exactly one 1 bit, which is in the $(s + 1)^{th}$ MSB. For stage $s$, apply the rule above for $M/2^s$ switches.

**Butterfly routing function**

- In a stage $s$ switch, if the $s + 1$th MSB of $j$ is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.
- Observe that for the Butterfly and the Omega networks, the paths from the different inputs to any one output form a spanning tree.
- This implies that collisions will occur when data is destined to the same output line. However, the advantage is that data can be combined at the switches if the application semantics (e.g., summation of numbers) are known.

A *multicomputer parallel system* is a parallel system in which the multiple processors *do not have direct access to shared memory.*

The memory of the multiple processors may or may not form a common address space. Such computers usually do not have a common clock.

- The processors are in close physical proximity and are usually very tightly coupled (homogenous hardware and software), and connected by an interconnection network.
- The processors communicate either via a com- mon address space or via message-passing.
- A multicomputer system that has a common address space *usually* corresponds to a non-uniform memory access (NUMA) architecture in which the latency to access various shared memory locations from the different processors varies.
- Examples of parallel multicomputers are: the NYU Ultracomputer and the Sequent shared memory machines, the CM* Connection machine and processors configured in regular and symmetrical topologies such as an array or mesh, ring, torus, cube, and hypercube (message-passing machines).
- The regular and symmetrical topologies have interesting mathematical properties that enable very easy routing and provide many rich features such as alternate routing.
- Figure 1.5(a) shows a wrap-around $4 \times 4$ mesh. For a $k \times k$ mesh which will contain $k^2$ processors, the maximum path length between any two processors is $2(k/2 - 1)$.
- Routing can be done along the Manhattan grid. Figure 1.5(b) shows a four-dimensional hypercube. A $k$-dimensional hyper- cube has $2^k$ processor-and-memory units .
- Each such unit is a node in the hypercube, and has a unique $k$-bit label. Each of the $k$ dimensions is associated with a bit position in the label.
- The labels of any two adjacent nodes are identical except for the bit position corresponding to the dimension in which the two nodes differ.
- Thus, the processors are labelled such that the shortest path between any two processors is the *Hamming distance* (defined as the number of bit positions in which the two equal sized bit strings differ) between the processor labels. This is clearly bounded by $k$.

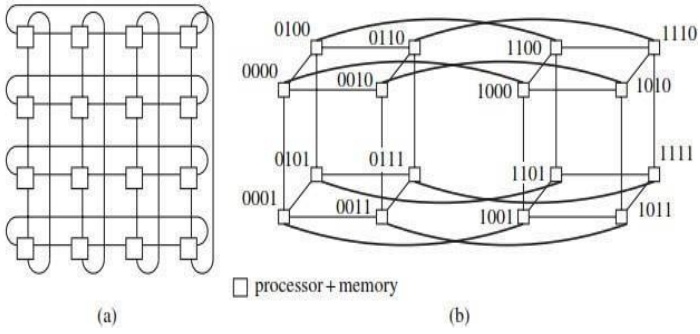



**Figure 1.5** Some popular topologies for multicomputer shared-memory machines. (a) Wrap-around 2D-mesh, also known as torus. (b) Hypercube of dimension 4.

**Example** Nodes 0101 and 1100 have a Hamming distance of 2. The shortest path between them has length 2.

- Routing in the hypercube is done hop-by-hop. At any hop, the message can be sent along any dimension corresponding to the bit position in which the current node's address and the destination address differ.
- The 4D hypercube shown in the figure is formed by connecting the corresponding edges of two 3D hypercubes (corresponding to the left and right "cubes" in the figure) along the fourth dimension; the labels of the 4D hypercube are formed by prepending a "0" to the labels of the left 3D hypercube and prepending a "1" to the labels of the right 3D hypercube.
- This can be extended to construct hypercubes of higher dimensions. Observe that there are multiple routes between any pair of nodes, which provides fault-tolerance as well as a congestion control mechanism.
- The hypercube and its variant topologies have very interesting mathematical properties with implications for routing and fault-tolerance.

1. *Array processors* belong to a class of parallel computers that are physically co- located, are very tightly coupled, and have a common system clock (but may not share memory and communicate by passing data using messages).

2. Array processors and systolic arrays that perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and image processing belong to this category.

3. These applications usually involve a large number of iterations on the data. This class of parallel systems has a very niche market.

The distinction between UMA multiprocessors on the one hand, and NUMA and message-passing multicomputers.

It is important because the algorithm design and data and task partitioning among the processors must account for the variable and unpredictable latencies in accessing mem- ory/communication.

As compared to UMA systems and array processors, NUMA and message-passing multicomputer systems are less suitable when the degree of granularity of accessing shared data and communication is very fine.

**4. Explain in detail about flynn's taxonomy.**

**Flynn's taxonomy**

Flynn identified four processing modes, based on whether the processors execute the same or different instruction streams at the same time, and whether or not the processors processed the same (identical) data at the same time. It is instructive to examine this classification to understand the range of options used for configuring systems:

**Single instruction stream, single data stream (SISD)**
This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.
**Single instruction stream, multiple data stream (SIMD)**
This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items.
Recent SIMD architectures include co-processing units such as the MMX units in Intel processors (e.g., Pentium with the streaming SIMD extensions (SSE) options) and DSP chips such as the Sharc.
**Multiple instruction stream, single data stream (MISD)**
This mode corresponds to the execution of different operations in parallel on the same data. This is a specialized mode of operation with limited but niche applications, e.g., visualization.

**Multiple instruction stream, multiple data stream (MIMD)**
In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems.

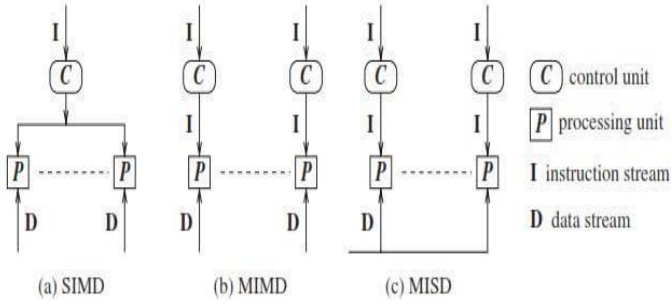



**Figure 1.6** Flynn's taxonomy of SIMD, MIMD, and MISD architectures for multiprocessor/multicomputer systems.

**5. Write a note on coupling, parallelism, concurrency and granularity.**

**Coupling, parallelism, concurrency, and granularity**

**Coupling**

- The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules.
- When the degree of coupling is high (low), the mod- ules are said to be tightly (loosely) coupled. SIMD and MISD architectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream.
- Here we briefly examine various MIMD architectures in terms of coupling:
- Tightly coupled multiprocessors (with UMA shared memory). These may be either switch-based (e.g., NYU Ultracomputer, RP3) or bus-based (e.g., Sequent, Encore).
- Tightly coupled multiprocessors (with NUMA shared memory or that communicate by message passing). Examples are the SGI Origin 2000 and the Sun Ultra HPC servers (that communicate via NUMA shared memory), and the hypercube and the torus (that communicate by message passing).
- Loosely coupled multicomputers (without shared memory) physically co-located. These may be bus-based (e.g., NOW connected by a LAN or Myrinet card) or using a more general communication network, and the processors may be heterogenous.

**Parallelism or speedup of a program on a specific system**

- This is a measure of the relative speedup of a specific program, on a given machine. The speedup depends on the number of processors and the mapping of the code to the processors.
- It is expressed as the ratio of the time *T(1)* with a single processor, to the time *T(n)* with *n* processors.

**Parallelism within a parallel/distributed program**

- This is an aggregate measure of the percentage of time that all the processors are executing CPU instructions.
- It is opposed to waiting for communication (either via shared memory or message-passing) operations to complete.

**Concurrency of a program**

- This is a broader term that means roughly the same as parallelism of a program, but is used in the context of distributed programs.
- The *parallelism/concurrency* in a parallel/distributed program can be measured by the ratio of the number of local (non-communication and non-shared memory access) operations to the total number of operations, including the communication or shared memory access operations.

**Granularity of a program**

- The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as *granularity*.
- If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions.
- Compared to the number of times the processors communicate either via shared memory or message- passing and wait to get synchronized with the other processors.
- Programs with fine-grained parallelism are best suited for tightly coupled systems. These typically include SIMD and MISD architectures, tightly coupled MIMD multiprocessors (that have shared memory).
- And loosely coupled multicomputers (without shared memory) that are physically colocated. If programs with fine-grained parallelism were run over loosely coupled multiprocessors.

**6. Discuss in detail about Primitives for distributed communication. (or) Compare Synchronous versus Asynchronous execution. (Nov/Dec 2020/Apr/May 2021)**

**Blocking/non-blocking, synchronous / asynchronous primitives**

- Message send and message 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.
- Similarly, a *Receive* primitive has at least two parameters – the source from which the data is to be received (this could be a wildcard), and the user buffer into which the data is to be received.
- There are two ways of sending data when the *Send* primitive is invoked – the buffered option and the unbuffered option.
- The *buffered option* which is the standard option copies the data from the user buffer to the kernel buffer.
- The data later gets copied from the kernel buffer onto the network. In the *unbuffered option*, the data gets copied directly from the user buffer onto the network.
- For the *Receive* primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.
- 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 invoking processor learns that the other corresponding *Receive* primitive has also been invoked and that the receive operation has 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 has been copied out of the user-specified buffer.
- It does not make sense to define asynchronous *Receive* primitives.

**Blocking primitives,** A primitive is *blocking* if control returns to the invoking process after the processing for the primitive (whether in synchronous or

asynchronous mode) completes.

- **Non-blocking primitives** A primitive is *non-blocking* if control returns back to the invoking process immediately after invocation, even though the operation has not completed.
- For a non-blocking *Send*, control returns to the process even before the data is copied out of the user buffer. For a non-blocking *Receive*, control returns to the process even before the data may have arrived from the sender.
- For non-blocking primitives, a return parameter on the primitive call returns a system-generated *handle* which can be later used to check the status of completion of the call.
- The process can check for the completion of the call in two ways. First, it can keep checking (in a loop or periodically) if the handle has been flagged or *posted*. Second, it can issue a *Wait* with a list of handles as parameters.
- The *Wait* call usually blocks until one of the parameter handles is posted. After issuing the primitive in non-blocking mode, the process has done whatever actions it could. And now needs to know the status of completion of the call, therefore using a blocking *Wait()* call is usual programming practice.

**Figure 1.7** A non-blocking *send* primitive. When the *Wait* call returns, at least one of its parameters is posted.

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

- If at the time that *Wait()* is issued, the processing for the primitive (whether synchronous or asynchronous) has completed, the *Wait* returns immediately.
- The completion of the processing of the primitive is detectable by checking the value of *handlek*.
- 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.

- There are therefore four versions of the *Send* primitive – synchronous blocking, synchronous non-blocking, asynchronous blocking, and asynchronous non-blocking.
- For the *Receive* primitive, there are the blocking synchronous and non-blocking synchronous versions. These versions of the primitives are illustrated in Figure 1.8 using a timing diagram. Here, three-time lines are

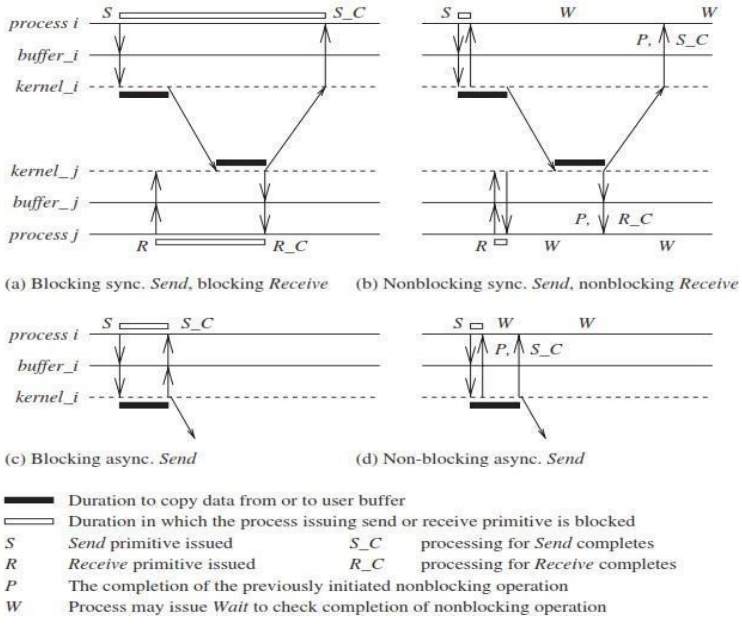



**Figure 1.8** Blocking/non-blocking and synchronous/asynchronous primitives [12]. Process $P_i$ is sending and process $P_j$ is receiving. (a) Blocking synchronous *Send* and blocking (synchronous) *Receive*. (b) Non-blocking synchronous *Send* and nonblocking (synchronous) *Receive*. (c) Blocking asynchronous *Send*. (d) Non-blocking asynchronous *Send*.

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*** (See Figure 1.8(a)) The data gets copied from the user buffer to the kernel buffer and is then sent over the network.
- After the data is copied to the receiver's system buffer and a *Receive* call has been issued, an acknowledgement back to the sender causes control to return to the process that invoked the *Send* operation and completes the *Send*.
- **non-blocking synchronous *Send*** Control returns back to the invoking process as soon as the copy of data from the user buffer to the kernel buffer is initiated.
- A parameter in the non-blocking call also gets set with the handle of a location that the user process can later check for the completion of the synchronous send

operation.

- The location gets posted after an acknowledgement returns from the receiver, as per the semantics described for (a).
- The user process can keep checking for the completion of the non-blocking synchronous *Send* by testing the returned handle, or it can invoke the blocking *Wait* operation on the returned handle (Figure 1.8(b)).
- **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, the user process that invokes the *Send* is blocked until the data is copied from
  the user's buffer to the network.)
- **non-blocking asynchronous *Send*** The user process that invokes the *Send* is blocked until the transfer of the data from the user's buffer to the kernel buffer is initiated. (For the unbuffered option, the user process that invokes the *Send* is blocked until the transfer of the data from the user's buffer to the network is initiated.)
- Control returns to the user process as soon as this transfer is initiated, and a parameter in the non-blocking call also gets set with the handle of a location that the user process can check later using the *Wait* operation for the completion of the asynchronous *Send* operation.
- The asynchronous *Send* completes when the data has been copied out of the user's buffer. The checking for the completion may be necessary if the user wants to reuse the buffer from which the data was sent.
- **Blocking *Receive*** The *Receive* call blocks until the data expected arrives and is written in the specified user buffer. Then control is returned to the user process.
- **non-blocking *Receive*** The *Receive* call will cause the kernel to register the call and return the handle of a location that the user process can later check for the completion of the non-blocking *Receive* operation.
- A synchronous *Send* is easier to use from a programmer's perspective because the handshake between the *Send* and the *Receive* makes the communication appear instantaneous, thereby simplifying the program logic.
  - The *Receive* may not get issued until much after the data arrives at $P_j$, in which case the data arrived would have to be buffered in the system buffer at $P_j$ and not in the user buffer.
  - At the same time, the sender would remain blocked. Thus, a synchronous *Send* lowers the efficiency within process $P_i$.

- 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 potentially 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 is not attainable in a distributed system, what is more generally indicated is that for a large granularity of code, usually termed as a *step*, the processors are synchronized.
- This abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

**Libraries and standards**

- There exists a wide range of primitives for message-passing.
- Many commercial software products (banking, payroll, etc., applications) use proprietary primitive libraries supplied with the software marketed by the vendors (e.g., the IBM CICS software which has a very widely installed customer base worldwide uses its own 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. An example asynchronous execution with four processes $P0$ to $P3$ is shown in Figure 1.9.

- The arrows denote the messages; the tail and head of an arrow mark the *send* and *receive* event for that message, denoted by a circle and vertical line, respectively. Non-communication events, also termed as *internal* events, are shown by shaded circles.
- A *synchronous execution* is an execution in which (i) processors are synchronized and the clock drift rate between any two processors is bounded,

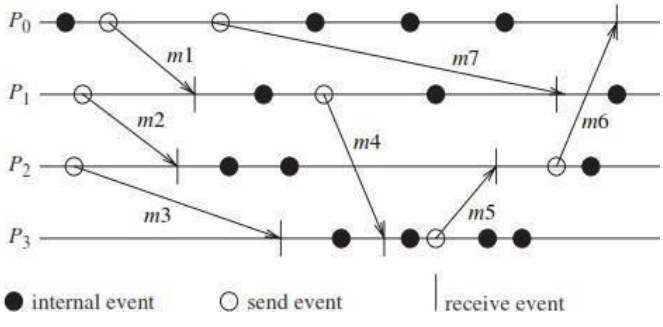



**Figure 1.9** An example of an asynchronous execution in a message-passing system. A timing diagram is used to illustrate the execution.

(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.

- It is easier to design and verify algorithms assuming synchronous executions because of the coordinated nature of the executions at all the processes.

- However, there is a hurdle to having a truly synchronous execution. It is practically difficult to build a completely synchronous system, and have the messages delivered within a bounded time.

- Therefore, this synchrony has to be simulated under the covers, and will inevitably involve delaying or blocking some processes for some time durations.

- Thus, synchronous execution is an abstraction that needs to be provided to the programs. When implementing this abstraction, observe that the fewer the steps or "synchronizations" of the processors, the lower the delays and costs.

- 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.

- This is the commonly understood notion of a synchronous execution. Within each round/phase/step, there may be a finite and bounded number of sequential sub-rounds (or sub- phases or sub-steps) that processes execute. Each sub-round is assumed to send at most one message per process; hence the message(s) sent will reach in a single message hop.
- The timing diagram of an example synchronous execution is shown in Figure 1.10. In this system, there are four nodes $P0$ to $P3$. In each round, process $Pi$ sends a message to $P(i+1) \ mod \ 4$ and $P(i-1) \ mod \ 4$ and calculates some
- application-specific function on the received values.

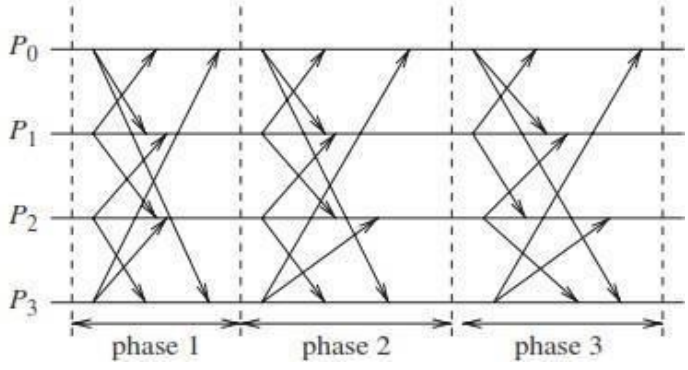



**Figure 1.10** An example of a synchronous execution in a message-passing system. All the messages sent in a round are received within that same round.

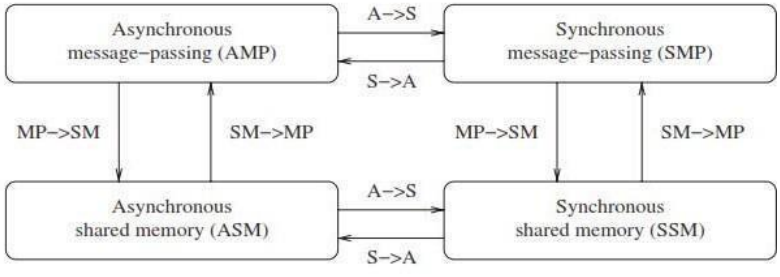



**Figure 1.11** Emulations among the principal system classes in a failure-free system.

**7. Explain the Design issues and challenges in distributed systems. (or) What are the functions must be addressed while designing and building a distributed system? Explain. (Nov/Dec 2020/Apr/May 2021)**

- Distributed computing systems have been in widespread existence since the 1970s when the Internet and ARPANET came into being.
- At the time, the primary issues in the design of the distributed systems included providing access to remote data in the face of failures, file system design, and directory structure design.
- While these continue to be important issues, many newer issues have surfaced as the widespread proliferation of the high-speed high- bandwidth internet and distributed applications continues rapidly.

  We describe the important design issues and challenges after categorizing them as

  (i) having a greater component related to systems design and operating systems design, or

  (ii) having a greater component related to algorithm design, or

  (iii) emerging from recent technology advances and/or driven by new applications.

  There is some overlap between these categories.

  However, it is useful to identify these categories because of :

  (i)the systems community, (ii) the theoretical algorithms community within distributed computing, and (iii) the forces driving the emerging applications and technology.

**Distributed systems challenge 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. Some example mechanisms are: remote procedure call (RPC), remote object invocation (ROI), message-oriented communication versus stream-oriented communication.
- **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 trans- parent and

scalable manner.

- Naming in mobile systems provides additional challenges because naming cannot easily be tied to any static geographical topology.

- **Synchronization** Mechanisms for synchronization or coordination among the processes are essential. Mutual exclusion is the classical example of synchronization.
- Synchronizing physical clocks, and devising logical clocks that capture the essence of the passage of time, as well as global state recording algorithms, all require different forms of synchronization.
- **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. Traditional issues such as file system design have to be reconsidered in the setting of a distributed system.
- **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.
- A simple example issue is deciding the level of granularity (i.e., size) of data access.
- **Fault tolerance** Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes. Process resilience, reliable communication, distributed commit, checkpointing and recovery, agreement and consensus, failure detection, and self-stabilization are some of the mechanisms to provide fault-tolerance.
- **Security** Distributed systems security involves various aspects of cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management.
- **Applications Programming Interface (API) and transparency** The API for communication and other specialized services is important for the ease of use and wider adoption of the distributed systems services by non-technical users.
- Transparency deals with hiding the implementation policies from the user, and can be 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.

- The ability to relocate the resources as they are being accessed is *relocation transparency*. *Replication transparency* does not let the user become aware of any replication. *Concurrency transparency* deals with masking the con- current use of shared resources for the 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.

**Algorithmic challenges in distributed computing**

**Designing useful execution models and frameworks**

- The *interleaving* model and *partial order* model are two widely adopted models of distributed system executions.
- They have proved to be particularly 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 reasoning more formally with and proving the correctness of distributed programs.

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

- The distributed system is modeled as a distributed graph, and the graph algorithms form the building blocks for a large number of higher-level com- munication, data dissemination, object location, and object search functions.
- The algorithms need to deal with dynamically changing graph characteristics, such as to model varying link loads in a routing algorithm.
- The efficiency of these algorithms impacts not only the user-perceived latency but also the traffic and hence the load or congestion in the network. Hence, the design of efficient distributed graph algorithms is of paramount importance.

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

- The processes in the system are spread across three-dimensional physical space. Another dimension, time, has to be superimposed uniformly across space.
- The challenges pertain to providing accurate *physical time*, and to providing a variant of time, called *logical time*.
- Logical time is relative time, and eliminates the overheads of providing physical time for applications where physical time is not required.

- More importantly, logical time 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.
- Due to the inherent distributed nature of the system, it is not possible for any one process to directly observe a meaningful global state across all the processes.

**8. Explain in detail about Synchronization/coordination mechanisms**

**.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.
- Overcoming this limited observation is necessary for taking any actions that would impact other processes.
- The synchronization mechanisms can also be viewed as resource management and concurrency management mechanisms to streamline the behavior of the processes that would otherwise act independently. Here are some examples of problems requiring synchronization:

- **Physical clock synchronization** Physical clocks usually diverge in their values due to hardware limitations. Keeping them synchronized is a fundamental challenge to maintain common time.
- **Leader election** All the processes need to agree on which process will play the role of a distinguished process – called a leader process. A leader is necessary even for many distributed algorithms because there is often some asymmetry – as in initiating some action like a broadcast or collecting the state of the system, or in "regenerating" a token that gets "lost" in 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 should be coordinated to avoid duplicate work, and deadlock resolution should be coordinated 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.

**Explain the types of Group Communications used in Distributed System. (Nov/Dec 2022)**

**Group communication, multicast, and ordered message delivery**

- A group is a collection of processes that share a common context and collaborate on a common task within an application domain.
- 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, possibly violating the semantics of the distributed program.

**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 such as debugging, sensing the environment, and in industrial process control.
- On-line algorithms for monitoring such predicates are hence important.
- An important paradigm for monitoring distributed events is that of *event streaming*, wherein streams of relevant events reported from different processes are examined collectively to detect predicates. Typically, the specification of such predicates uses physical or logical time relationships.

**Distributed program design and verification tools**

- Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering.
- Designing mechanisms to achieve these design and verification goals is a challenge.

**Debugging distributed programs**

- Debugging sequential programs is hard; debugging distributed programs is that much harder because of the concurrency in actions and the ensuing 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 them to be replicated in the distributed system.
- Managing such replicas in the face of updates introduces the problems of ensuring consistency among the replicas and cached copies. World Wide Web design – caching, searching, scheduling

**Distributed shared memory abstraction**

- A shared memory abstraction simplifies the task of the programmer because he or she has to deal only with read and write operations, and no message communication primitives.
- However, under the covers in the middleware layer, the abstraction of a shared address space has to be implemented by using message-passing.

**Wait-free algorithms**

- Wait-freedom, which can be informally defined as the ability of a process to complete its execution irrespective of the actions of other processes.
- Gained prominence in the design of algorithms to control acccess to shared resources in the shared memory abstraction.
- It corresponds to $n - 1$-fault resilience in a $n$ process system and is an important principle in fault-tolerant system design.
- While wait-free algorithms are highly desirable, they are also expensive, and designing low overhead wait-free algorithms is a challenge.
- **Mutual exclusion** A first course in operating systems covers the basic algorithms (such as the Bakery algorithm and using semaphores) for mutual exclusion in a multiprocessing (uniprocessor or multiprocessor) shared memory setting.
- **Register constructions** In light of promising and emerging technologies of tomorrow – such as biocomputing and quantum computing – that can alter the present foundations of computer "hardware" design, we need to revisit the assumptions of memory access of current systems that are exclusively based on semiconductor technology and the von Neumann architecture.
- Specifically, the assumption of single/multiport memory with serial access via the bus in tight synchronization with the system hardware clock may not be a valid assumption in the possibility of "unrestricted" and "overlapping" concurrent access to the same memory location.
- 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 field 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 permitted.

**Consistency models**

- For multiple copies of a variable/object, varying degrees of consistency among the replicas can be allowed.
- These represent a trade-off of coherence versus cost of implementation.
- Definition of consistency (such as in a uniprocessor system) would be expensive to implement in terms of high latency, high message over- head, and low concurrency.

**Reliable and fault-tolerant distributed systems**

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

**Consensus algorithms**

- All algorithms ultimately rely on message- passing, and the recipients take actions based on the contents of the received messages.
- Consensus algorithms allow correctly functioning 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.
- The goal of the malicious processes is to prevent the correctly functioning processes from reaching agreement.
- The malicious processes operate by sending messages with misleading information, to confuse the correctly functioning processes.

**Replication and replica management**

- Replication (as in having backup servers) is a classical method of providing fault-tolerance.
- The triple modular redundancy (TMR) technique has long been 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.

**Distributed databases and distributed commit**

- For distributed databases, the traditional properties of the transaction (A.C.I.D. – atomicity, consistency, isolation, durability) need to be preserved in the

distributed setting.

- The field of traditional "transaction commit" protocols is a fairly mature area. Transactional properties can also be viewed as having a counterpart 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.
- Self-stabilizing algorithms require some in-built redundancy to track additional variables of the state and do extra work. Designing efficient self-stabilizing algorithms is a challenge.

**Checkpointing and recovery algorithms**

- Checkpointing involves peri- odically recording the current state on secondary storage so that, in case of a failure, the entire computation is not lost but can be recovered from one of the recently taken checkpoints.
- Checkpointing in a distributed environment is difficult because if the checkpoints at the different processes are not coordinated, the local checkpoints may become useless because they are inconsistent with the checkpoints at other processes.

**Failure detectors**

- A fundamental limitation of asynchronous distributed systems is that there is no theoretical bound on the message transmission times.
- This implies that it is impossible using message transmission to determine whether some other process across the network is alive or has failed.
- Failure detectors represent a class of algorithms that probabilistically suspect another process as having failed (such as after timing out after non-receipt of a message for some time), 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.
- Load balancing may be necessary because of a variety of factors such as high network traffic or high request rate causing the network connection to be a bottleneck, or high computational load.
- A common situation where load balancing is used is in server farms, where the objective is to service incoming client requests with the least turnaround time.
- Several results from traditional operating systems can be used here, although they need to be adapted to the specifics of the distributed environment. The following are some forms of load balancing:

- **Data migration** The ability to move data (which may be replicated) 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.
- On-line or dynamic changes to the schedule are also harder to make without a global view of the state.

**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 (propagation and transmission times) and access to shared resources can lead to large delays which must be minimized.
- The user- perceived turn-around time is very important.
- The following are some example issues arise in determining the performance:

- **Metrics** Appropriate metrics must be defined or identified for measuring the performance of theoretical distributed algorithms.

- The former would involve various complexity measures on the metrics, whereas the latter would involve various system and statistical metrics.

**Measurement methods/tools**

- As a real distributed system is a complex entity and has to deal with all the difficulties that arise in measuring performance over a WAN/the Internet, appropriate methodologies and tools must be developed for measuring the performance metrics.

**9. Discuss the 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.
- The characteristics of communication are different; many issues such as range of transmission and power of transmission come into play, besides various engineering issues such as battery power conservation, interfacing with the wired Internet, signal processing and interference.
- From a computer science perspective, there is a rich set of problems such as routing, location management, channel allocation, localization and position estimation, and the overall management of mobility.
- There are two popular architectures for a mobile network. The first is the *base-station* approach, also known as the *cellular approach*, wherein a *cell* which is the geographical region within range of a static but powerful base transmission station is associated with that base station.
- All mobile processes in that cell communicate with the rest of the system via the base station. T
- The second approach is the *ad-hoc network* approach where there is no base station (which essentially acted as a centralized node for its cell).
- All responsibility for communication is distributed among the mobile nodes, wherein mobile nodes have to participate in routing by forwarding packets of other pairs of communicating nodes. It poses many graph-theoretical challenges from a computer science perspective, in addition to various engineering challenges.

**Sensor networks**

- A sensor is a processor with an electro-mechanical interface that is capable of sensing physical parameters, such as temperature, velocity, pressure, humidity, and chemicals.

- An important paradigm for monitoring distributed events is that of *event streaming,* which was defined earlier.
- The streaming data reported from a sensor network differs from the streaming data reported by "computer processes" in that the events reported by a sensor network are in the environment, external to the computer network and processes.
- This limits the nature of information about the reported event in a sensor network.
- Sensor networks have a wide range of applications. Sensors may be mobile or static; sensors may communicate wirelessly, although they may also communicate across a wire when they are statically installed.
- Sensors may have to self-configure to form an ad-hoc network, which introduces a whole new set of challenges, such as position estimation and time estimation.

**Ubiquitous or pervasive computing**

- Ubiquitous systems represent a class of computing where the processors embedded in and seamlessly pervading through the environment perform application functions in the background.
- The intelligent home, and the smart workplace are some example of ubiquitous environments currently under intense research and development.
- Ubiquitous systems are essentially distributed systems; recent advances in technology allow them to leverage wireless communication and sensor and actuator mechanisms.
- They can be self-organizing and network-centric, while also being resource constrained.
- Such systems are typically characterized as having many small processors operating collectively in a dynamic ambient network.
- The processors may be connected to more powerful networks and processing resources in the background for processing and 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, without any hierarchy among the processors.
- Thus, all processors are equal and play a symmetric role in the computation. P2P computing arose as a paradigm shift from client–server computing where the roles among the processors are essentially asymmetrical.

- P2P networks are typically self-organizing, and may or may not have a regular structure to the network.
- No central directories (such as those used in domain name servers) for name resolution and object lookup are allowed.

**Publish-subscribe, content distribution, and multimedia**

With the explosion in the amount of information, there is a greater need to receive and access only information of interest.

Such information can be specified using filters. In a dynamic environment where the information constantly fluctuates (varying stock prices is a typical example), there needs to be:

(i) an efficient mechanism for distributing this information (*publish*),

(i) an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information (*subscribe*), and (iii) an efficient mechanism for aggregating large volumes of published information and filtering it as per the user's subscription filter.

**Distributed agents**

- Agents are software processes or robots that can move around the system to do specific tasks for which they are specially programmed.
- The name "agent" derives from the fact that the agents do work on behalf of some broader objective.
- Agents collect and process information, and can exchange such information with other agents is 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.
- A traditional example is: examining the purchasing patterns of customers in order to profile the customers and enhance the efficacy of directed marketing schemes.
- The mining can be done by applying database and artificial intelligence techniques to a data repository.
- In many situations, the data is necessarily distributed and cannot be collected in a single repository, as in 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.

**Grid computing**

- Analogous to the electrical power distribution grid, it is envisaged that the information and computing grid will become a reality some day.
- Very simply stated, idle CPU cycles of machines connected to the network will

be avail- able to others.

**Security in distributed systems**

- The traditional challenges of security in a distributed setting include: confidentiality (ensuring that only authorized processes can access certain information), authentication (ensuring the source of received information and the identity of the sending process), and availability (maintaining allowed access to services despite malicious actions).
- The goal is to meet these challenges with efficient and scalable solutions. These basic challenges have been addressed in traditional distributed settings.

**10. Explain in detail about a 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 application runs as a collection of processes on a distributed system.

**A distributed program**

- A distributed program is composed of a set of $n$ asynchronous processes $p1$, $p2$, ... , $pi$, ... , $pn$ that communicate by message passing over the communication network. Without loss of generality, we assume that each process is running on a different processor.
- The processes do not share a global memory and communicate solely by passing messages.
- Let $Cij$ denote the channel from process $pi$ to process $pj$ and let $mij$ denote a message sent by $pi$ to $pj$.

- The communication delay is finite and unpredictable. Also, these processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous – a process may execute an action spontaneously and a process sending a message does not wait for the delivery of the message to be complete.
- The global state of a distributed computation is composed of the states of the processes and the communication channels.
- The state of a process is characterized by the state of its local memory and depends upon the context. The state of a channel is characterized by the set of messages in transit in the channel.

**A model of distributed executions**

- The execution of a process consists of a sequential execution of its actions. The actions are atomic and the actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events.
- Let $e^x$ denote the $x^{th}$ event at process $p_i$. Subscripts and/or superscripts will be dropped when they are irrelevant or are clear from the context. For a message $m$, let *send(m)* and *rec(m)* denote its 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 (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received).
- 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 $e^1, e^2, \ldots, e^x, e^{x+1}, \ldots$ and is denoted by

$$H_i = (h_i, \rightarrow_i),$$

- 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 $p_i$.
- The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process.

A relation $\rightarrow_{msg}$ that captures the causal dependency due to message exchange, is defined as follows. For every message $m$ that is exchanged between two processes, we have

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

- Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of correspond- ing send and receive events.

The evolution of a distributed execution is depicted by a space–time diagram. Figure 2.1 shows the space–time diagram of a distributed execution involving three processes. A horizontal line represents the progress of the

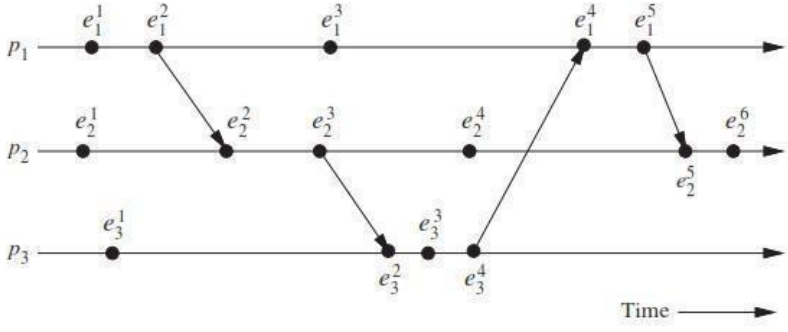



**Figure 2.1** The space–time diagram of a distributed execution.

process; a dot indicates an event; a slant arrow indicates a message transfer.

- In this figure, for process $p1$, the second event is a message send event, the third event is an internal event, and the fourth 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 $H = \cup_i h_i$ denote the set of events executed in a distributed computation.
- Next, we define a binary relation on the set $H$, denoted as $\rightarrow$, that expresses causal dependencies between events in the distributed execution.

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

- The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as fl=(*H*, →).
- Note that the relation → is Lamport's "happens before" relation. For any two events $e_i$ and $e_j$, if $e_i \rightarrow e_j$, then event $e_j$ is directly or transitively dependent on event $e_i$; graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space–time diagram that starts at $e_i$ and ends at $e_j$. For example, in Figure 2.1, $e^1 \rightarrow e^3$ and $e^3 \rightarrow$ $^6$ $^1$ $^3$ $^3$ $^2$ $e$ . Note that relation → denotes flow of information in a distributed computation and $e_i \rightarrow e_j$ dictates that all the information available at $e_i$ is potentially accessible at $e_j$. For example, in Figure 2.1, event $e^6$ has $^2$ the knowledge of all other events shown in the figure.
- For any two events $e_i$ and $e_j$, $e_i \bullet\!\!\rightarrow e_j$ denotes the fact that event $e_j$ does not directly or transitively dependent on event $e_i$. That is, event $e_i$ does not causally affect event $e_j$. Event $e_j$ is not aware of the execution of $e_i$ or any event executed after $e_i$ on the same process. For example, in Figure 2.1, $e^3 \bullet\!\!\rightarrow e^3$ and $e^4 \bullet\!\!\rightarrow$ $^1$ $^3$ $^2$ $^3$ $e$ . Note the following two rules:
  - for any two events $e_i$ and $e_j$, $e_i \bullet\!\!\rightarrow e_j \bullet\!\!\Rightarrow e_j \bullet\!\!\rightarrow e_i$
  - for any two events $e_i$ and $e_j$, $e_i \rightarrow e_j \Rightarrow e_j \quad e_i$.

  For any two events $e_i$ and $e_j$, if $e_i \; e_j$ and $e_j \bullet\!\!\rightarrow e_i$, then events $e_i$ and $e_j$ are said to be concurrent and the relation is denoted as $e_i \equiv e_j$. In the execution of Figure 2.1, $e^3 \equiv e^3$ and $e^4 \equiv e^1$. Note that relation ≡ is not transitive; that is, $(e_i \equiv e_j) \wedge (e_j \equiv e_k) \bullet\!\!\Rightarrow e_i \equiv e_k$. For example, in Figure 2.1, $e^3 \equiv e^4$ and $e^4 \equiv e^5$, however, $e^3 \bullet\!\!\equiv e^5$.

Note that for any two events $e_i$ and $e_j$ in a distributed execution, $e_i \rightarrow e_j$ or $e_j \rightarrow e_i$, or $e_i \equiv e_j$.

**Logical vs. physical concurrency**

In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other.

- Physical concurrency, on the other hand, has a connotation that the events occur at the same instant in physical time.
- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time. For example, in Figure 2.1, events in the set $\{e^3, e^4, e^3\}$ are logically concurrent, but they occurred at different instants in physical time.
- However, note that if processor speed and message delays had been different, the execution of these events could have very well coincided in physical time.
- Whether a set of logically concurrent events coincide in the physical time or in what order in the physical time they occur does not change the outcome of the computation.
- Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, for all practical and theoretical purposes, we can assume that these events occured at the same instant in physical time.

**11. Write in detail about Models of communication networks. (or) Why global states are essential in distributed computing systems? Elaborate with an example. (NOV/DEC 2021)**

**Models of communication networks**

- There are several models of the service provided by communication networks, namely, FIFO (first-in, first-out), non-FIFO, and causal order37ing. In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- 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)* −→ *send(mkj)*, then *rec(mij)* −→ *rec(mkj)*.

- That is, this property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causal- ity relation.

- Causally ordered delivery of messages implies FIFO message delivery.

- Furthermore, note that CO ⊂ FIFO ⊂ Non-FIFO.
- Causal ordering model is useful in developing distributed algorithms.
- Generally, it considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.
- For example, in replicated database systems, it is important that every process responsible for updating a replica receives the updates in the same order to maintain database consistency.
- Without causal ordering, each update must be checked to ensure that database consistency is not being violated. Causal ordering eliminates the need for such checks.

**Global state of a distributed system**

- The global state of a distributed system is a collection of the local states of its components, namely, the processes and the communication channels.

- The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.

- The state of a channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels, thus causing transitions in global system state.
- For example, an internal event changes the state of the process at which it occurs. A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received).

## Global state

The global state of a distributed system is a collection of the local states of the processes and the channels. Notationally, the global state *GS* is defined as

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

- This will be possible if the local clocks at processes were perfectly synchronized or there was a global system clock that could be instantaneously read by the processes. However, both are impossible.
- However, it turns out that even if the state of all the components in a distributed system has not been recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent.
- Basic idea is that an effect should not be present without its cause.
- A message cannot be received if it was not sent; that is, the state should not violate causality.
- Such states are called *consistent global states* and are meaningful global states.
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

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

In the distributed execution of Figure 2.2, a global state $GS1$ consisting of local states $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is inconsistent because the state of $p2$ has recorded the receipt of message $m12$, however, the state of $p1$ has not recorded its send. On the contrary, a global state $GS2$ consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all the channels are empty except $C_{21}$ that contains message $m_{21}$.

A global state $GS = \{ LS_i^{x_i}, SC_{j,k}^{y_j,z_k} \}_{jk}$ is *transitless* iff

**12. Explain in detail about Models of process communications. (Nov/Dec 2022)**

**Models of process communications**

- There are two basic models of process communications synchronous and asynchronous.
- The *synchronous* communication model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process.
- The sender process resumes execution only after it learns that the receiver process has accepted the message.
- Thus, the sender and the receiver processes must synchronize to exchange a message.
- On the other hand, *asynchronous* communication model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
- After having sent a message, the sender process does not wait for the message to be delivered to the receiver process.
- The message is bufferred by the system and is delivered to the receiver process when it is ready to accept the message.
- A buffer overflow may occur if a process sends a large number of messages in a burst to another process.
- Neither of the communication models is superior to the other.
- Asynchronous communication provides higher parallelism because the sender process can execute while the message is in transit to the receiver.
- However, an implementation of asynchronous communication requires more complex buffer management.
- In addition, due to higher degree of parallelism and non-determinism, it is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications.
- The state space of such algorithms are likely to be much larger. Synchronous communication is simpler to handle and implement. However, due to frequent blocking, it is likely to have poor performance and is likely to be more prone to deadlocks.

## 2-Marks

**1. What is distributed system? (or) Define Distributed Systems. (Apr/May 2023)**

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. Distributed systems have been in existence since the start of the universe.

**2. What are the characteristics of distributed system.**

A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features:

- No common physical clock
- shared memory
- Geographical separation
- Autonomy and heterogeneity

**3. What are the advantages of the distributed system?**

**Scalability** As the processors are usually connected by a wide-area net- work, adding more processors does not pose a direct bottleneck for the communication network.

**Modularity and incremental expandability** Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algorithms. Similarly, existing processors may be easily replaced by other processors.

**4. What are the Characteristics of parallel systems?**

A parallel system may be broadly classified as belonging to one of three types:

- A *multiprocessor system* is a parallel system in which the multiple proces- sors have *direct access to shared memory* which forms a common address space.
- A multiprocessor system *usually* corresponds to a uniform memory access (UMA) architecture in which the access latency.
- The processors are in very close physical proximity and are connected by an interconnection network.

- Interprocess communication across processors is traditionally through read and write operations on the shared memory.

**5. Define multi computer parallel system.**

A *multicomputer parallel system* is a parallel system in which the multiple processors *do not have direct access to shared memory.*

- The memory of the multiple processors may or may not form a common address space. Such computers usually do not have a common clock.
- A multicomputer system that has a common address space *usually* corresponds to a non-uniform mem- ory access (NUMA) architecture in which the latency to access various shared memory locations from the different processors varies.

**6. Define MISD?**

This mode corresponds to the execution of different operations in parallel on the same data. This is a specialized mode of operation with limited but niche applications, e.g., visualization.

**7. Define MIMD.**

In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems.

**8. Define Coupling.**

The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules.

**9. Define Parallelism or speedup of a program on a specific system.**

- This is a measure of the relative speedup of a specific program, on a given machine. The speedup depends on the number of processors and the mapping of the code to the processors.
- It is expressed as the ratio of the time *T(1)* with a single processor, to the time *T(n)* with *n* processors.

**10. What Parallelism within a parallel/distributed program**

- This is an aggregate measure of the percentage of time that all the proces- sors are executing CPU instructions.
- It is opposed to waiting for communication (either via shared memory or message-passing) operations to complete.

**11. What is the Concurrency of a program?**

- The *parallelism/concurrency* in a parallel/distributed program can be measured by the ratio of the number of local (non-communication and non-shared memory access) operations to the total number of operations, including the communi- cation or shared memory access operations.

**12. What is the Granularity of a program?**

- The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as *granularity*.
- If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions.

**13. What are Synchronous primitives?**

A *Send* or a *Receive* primitive is *synchronous* if both the *Send()* and *Receive()* handshake with each other.

**14. What is 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 has been copied out of the user-specified buffer.

**15. What is Blocking primitives?**

A primitive is *blocking* if control returns to the invoking process after the processing for the primitive (whether in syn- chronous or asynchronous mode) completes.

**16. What are Non-blocking primitives?**

A primitive is *non-blocking* if control returns back to the invoking process immediately after invocation, even though the operation has not completed.

***17.* Define Blocking synchronous *Send.***

The data gets copied from the user buffer to the kernel buffer and is then sent over the network.

**18. Define non-blocking synchronous *Send.***

Control returns back to the invoking process as soon as the copy of data from the user buffer to the kernel buffer is initiated.

**19. Define 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.

**20. Define non-blocking asynchronous *Send.***

The user process that invokes the *Send* is blocked until the transfer of the data from the user's buffer to the kernel buffer is initiated.

**21. What is Blocking *Receive?***

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

**22. What is non-blocking *Receive?***

The *Receive* call will cause the kernel to register the call and return the handle of a location that the user process can later check for the completion of the non-blocking *Receive* operation.

**23. What is Processor synchrony?**

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

**24. Differentiate between Synchronous versus asynchronous executions.**

- An *asynchronous execution* is an execution in which

  (i) there is no pro- cessor 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.
- 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.

**25. What is Communication in the network?**

This task involves designing appropriate mechanisms for communication among the processes in the network. Some exam- ple mechanisms are: remote procedure call (RPC), remote object invo-cation (ROI), message-oriented communication versus stream-oriented communication.

**26. What are 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.

**27. What is Naming?**

Devising easy to use and robust schemes for names, identifiers, and addresses is essential for locating resources and processes in a trans- parent and scalable manner.

Naming in mobile systems provides additional challenges because naming cannot easily be tied to any static geographical topology.

**28. What is Synchronization?**

Mechanisms for synchronization or coordination among the processes are essential. Mutual exclusion is the classical example of synchronization.

**29. What is 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. Traditional issues such as file system design have to be reconsidered in the setting of a distributed system.

**30. What is the use of replication?**

To avoid bottlenecks, to provide fast access to data, and to provide scalability, replication of data objects is highly desirable.

**31. Define Fault tolerance.**

Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes. Process resilience, reliable communication, distributed commit, checkpointing and recovery, agreement and consensus, failure detection, and self-stabilization are some of the mechanisms to provide fault-tolerance.

**32. Define Security in distributed systems.**

Distributed systems security involves various aspects of cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management.

**33. What are the Applications Programming Interface (API)?**

The API for communication and other specialized services is important for the ease of use and wider adoption of the distributed systems services by non-technical users.

**34. What is transparency?**

Transparency deals with hiding the implementation policies from the user, and can be classified as follows. *Access transparency* hides differences in data representation on different systems and provides uniform operations to access system resources.

**35. What is Location transparency?**

*Location transparency* makes the locations of resources transparent to the users. *Migration transparency* allows relocating resources without changing names.

**36. What is relocation transparency?**

The ability to relocate the resources as they are being accessed is *relocation transparency*.

**37. What is Replication transparency?**

*Replication transparency* does not let the user become aware of any replication. *Concurrency transparency* deals with masking the con- current use of shared resources for the user.

**38. Define Failure transparency.**

*Failure transparency r*efers to the system being reliable and fault-tolerant.

**39. What is 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.

**40. What are the Designing useful execution models and frameworks?**

- The *interleaving* model and *partial order* model are two widely adopted models of distributed system executions.
- They have proved to be particularly useful for operational reasoning and the design of distributed algorithms.

List the types examples of models.

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 reasoning more formally with and proving the correctness of distributed programs.

**41. What is global state?**

Observing the *global state* of the system (across space) also involves the time dimension for consistent observation.

**42. Define Physical clock synchronization.**

Physical clocks usually diverge in their values due to hardware limitations. Keeping them synchronized is a fundamental challenge to maintain common time.

**43. What is Leader election?**

All the processes need to agree on which process will play the role of a distinguished process – called a leader process. A leader is necessary even for many distributed algorithms because there is often some asymmetry – as in initiating some action like a broadcast or collecting the state of the system, or in "regenerating" a token that gets "lost" in the system.

**44. What is Mutual exclusion?**

This is clearly a synchronization problem because access to the critical resource(s) has to be coordinated.

**45. What is Deadlock detection and resolution?**

Deadlock detection should be coordinated to avoid duplicate work, and deadlock resolution should be coordinated to avoid unnecessary aborts of processes.

**46. What is Termination detection?**

This requires cooperation among the processes to detect the specific global state of quiescence.

**47. Define 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.

**48. What is Group communication?**

- A group is a collection of processes that share a common context and collaborate on a common task within 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.

**49. What is event streaming?**

- An important paradigm for monitoring distributed events is that of *event streaming*, wherein streams of relevant events reported from differ- ent processes are examined collectively to detect predicates. Typically, the specification of such predicates uses physical or logical time relationships.

**50. Define Data replication, consistency models, and caching**

- Fast access to data and other resources requires them to be replicated in the distributed system.
- Managing such replicas in the face of updates introduces the problems of ensuring consistency among the replicas and cached copies.World Wide Web design – caching, searching, scheduling.

**51. Define Distributed shared memory abstraction**

- A shared memory abstraction simplifies the task of the programmer because he or she has to deal only with read and write operations, and no message communication primitives.

**52. What is Wait-free algorithms?**

- Wait-freedom, which can be informally defined as the ability of a process to complete its execution irrespective of the actions of other processes.

**53. What is Consistency models?**

- Definition of consistency (such as in a uniprocessor system) would be expensive to implement in terms of high latency, high message over- head, and low concurrency.

**54. Define Consensus algorithms.**

- Consensus algorithms allow correctly functioning 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.

**55. What is a distributed program? (Nov/Dec 2022)**

- A distributed program is composed of a set of $n$ asynchronous processes $p1$, $p2$, ... , $pi$, .. . , $pn$ that communicate by message passing over the communication network. Without loss of generality, we assume that each process is running on a different processor.
- The processes do not share a global memory and communicate solely by passing messages.
- Let $Cij$ denote the channel from process $pi$ to process $pj$ and let $mij$ denote a message sent by $pi$ to $pj$.

**56. What is 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, namely, internal events, message send events, and message receive events.

**57. Define Causal precedence relation**

- The execution of a distributed application results in a set of distributed events produced by the processes. Let $H = \cup_i h_i$ denote the set of events executed in a distributed computation.

**58. Differentiate between Logical vs. physical concurrency**

- Physical concurrency, on the other hand, has a connotation that the events occur at the same instant in physical time.
- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time. For example, events in the set $\{e^3, e^4, e^3\}$ are logically concurrent, but they occurred at different instants in physical time.

**59. Name the primitives for distributed communication. (NOV/DEC2021)**

- Synchronous primitives
- ASynchronous primitives
- Blocking primitives
- Non-blocking primitives

**60. Compare message passing systems and shared memory systems. (NOV/DEC 2021)**

| Shared Memory | Message Passing |
|---|---|
| It is one of the region for data communication | Mainly the message passing is used for communication. |
| It is used for communication between single processor and multiprocessor systems where the processes that are to be communicated present on the same machine and they are sharing common address space. | It is used in distributed environments where the communicating processes are present on remote machines which are connected with the help of a network. |
| The shared memory code that has to be read or write the data that should be written explicitly by the application programmer. | Here no code is required because the message passing facility provides a mechanism for communication and synchronization of actions that are performed by the communicating processes. |

**61. Why do we need a distributed system? (Nov/Dec 2020)(Apr/May 2021)**

Distributed systems **provide scalability and improved performance in ways that monolithic systems can't**, and because they can draw on the capabilities of other computing devices and processes, distributed systems can offer features that would be difficult or impossible to develop on a single system.

**62. List out the distributed system challenges. (Nov/Dec 2020/Apr/May 2021)**

Important challenges

(i) having a greater component related to systems design and operating systems design, or

(ii) Having a greater component related to algorithm design

(iii) Emerging from recent technology advances and/or driven by new applications.

**63. What do mean by Message Passing? (Nov/Dec 2022)**

Mainly the message passing is used for communication. It is used in distributed environments where the communicating processes are present on remote machines which are connected with the help of a network.

Here no code is required because the message passing facility provides a mechanism for communication and synchronization of actions that are performed by the communicating processes.

**64. What are the main difference between a parallel system and a distributed system. (Apr/May 2023)**

**The main difference between distributed system and parallel system is that**:

- A distributed system consists of multiple independent computers connected by a communication network, while a parallel system consists of multiple processors that communicate with each other using a shared memory.
- A distributed system is designed to provide a single cohesive system to the user, while a parallel system is designed to solve computationally intensive problems by dividing them into smaller sub-problems and solving them simultaneously.
- A distributed system uses message passing to communicate between computers, while a parallel system uses multi-threading to make full use of a single CPU.

## 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 (CO) – Total Order; Global State and Snapshot Recording Algorithms: Introduction –System Model and Definitions –Snapshot Algorithms for FIFO Channels

## PART A:

**1. Define Vector time.**

In the system of vector clocks, 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 and describes the logical time progress pi. *Vti[j]* represents process *pi's* latest knowledge of process *pj* local time.

**2. What do mean by Message Passing? (Nov/Dec 2022)**

The message passing is used for communication. It is used in distributed environment where the communicating processes are present on remote machines which are connected with the help of a network.

Here no code is required because the message passing facility provides a mechanism for communication and synchronization of actions that are performed by the communicating process.

**3. What are the various forms of message ordering? (or) Name the various message ordering paradigms used in distributed systems. (Nov/Dec 2020/Apr/May 2021)**

The various forms of message ordering paradigms are

(i) non-FIFO,
(ii) FIFO,
(iii) causal order
(iv) Synchronous order.

**4. What are the main difference between a parallel system and a distributed system? (Apr/May 2023)**

- A distributed system consists of multiple independent computers connected by a communication network, while a parallel system consists of multiple processors that communicate with each other using a shared memory.
- A distributed system is designed to provide a single cohesive system to the user, while a parallel system is designed to solve computationally intensive problems by dividing them into smaller sub-problems and solving them simultaneously.
- A distributed system uses message passing to communicate between computers, while a parallel system uses multi-threading to make full use of a single CPU.

**5. Define Asynchronous executions or a executions. (Apr/May 2023)**

**Definition:**

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

**6. Define FIFO executions**

**Definition**

A FIFO execution is an A-execution in which, for all (s, r) and $(s^t, r^t) \in T$, ($s \sim s^t$ and $r \sim r^t$ and $s \sim s^t$) $\Rightarrow r \prec r^t$.

**Explanation:**

✓ Messages are necessarily delivered in the order in which they are sent in any logical link in the system.

**7. Define Casusally ordered (CO) executions.**

**Definition**

A CO execution is an A-execution in which, for all (s, r) and $(s^t, r^t) \in T$, ( $r \sim r^t$ and $s \prec s^t$) $\Rightarrow r \prec r$

**Explanations**

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

✓ If s and $s^t$ are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false.

emphasis.

**8. Define Causal order (CO) for implementations.**

**Definition**

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

**Explanation**

✓ If the definition of causal order is restricted so that $m^1$ and $m^2$ are sent by the same process, then the property degenerates into the FIFO property.

**9. Write simple example for Message ordering. (NOV/DEC 2021) (or) Define Message order (MO).**

**Definition**

A MO execution is an A-execution in which, for all (s, r) and $(s^t, r^t) \in T$, $s \prec s^t \Rightarrow \neg(r^t \prec r)$.

**Example**

- Consider any message pair, say $m^1$ and $m^3$. $s^1 \prec s^3$ but $\neg(r^3 \prec r^1)$ isfalse.
- Hence, the execution does not satisfy MO.

**10. Define Empty-interval execution.**

**Definition**

An execution (E, $\prec$) is an empty-interval (EI) execution if for each pair of events (s, r) $\prec$ T, the open interval set {x E €| s $\prec$ x $\prec$ r} in the partial order is empty.

**11. What are the uses of 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 occurring instantaneously and

atomically.

**12. Define Causality in a synchronous execution.**

**Definition**

The synchronous causality relation on E is the smallest transitive relation that satisfies the following:

S1: If x **occurs** before y at the same process, then $x \sim y$.

S2: If $(s, r) \in T$, then for all $x \in E$, $[(x \sim s \Leftrightarrow x \sim r)$ and $(s \sim x \Leftrightarrow r \sim x)]$.

S3: If $x \sim y$ and $y \sim z$, then $x \sim z$.

**13. What is Synchronous execution?**

**Definition**

A synchronous execution (or S-execution) is an execution $(E, \sim)$ for which the causality relation $\sim$ is a Partial order.

a. Execution in an asynchronous system

b. Equivalent instantaneous communication

**14. What is Time stamping a synchronous execution?**

**Definition**

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 $P_i$, if $e_i \prec e^t$ then $T(e) < T(e^t)$.

By assuming that a send event and its corresponding receive event are viewed atomically, i.e., $s(M) \prec r(M)$ and $r(M) \prec s(M)$, it follows that for any events $e_i$ and $e_j$ that are not the send event and the receive event of the same message, $e_i \prec e_j \Rightarrow T(e_i) < T(e_j)$.

**15. What is Non-separated linear extension?**

**Definition**

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.

**Examples**

- Figure 2.2(d): $(s^2,r^2, s^3,r^3, s^1,r^1)$ is a linear extension that is non-separated. $(s^2, s^1,r^2, s^3,r^3, s^1)$ is a linear extension that is separated.
- Figure 2.3(b): $(s^1,r^1, s^2,r^2, s^3,r^3, s^4,r^4, s^5,r^5, s^6,r^6)$ is a linear extension that is non-separated. $(s^1, s^2,r^1,r^2, s^3, s^4,r^4,r^3, s^5, s^6,r^6,r^5)$ is a linear extension that is separated.

**16. Define RSC execution.**

**Definition**

- ✓ An A-execution $(E, \prec)$ is an RSC execution if and only if there exists a non-separated linear extension of the partial order $(E, \prec)$.
- ✓ In the non-separated linear extension, if the adjacent send event and its corresponding receive event are viewed atomically, then that pair of events shares a common past and a common future with each other.
- ✓ Characterization of the execution in terms of a graph structure called a *crown*; the crown leads to a feasible test for a RSC execution.

**17. What is Crown?**

**Definition**

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

**Examples**

- Figure 2.5(a): The crown is $((s^1,r^1), (s^2,r^2))$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$. This execution represents the program execution in Figure 6.4.
- Figure 2.5(b): The crown is $((s^1,r^1), (s^2,r^2))$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$.
- Figure 2.5(c): The crown is $((s^1,r^1), (s^3,r^3), (s^2,r^2))$ as we have $s^1 \prec r^3$ and $s^3 \prec r^2$ and $s^2 \prec r^1$.
- Figure 2.2(a): The crown is $((s^1,r^1), (s^2,r^2), (s^3,r^3))$ as we have $s^1 \prec r^2$ and $s^2 \prec r^3$ and $s^3 \prec r^1$.

**18. Define Crown criterion Theorem.**

The crown criterion states that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, if and only if it contains no crown.

**19. Define Time stamp for a RSC execution.**

**Definition**

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

- for any message M, $T(s(M)) = T(r(M))$;
- for each $(a, b)$ in $(E \times E) \backslash T$, $a \prec b \Rightarrow T(a) < T(b)$.

**20. What are deterministic and Non-determinism?**

The distributed programs are *deterministic*, i.e., repeated runs of the same program will produce the same partial order. **programs are *non-deterministic*.**

**21. What is Rendezvous?**

- ✓ One form of group communication is called multiway rendezvous, which is a synchronous communication among an arbitrary number of asynchronous processes.
- ✓ All the processes involved "meet with each other," i.e., communicate "synchronously" with each other at one time.
- ✓ Support for binary rendezvous communication was first provided by programming languages such as CSP and Ada.

**22. What is closed group algorithm?**

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.

**23. What is open group algorithm?**

If the sender of the multicast can be outside the destination group, the multicast algorithm is said to be an *open group* algorithm.

**24. What is Explicit tracking?**

Tracking of (source, timestamp, destination) information for messages

(i) not known to be delivered and

(ii) not guaranteed to be delivered in CO, is done explicitly using the l.Dests field of entries in local logs at nodes and o.Dests field of entries in messages.

(iii) Sets $l_{i,a}$.Dests and $o_{i,a}$.Dests contain explicit information of destinations to which $M_{i,a}$ is not guaranteed to be delivered in CO and is not known to be delivered.

The information about "d € $M_{i,a}$.Dests" is propagated up to the earliest events on all causal paths from (i, a) at which it is known that $M_{i,a}$ is delivered to d or is guaranteed to be delivered to d in CO.

**25. Explain Implicit tracking?**

Tracking of messages that are either

(i) already delivered, or

(ii) guaranteed to be delivered in CO

The information about messages

(i) already delivered or

(ii) guaranteed to be delivered in CO is deleted and not propagated because it is redundant as far as enforcing CO is concerned

(iii) However, it is useful in determining what information that is being carried in other messages and is being stored in logs at other nodes has become redundant and thus can be purged.

(iv) The semantics are implicitly stored and propagated. This information about messages that are already delivered or guaranteed to be delivered.

**26. Define Total order.**

**Definition**

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

**27. Write the condition for non-deterministic event.**

- A receive call can receive a message from any sender who has sent a messge. If the expected sender is not specified. The receive calls are non-deterministic thereceiver is willing to perform a rendezvous with any willing and ready sender.
- Multiple send and receive calls which are enabled at a process can be executed in an interchangeable order**.**

**28. What are the characteristics of group communication**

- Fault tolerance based on replicated server
  Replicated means it consist of group of servers
  Client send request to group
- Finding the discovery servers from spontaneous networks
- Better Performance through replicated data.

- Propagation of event notification.

**29. Compare open group and closed group algorithm.**

- If the sender of the multicast can be outside the destination group, the multicastalgorithm is said to be an open group algorithm
- Open group algorithms are more general and therefore more difficult to design andmore expensive to implement than closed group algorithm.
- Closed group algorithms cannot be used in several scenarios such as in largesystem where client processes are short lived and in large numbers.

**30. What are the 2 phases in obtaining in a global snapshot? (Apr/May 2023)**

- First locally recording the snapshot at every process
- Second distributing the resultant global snapshot to all the initiators.

**31. What are the 2 optimization techniques are provided to the chandy-Lamport Algorithm?**

- The first optimization combines snapshot concurrently initiated by multipleprocess into a single snapshot.
- This optimization is linked with the second optimization which deals with theefficient distribution of the global snapshot.

**32. Define Wave.**

A Wave is nothing but a flow of control message such that every process in the system is visited exactly once by a wave control message and at least one process in system can determine when this flow of control message terminates**.**

**33. How a FIFO execution is implemented?**

- To implement a FIFO logical channel over a non FIFO channel a separate numbering scheme is used to sequence the message.
- The sender assigns a sequence number and appends connection-id to each message and then transmitted then the receiver arranges the incoming messages according to the senders sequence numbers and accepts next messages as per sequence.

**34. What is guard?**

A guard Gig is a Boolean expression. If a guard Gi evaluates to true then CLi is said to be enabled, otherwise CLi is said to be disabled.

**35. List the criteria to be met by a casual ordering protocol.**

- Safety
- Liveness

**36. List the application of casual order protocol.**

Updating replicated data

- Allocating request in a fair manner and
- Synchronizing multimedia streams.

**37. Write the drawbacks of centralized algorithms.**

- Single point of failure
- Congestion

**38. Write the condition for global state to be consistent global state.**

A global state GS is a consistent global state if it satisfies the following twocondition

C1: send (mij) ∈Lsi=>mij∈SCij+rec (mij) LSj

C2: send(mij) .Lsi=>mij.SCij^rec(mij)LSj

**39. State of law of conservation of messages.**

Every Message mij that is recorded as sent in the local state of a process pi must becaptured in the state of the channel cij or in the collected local state of the receiver process Pj

**40. Define Cut and Consistent Cut.**

- A cut is a line joining an arbitrary point on each process line that slices thespace time diagram into a past and a future.
- A consistent global state corresponds t o a cut in which every message receivedin the PAST of the cut has been sent in the PAST of that cut. Such a cut is known as a consistent cut.

**41. What is the purpose of chandy and lamport algorithm? (NOV/DEC 2021)**

- Chandy and Lamport proposed a snapshot algorithm for determining globalstates of distributed systems.
- This algorithm records a set of process and channels as a snapshot for theprocess set. The recorded global state is consistent.

**42. What is the purpose of Venkatesans incremental snapshot algorithm? (NOV/DEC 2021)**

- Venkatesan's algorithms optimize the basic snapshot algorithm to efficiencyrecord repeated snapshot of a distributed system that is required in recoveryalgorithms with synchronous check pointing.

**43. State the role of special marker message.**

- As a prompt for the receiver to save its own state, if it has not already done soand as a means of determining which messages to be included in the channel state.

**44. How the messages are tracked using implicit tracking?**

(i) Tracking of messages that are either

(ii) Already delivered, or

(iii) Guaranteed to be delivered in CO, is performed implicitly.

**45. How the messages are tracked using explicit tracking?**

Tracking of (source, timestamp, destination) information for messages

i. Not known to be delivered

ii. Not guaranteed to be delivered in CO, is done explicitly using the l.Dests field of entries in local logs at nodes and o.Dests field of entries in messages.

iii. Sets li,a.Dests and oi,a.Dests contain explicit information of destinations to which Mi,a

is not guaranteed to be delivered in CO and is not known to be delivered.

**46. Define causal order execution. (Nov/Dec 2020) (Apr/May 2021)**

The causal order establishes that for each participant in the system the events must be seen in the cause-effect order as they have occurred, whereas the Δ-causal order establishes that the events must be seen in the cause-effect order only if the cause has been seen before its lifetime expires.

**47. What do you mean by Synchronous and Asynchronous Execution. (Nov/Dec 2022)**

Synchronous execution means the first task in a program must finish processing before moving on to executing the next task whereas asynchronous execution means a second task can begin executing in parallel, without waiting for an earlier task to finish.

Synchronous distributed systems have the following characteristics: **the time to execute each step of a process has known lower and upper bounds**; each message transmitted over a channel is received within a known bounded time; each process has a local clock whose drift rate from real time has a known bound.

In a distributed system, **communication between elements is inherently asynchronous**. There is no global clock nor consistent clock rate. Each computer processes independently of others. Some computers in the system have fast clock cycles while others have slower clock cycles.

**48. What is meant by Asynchronous programming. (Nov/Dec 2022)**

Asynchronous programming provides opportunities for a program to continue running other code while waiting for a long-running task to complete. The time-consuming task is executed in the background while the rest of the code continues to execute.

Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.

Example: When a photo is loaded and sent on Instagram, the user does not have to stay on the same screen waiting for the photo to finish loading.

**PART B:**

1. **Discuss in detail about Logical time. (or) Elaborate any two logical dock categories in distributed systems withan example. (NOV/DEC 2021)**

**Logical time - Introduction**

- The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems.
- Usually causality is tracked using physical time. However, in distributed sys- tems, it is not possible to have global physical time; it is possible to realize only an approximation of it.
- As asynchronous distributed computations make progress in spurts, it turns out that the logical time, which advances in jumps, is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems.
- Three ways to implement logical time (e.g., scalar time, vector time, and matrix time) that have been proposed to capture causality between events of a distributed computation.
- Causality (or the causal precedence relation) among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation.

The knowledge of the causal precedence relation among the events of processes helps solve a variety of problems in distributed systems. Examples of some of these problems is as follows:

### Distributed algorithms design

- The knowledge of the causal precedence relation among events helps ensure liveness and fairness in mutual exclusion algorithms, helps maintain consistency in replicated databases,
- And helps design correct deadlock detection algorithms to avoid phantom and undetected deadlocks.

### Tracking of dependent events

- In distributed debugging, the knowledge of the causal dependency among events helps construct a consistent state for resuming reexecution.
- In failure recovery, it helps build a checkpoint; in replicated databases, it aids in the detection of file inconsistencies in case of a network partitioning.

### Knowledge about the progress

- The knowledge of the causal dependency among events helps measure the progress of processes in the distributed computation.
- This is useful in discarding obsolete information, garbage collection, and termination detection.

### Concurrency measure

- The knowledge of how many events are causally dependent is useful in measuring the amount of concurrency in a computation.
- All events that are not causally related can be executed concurrently. Thus, an analysis of the causality in a computation gives an idea of the concurrency in the program.
- The concept of causality is widely used by human beings, often unconsciously, in the
- planning, scheduling, and execution of a chore or an enterprise, or in determining the
- infeasibility of a plan or the innocence of an accused.

- In day-to-day life, the global time to deduce causality relation is obtained from loosely synchronized clocks (i.e., wrist watches, wall clocks).
- However, in distributed computing systems, the rate of occurrence of events is several magnitudes higher and the event execution time is several magnitudes smaller.
- Consequently, if the physical clocks are not precisely synchronized, thecausality relation between events may not be accurately captured.
- Net- work Time Protocols, which can maintain time accurate to a fewtens of milliseconds on the Internet, are not adequate to capture the causality relation in distributed systems.
- However, in a distributed computation, generally the progress is madein spurts and the interaction between processes occurs in spurts.
- Consequently, it turns out that in a distributed computation, the causalityrelation between events produced by a program execution and its fundamental monotonicity property can be accurately captured by logical clocks.

- In a system of logical clocks, every process has a logical clock that is advanced using a set of rules.
- Every event is assigned a timestamp and the causality relation between events can be generally inferred from their times- tamps.
- The timestamps assigned to events obey the fundamental monotonicity property; that is, if an event *a* causally affects an event *b*, then the timestamp of *a* is smaller than the timestamp of *b*.

### A framework for a system of logical clocks Definition

A system of logical clocks consists of a time domain $T$ and a logical clock $C$ [19]. Elements of $T$ form a partially ordered set over a relation $<$. This relation is usuallycalled the *happened before* or *causal precedence*. Intuitively, this relation is analogous to the *earlier than* relation provided by the physical time. The logical clock $C$ is a function that maps an event $e$ in a distributed system to an element in the time domain $T$ , denoted as $C(e)$ and called the timestamp of $e$, and is defined as follows:

$$C : H \supseteq \rightarrow T,$$

such that the following property is satisfied:

for two events $e_i$ and $e_j$, $e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j)$.

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

for two events $e_i$ and $e_j$, $e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$, the system of clocks is said to be *strongly consistent*.

### Implementing logical clocks

Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol (set of rules) to update the data structures to ensure the consistency condition.

Each process $p_i$ maintains data structures that allow it the following two capabilities:

- A *local logical clock*, denoted by $lc_i$, that helps process $p_i$ measure its own progress.
- A *logical global clock*, denoted by $gc_i$, that is a representation of process $p_i$'s local view of

the logical global time. It allows this process to assign consistent timestamps to its local events. Typically, $lc_i$ is a part of $gc_i$.

- The protocol ensures that a process's logical clock, and thus its view of the 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.

- Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks.
- However, all logical clock systems implement rules **R1** and **R2** and consequently ensure the fundamental monotonicity property associated with causality.
- Moreover, each particular logical clock system provides its users with some additional properties.

The logical local clock of a process $p_i$ and its local view of the global time aresquashed into one integer variable $C_i$.
Rules **R1** and **R2** to update the clocks are as follows:

**R1** Before executing an event (send, receive, or internal), process $p_i$ executes the following:

$$C_i := C_i + d \qquad (d > 0).$$

In general, every time
**R1** is executed, $d$ can have a different value, and this value may be application- dependent. However, typically $d$ is kept at 1 because this is able to identify the timeof each event uniquely at a process, while keeping the rate of increase of $d$ to its lowest level.

**R2** Each message piggybacks the clock value of its sender at sending time. When a process $p_i$ receives a message with timestamp $C_{msg}$, it executes the following actions:

1. $C_i := max(C_i, C_{msg})$;
2. execute **R1**;
3. deliver the message.

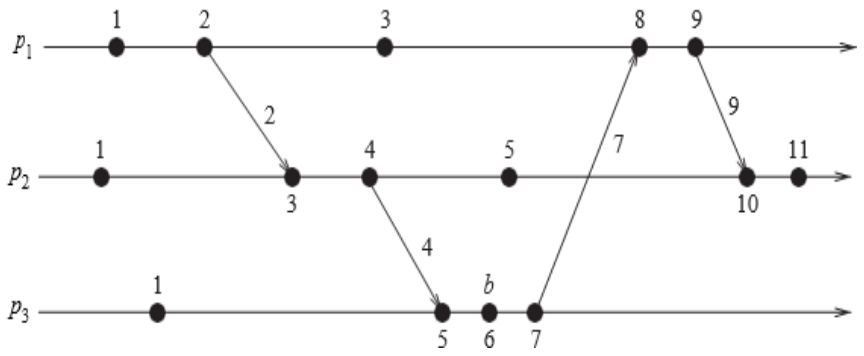



**Figure 3.1** Evolution of scalar time [19].

## Basic properties

**Consistency property**

Scalar clocks satisfy the monotonicity and hence the consistency property:for two events $e_i$ and

$e_j$, $e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j)$.

**Total Ordering**

- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events atdifferent processes may have an identical timestamp.
- (Note that for two events $e1$ and $e2$, $C(e1) = C(e2) \Rightarrow e1 \equiv e2$.) For example,in Figure 3.1, the third event of process $P1$ and the second event of process $P2$ have identical scalar timestamp.
- Thus, a tie-breaking mechanism is needed to order such events. Typically, atie is broken as follows: process identifiers are linearly ordered and a tie among events with identical scalar timestamp is broken on the basis of theirprocess identifiers.
- The lower the process identifier in the ranking, the higher the priority. The timestamp of an event is denoted by a tuple $(t, i)$ where $t$ is its timeof occurrence and $i$ is the identity of the process where it occurred.
- The total order relation $\prec$ on two events $x$ and $y$ with timestamps $(h,i)$ and $(k,j)$, respectively, is defined as follows:

$$x \prec y \iff (h<k \textbf{ or } (h = k \textbf{ and } i < j)).$$

- Since events that occur at the same logical scalar time are independent (i.e., they are not causally related), they can be ordered using anyarbitrarycriterion without violating the causality relation $\rightarrow$. Therefore, a total order is consistent with the causality relation "$\rightarrow$". Note that $x \prec y \Rightarrow x \rightarrow y \vee x \equiv y$.

## Event counting

- If the increment value $d$ is always 1, the scalar time has the following interest- ing property: if event $e$ has a timestamp $h$, then $h-1$ represents the minimum logical duration, counted in units of events, required before producing the event $e$; we call it the height of the event $e$.
- In other words, *h-1* events have been produced sequentially before the event $e$ regardless of the processes that produced these events. For example, in Figure 3.1, five events precede event $b$ on the longest causal path ending at $b$.

## No strong consistency

- The system of scalar clocks is not strongly consistent; that is, for two events $e_i$
- and $e_j$, $C(e_i) < C(e_j)$ $\not\Rightarrow e_i \rightarrow e_j$. For example, in Figure 3.1, the third event
- of process $P1$ has smaller scalar timestamp than the third event of process $P2$.
- However, the former did not happen before the latter. The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one,

resulting in the loss causal dependency information among events at different processes.

- For example, in Figure 3.1, when process $P2$ receives the first message from process $P1$, it updates its clock to 3, forgetting that the timestamp of the latest event at $P1$ on which it depends is 2.

**2. Write in detail about Vector time.**

In the system of vector clocks, the time domain is represented by a set of *n*- dimensional non-negative integer vectors. Each process $p_i$ maintains a vector $vt_i[1..n]$, where $vt_i[i]$ is the local logical clock of $p_i$ and describes the logical time progress at process $p_i$. $vt_i[j]$ rep- resents process $p_i$'s latest knowledge of process $p_j$ local time. If $vt_i[j] = x$, then process $p_i$ knows that local time at process $p_j$ has progressed till $x$. The entire vector $vt_i$ constitutes $p_i$'s view of theglobal logical time and is used to timestamp events.

Process $p_i$ uses the following two rules **R1** and **R2** to update its clock:

**R1** Before executing an event, process $p_i$ updates its local logical time asfollows:

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

**R2** Each message $m$ is piggybacked with the vector clock $vt$ of the sender process at sending time. On the receipt of such a message $(m,vt)$, process $p_i$ executes the following sequence of actions:

1. update its global logical time as follows:$1 \le k \le n : vt_i[k] :=$

   $max(vt_i[k], vt[k]);$

2. execute **R1**;
3. deliver the message $m$.

The timestamp associated with an event is the value of the vector clock of its process when the event is executed. Figure 3.2 shows an example of vector clocks progress withthe increment value $d = 1$. Initially, a vector clock is $[0, 0, 0,. , 0]$.

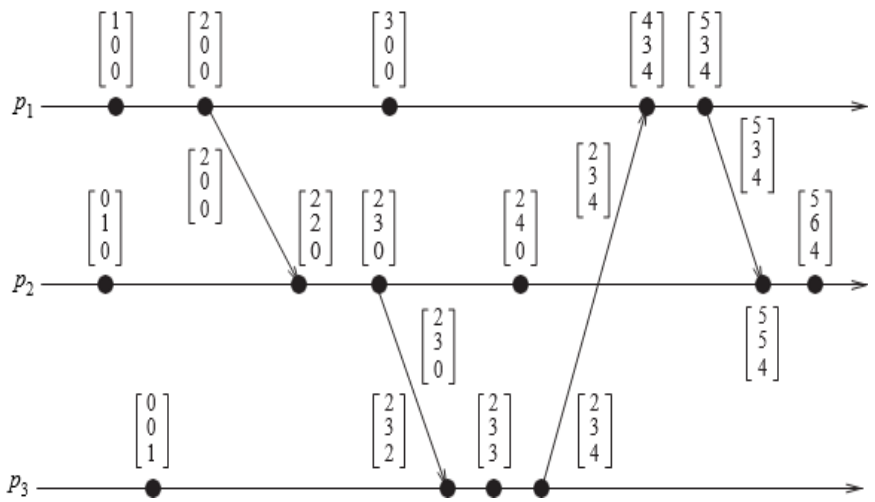



**Figure 3.2** Evolution of vector time [19].

The following relations are defined to compare two vector timestamps, *vh* and *vk*:

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$

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

$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh \equiv vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh).$$

**Basic Properties**

**Isomorphism**

Recall that relation "→" induces a partial order on the set of events that areproduced by a distributed execution. If events in a distributed system are timestamped using a system of vector clocks, we have the following property.

If two events *x* and *y* have timestamps *vh* and *vk*, respectively, then

$$x \rightarrow y \Leftrightarrow vh < vk \; x \equiv y \Leftrightarrow vh \equiv vk.$$

Thus, there is an isomorphism between the set of partially ordered events produced by adistributed computation and their vector timestamps. This is a very powerful, useful, and interesting property of vector clocks.

**Strong consistency**

- The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related. However, Charron–Bost showed that the dimension of vector clocks cannot be less than *n*, the total number of processes in the distributed computation, for this property to hold.

**Event counting**

- If *d* is always 1 in rule **R1**, then the *ith* component of vector clock at process $p_i$, $vt_i[i]$, denotes the number of events that have occurred at $p_i$ until that instant. So, if an event *e* has timestamp *vh*, *vh[j]* denotes the number of events executed by process $p_j$ that causally precede *e*. Clearly, *vh[j]* − 1 represents the total number ofevents that causally precede *e* in the distributed computation.

**3. Explain Different types of Message ordering paradigms in detail. (or) Discuss the purpose of message ordering paradigms and provide example for asynchronous execution communication in detail. What are the four different types of ordering the message? Explain. (NOV/DEC 2021) (Nov/Dec 2022)**

**Message ordering paradigms**

To determine the message behavior, ordering of messages is needed in distributed Systems.

The various forms of message ordering paradigms are

(i) non-FIFO,
(ii) FIFO,
(iii) causal order
(iv) Synchronous order.

### (i)Asynchronous executions (A-execution)

**Definitions**

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

**Explanation:**

- ✓ There cannot exist any causality cycles in any real asynchronous execution because cycles lead to the absurdity that an event causes itself.
- ✓ Although each physical link typically delivers the messages sent on it in FIFO order due to the physical properties of the medium, a logical link may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link.
- ✓ As an example, the mode of ordering at the Network Layer in connectionless networks such as IPv4 is non-FIFO.

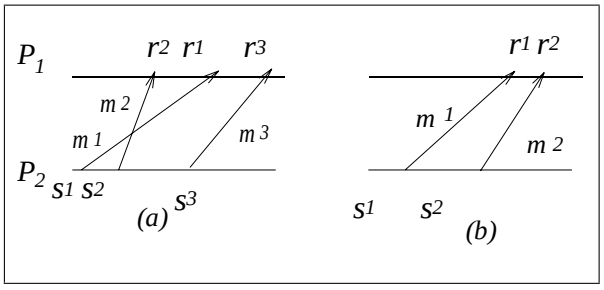



Figure 2.1(a) illustrates an A-execution under non-FIFO ordering.

### (ii) FIFO executions

**Definition**

A FIFO execution is an A-execution in which, for all (s, r) and $(s^t, r^t) \in T$, $(s \sim s^t$ and $r \sim r^t$ and $\tilde{s} \sim s^t)$

$\Rightarrow r \prec r^t$.

**Explanation:**

- ✓ Messages are necessarily delivered in the order in which they are sent in any logical link in the system.
- ✓ The network protocols provide a connection-oriented service at the transport layer.
- ✓ To implement FIFO logical channels over a non-FIFO channel would use a separate numbering scheme to sequence the messages on each logical channel.
- ✓ 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 2.1(b) illustrates an A-execution under FIFO ordering.

### (iii) Causally ordered (CO) executions

**Definition**

A CO execution is an A-execution in which, for all (s, r) and $(s^t, r^t) \in T$, ( $r \sim r^t$ and $s \prec s^t$) $\Rightarrow r \prec r$

**Explanations**

- ✓ If two send events s and $s^t$ are related by causality ordering (not physical time ordering), then a causally ordered execution requires that their corresponding receive events r and $r^t$ occur in the same order at all common destinations.
- ✓ If s and $s^t$ are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false.

**Examples**

- **Figure 2.2(a)** shows an execution that violates CO because $s^1 \prec s^3$ and at the common destination $P_1$, we have $r^3 \prec r^1$.
- **Figure 2.2(b)** shows an execution that satisfies CO. Only $s^1$ and $s^2$ are related by causality but the destinations of the corresponding messages are different.

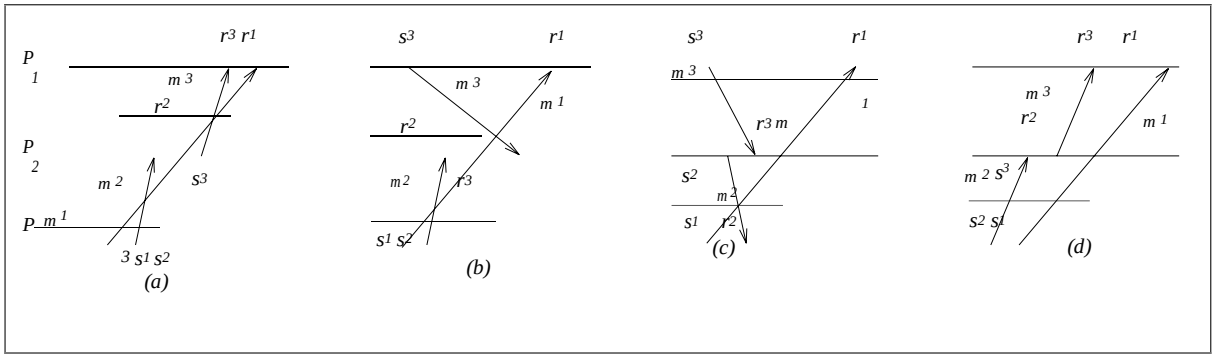



FIG 2.2 illustration of causally ordered executions

- **Figure 2.2(c)** shows an execution that satisfies CO. No send events are related by causality.
- **Figure 2.2(d)** shows an execution that satisfies CO. $s^2$ and $s^1$ are related by causality but the destinations of the corresponding messages are different. Similarly for $s^2$ and $s^3$.

- ✓ Causal order is useful for applications requiring updates to shared data, implementing distributed shared memory, and fair resource allocation
- ✓ To implement CO, we distinguish between the arrival of a message and its delivery.
- ✓ A message m that arrives in the local OS buffer at $P_i$ may have to be delayed until the messages that were sent to $P_i$ causally before m was sent have arrived and are processed by the application.
- ✓ The delayed message m is then given to the application for processing. The event of an application processing an arrived message is referred to as a *delivery* event for emphasis.

**Example**

Figure 2.2(a) shows an execution that violates CO. To enforce CO, message $m^3$ should be kept pending in the local buffer after it arrives at $P_1$, until $m^1$ arrives and $m^1$ is delivered.

**Causal order (CO) for implementations**

**Definition**

If send($m^1$) $\prec$ send($m^2$) then for each common destination d of messages $m^1$ and $m^2$, deliver$_d$ ($m^1$) $\prec$ deliver$_d$ ($m^2$) must be satisfied.

**Explanation**

- ✓ If the definition of causal order is restricted so that $m^1$ and $m^2$ are sent by the same process, then the property degenerates into the FIFO property.
- ✓ In a FIFO execution, no message can be overtaken by another message between the same (sender, receiver) pair of processes.
- ✓ The FIFO property which applies on a per-logical channel basis can be extended globally to give the CO property. In a CO execution, no message can be overtaken by a chain of messages between the same (sender, receiver) pair of processes.

**Example**

Figure 2.2(a) shows an execution that violates CO. Message m1is overtaken by the messages in the chain (m2, m3).

**Message order (MO)**

**Definition**

A MO execution is an A-execution in which, for all (s, r) and ($s^t$,$r^t$ ) $\in T$, $s \prec s^t \Rightarrow \neg(r^t \prec r)$.

**Example**

- Consider any message pair, say $m^1$ and $m^3$ in Figure 2.2(a). $s^1 \prec s^3$ but $\neg(r^3 \prec r^1)$ is false.
- Hence, the execution does not satisfy MO.

**Empty-interval execution**

**Definition**

An execution (E, $\prec$) is an empty-interval (EI) execution if for each pair of events (s, r) $\prec$ T, the open interval set {x E €| s $\prec$ x $\prec$ r} in the partial order is empty.

**Example**

- Consider any message, say $m^2$, in Figure 6.2(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.
- For an empty interval (s, r), there exists some *linear* extension[1] < such that the corresponding interval {x € E |s < x < r} is also empty.
- Two events may be arbitrarily close and can be represented by a vertical arrow in a timing diagram
- An execution E is CO if and only if for each message, there exists *some* space–time diagram in which that message can be drawn as a vertical message arrow.

The following corollary can be derived from the EI characterization above

**Corollary 1** An execution (E, $\prec$) is CO if and only if for each pair of events (s, r) €T and each event e €E,

- weak common past: $e \prec r \Rightarrow \neg(s \prec e)$;
- weak common future: $s \prec e \Rightarrow \neg(e \prec r)$.

**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 occurring instantaneously and atomically.
- ✓ In a timing diagram, the "instantaneous" message communication can be shown by bidirectional vertical message lines.
- ✓ Figure 2.3(a) shows a synchronous execution on an asynchronous system. Figure 2.3(b) shows the equivalent timing diagram with the corresponding instantaneous message communication.

**Causality in a synchronous execution**

**Definition**

The synchronous causality relation on E is the smallest transitive relation that satisfies the following:

S1: If x **occurs** before y at the same process, then $x \sim y$.

S2: If $(s, r) \in T$, then for all $x \in E$, $[(x \sim s \Leftrightarrow x \sim r)$ and $(s \sim x \Leftrightarrow r \sim x)]$.

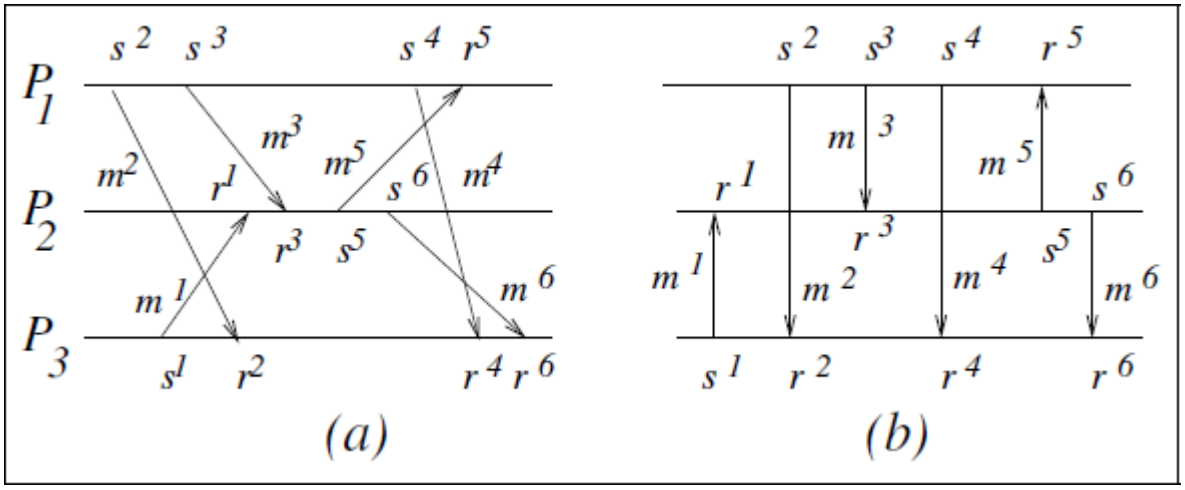
S3: If $x \sim y$ and $y \sim z$, then $x \sim z$.

## Synchronous execution

**Definition** A synchronous execution (or S-execution) is an execution $(E, \sim)$ for which the causality relation $\sim$ is a Partial order.

a. Execution in an asynchronous system

b. Equivalent instantaneous communication




**Time stamping a synchronous execution**

**Definition**

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 $P_i$, if $e_i \prec e^t$ then $T(e) < T(e^t)$.

By assuming that a send event and its corresponding receive event are viewed atomically, i.e., $s(M) \prec r(M)$ and $r(M) \prec s(M)$, it follows that for any events $e_i$ and $e_j$ that are not the send event and the receive event of the same message, $e_i \prec e_j \Rightarrow T(e_i) < T(e_j)$.

**Asynchronous execution with synchronous communication**

***4.*** **Discuss elaborately Asynchronous execution with synchronous communication.**

- ✓ When all the communication between pairs of processes is using synchronous send and receive primitives, the resulting order is synchronous order.
- ✓ Distributed algorithm designed to run correctly on asynchronous systems (called *A-executions*) may not run correctly on synchronous systems. An algorithm that runs on an asynchronous system may *deadlock* on a synchronous system.

**Executions realizable with synchronous communication (RSC)**

- ✓ An execution can be modeled (using the interleaving model) as a feasible schedule of the events to give a total order that extends the partial order $(E, \prec)$.
- ✓ 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 in this linear extension.
- ✓ Such an A-execution can be realized under synchronous communication and is called a realizable with synchronous communication (RSC) execution.

**Non-separated linear extension**

**Definition**

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.

**Examples**

- Figure 2.2(d): $(s^2,r^2, s^3,r^3, s^1,r^1)$ is a linear extension that is non-separated. $(s^2, s^1,r^2, s^3,r^3, s^1)$ is a linear extension that is separated.
- Figure 2.3(b): $(s^1,r^1, s^2,r^2, s^3,r^3, s^4,r^4, s^5,r^5, s^6,r^6)$ is a linear extension that is non-separated. $(s^1, s^2,r^1,r^2, s^3, s^4,r^4,r^3, s^5, s^6,r^6,r^5)$ is a linear extension that is separated.

**RSC execution**

**Definition**

- ✓ An A-execution (E, ≺ ) is an RSC execution if and only if there exists a non-separated linear extension of the partial order (E, ≺).
- ✓ In the non-separated linear extension, if the adjacent send event and its corresponding receive event are viewed atomically, then that pair of events shares a common past and a common future with each other.
- ✓ Characterization of the execution in terms of a graph structure called a *crown*; the crown leads to a feasible test for a RSC execution.

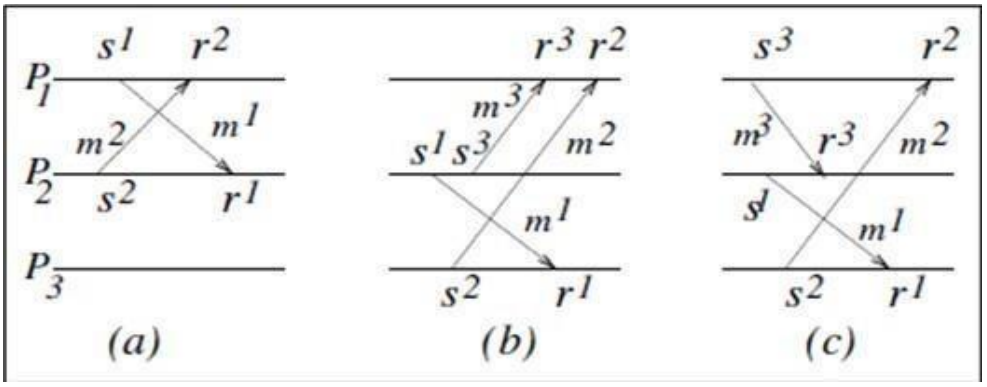



Figure 2.3 Illustration of asynchronous execution and of crowns
a.crown of size 2 b.another crown ofsize 2 c.crown of size 3

**Crown**

**Definition**

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

**Examples**

- Figure 2.3(a): The crown is $((s^1,r^1), (s^2,r^2))$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$. This execution represents the program execution in Figure 6.4.
- Figure 2.3(b): The crown is $((s^1,r^1), (s^2,r^2))$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$.
- Figure 2.3(c): The crown is $((s^1,r^1), (s^3,r^3), (s^2,r^2))$ as we have $s^1 \prec r^3$ and $s^3 \prec r^2$ and $s^2 \prec r^1$.
- Figure 2.2(a): The crown is $((s^1,r^1), (s^2,r^2), (s^3,r^3))$ as we have $s^1 \prec r^2$ and $s^2 \prec r^3$ and $s^3 \prec r^1$.

In a crown, the send event $s^i$ and receive event $r^{i+1}$ may lie on the same process or may lie on different processes

**We can also make the following observations:**

- In an execution that is not CO (see the example in Figure 2.2(a)), there must exist pairs (s, r) and $(s^t, r^t)$ such that $s \prec r^t$ and $s^t \prec r$. It is possible to generalize this to state that a non-CO execution must have a crown of size at least 2.
- CO executions that are not synchronous, also have crowns, e.g., the execution in

Figure2.2 (b) has a crown of size 3.

**Example**

By drawing the directed graph $(T, >\rightarrow)$ for each of the executions in Figures 2.2, 2.3, and 2.5, it can be seen that the graphs for Figures 2.2(d) and Figure2 are acyclic. The other graphs have a cycle.

1. Define the $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ relation on messages in the execution $(E, \prec)$ as follows. Let $\hookrightarrow ([s, r], [s', r'])$ iff $s \prec r'$. Observe that the condition $s \prec r'$ (which has the form used in the definition of a crown) is implied by all the four conditions: (i) $s \prec s'$, or (ii) $s \prec r'$, or (iii) $r \prec s'$, and (iv) $r \prec r'$.
2. Now define a *directed* graph $G_{\hookrightarrow} = (\mathcal{T}, \hookrightarrow)$, where the vertex set is the set of messages $\mathcal{T}$ and the edge set is defined by $\hookrightarrow$.
   Observe that $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ is a partial order iff $G_{\hookrightarrow}$ has no cycle, i.e., there must not be a cycle with respect to $\hookrightarrow$ on the set of corresponding $(s, r)$ events.
3. Observe from the defn. of a crown that $G_{\hookrightarrow}$ has a directed cycle iff $(E, \prec)$ has a crown.

**Crown criterion Theorem**

The crown criterion states that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, if and only if it contains no crown.

**Example**

- ✓ Using the directed graph $(T, >\rightarrow)$ for each of the executions in Figures 2.2, 2.3(a), and 2.3, it can be seen that the executions in Figures 2.2(d) and Figure 2.3(a) are RSC. The others are not RSC.
- ✓ Although checking for a non-separated linear extension of (E, $\prec$) has exponential cost, checking for the presence of a crown based on the message scheduling test of Figure 2.4 can be performed in time that is linear in the number of communication events
- ✓ An execution is not RSC and its graph G$>\rightarrow$ contains a cycle if and only if in the corresponding space–time diagram, it is possible to form a cycle by
  - moving along message arrows in either direction, but
  - always going left to right along the time line of any process.
- ✓ As an RSC execution has a non-separated linear extension, it is possible to assign scalar timestamps to events, as it was assigned for a synchronous execution, as follows.

**Timestamps for a RSC execution**

**Definition**

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

- for any message M, T(s(M)) = T(r(M));
- for each (a, b) in (E $\times$E)\$T$, a $\prec$ b $\Rightarrow$T(a) < T(b).

From the acyclic message scheduling criterion and the times- tamping property above, it can be observed that an A-execution is RSC if and only if its timing diagram can be drawn such that all the message arrows are vertical.

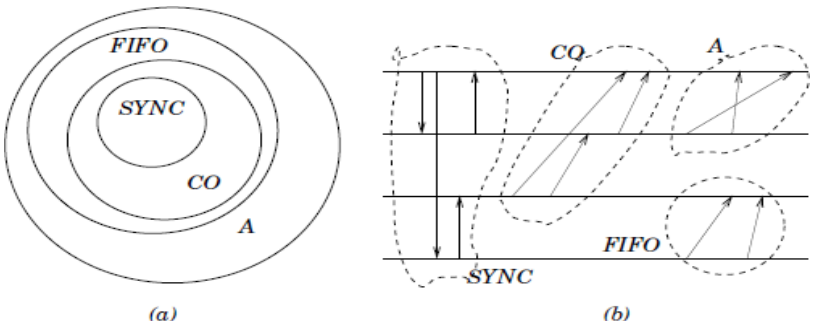


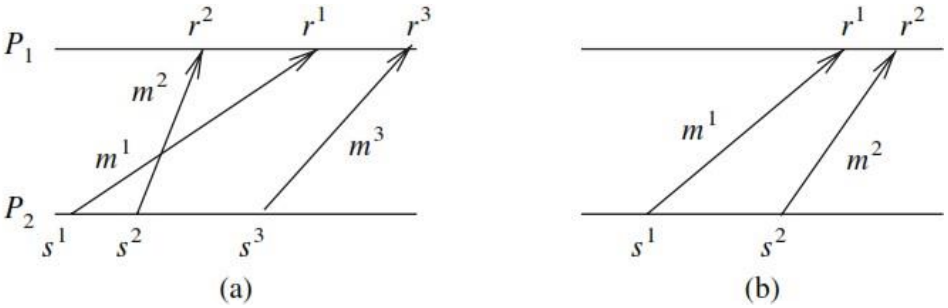

Figure 2.4 Hierarchy of execution classes a. Venn diagram b. example executions

**Hierarchy of ordering paradigms**

Let *SYNC* (or *RSC*), *CO*, *FIFO*, and *A* denote the set of all possible executions ordered by synchronous order, causal order, FIFO order, and non- FIFO order, respectively.
We 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 2.7(a), and example executions of each class are shown sid by-side in Figure 2.7(b).
- Figure 6.1(a) shows an execution that belongs to *A* but not to *FIFO*. Figure 6.2(a) shows an execution that belongs to *FIFO* but not to *CO*. Figures 6.2(b) and (c) show executions that belong to *CO* but not to *RSC*.

**Figure 6.1** 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.




- The above hierarchy implies that some executions belonging to a class X will not belong to any of the classes included in X. Thus, there are more restrictions on the possible message orderings in the smaller classes. Hence, we informally say that the included classes have less concurrency. The degree of concurrency is most in *A* and least in *SYNC*.

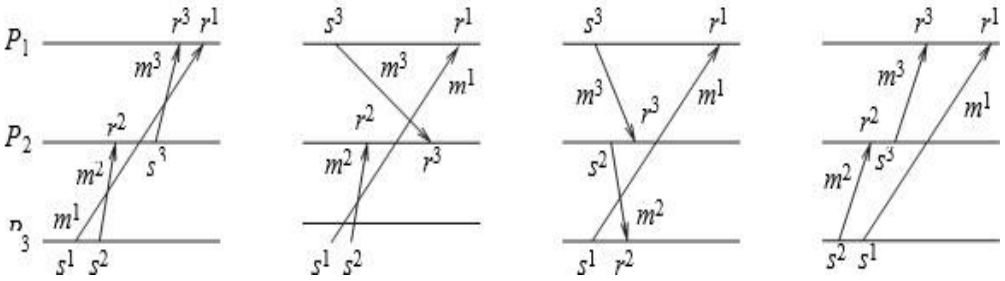



**Figure 6.2** Illustration of causally ordered executions. (a) Not a CO execution. (b), (c), and (d) CO executions.

- 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. This is because synchronous order offers the most simplicity due to the restricted number of possibilities, whereas non-FIFO order offers the greatest difficulties because it admits a much larger set of possibilities that the developer and verifier need to account for.

**Simulations**

**Asynchronous programs on synchronous systems**

- A-execution can be run using synchronous communication primitives if and only if it is an RSC execution.
- 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.

**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 an acknowledgment; after the ack is received, the synchronous send primitive completes.

## Synchronous program order on an asynchronous system

### 5. Discuss how synchronous program order implemented in an asynchronus system

**Non-determinism**

The distributed programs are *deterministic*, i.e., repeated runs of the same program will produce

the same partial order. **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. The receive calls in most of the algorithms in are non-deterministic in this sense – the receiver is willing to perform a rendezvous with any willing and ready sender.
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, there results a deadlock, similar to the one in Figure 2.4.
- ✓ However, there is no semantic dependency between the send and the immediately following receive at each of the processes.
- ✓ If the receive call at one of the processes can be scheduled before the send call, then there is no deadlock.

**Rendezvous**

- ✓ One form of group communication is called multiway rendezvous, which is a synchronous communication among an arbitrary number of asynchronous processes.
- ✓ All the processes involved "meet with each other," i.e., communicate "synchronously" with each other at one time.
- ✓ Support for binary rendezvous communication was first provided by programming languages such as CSP and Ada.

In these languages, the repetitive command (the $*$ operator) over the alternative command (the $\|$ operator) on multiple guarded commands (each having the form $Gi \longrightarrow CLi$) is used, as follows:

$$*_{\varphi}[G1 \longrightarrow CL1 \quad \| \; G2 \longrightarrow CL2 \quad \| \cdots \| \; Gk \longrightarrow CLk].$$

- ✓ Each communication command may be a part of a guard Gi, and may also appear within the statement block CLi.
- ✓ A guard Gi is a boolean expression. If a guard Gi evaluates to true then CLi is said to be enabled, otherwise CLi is said to be disabled.
- ✓ A send command of local variable x to process Pk is denoted as
- ✓ "x !Pk." A receive from process Pk into local variable x is denoted as "Pk ? x."

Some typical observations about synchronous communication under *binary rendezvous* are as follows:

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

✓ *Binary rendezvous* explicitly assumes that multiple send and receives are enabled. Any send or receive event that can be "matched" with the corresponding receive or send event can be scheduled. This is dynamically scheduling the ordering of events and the partial order of the execution.

## Algorithm for binary rendezvous

At each process, there is 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 problem reduces to one of scheduling messages satisfying the following constraints:

- Schedule on-line, atomically, and in a distributed manner, i.e., the scheduling code at any process does not know the application code of other processes.
- 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 over others during scheduling, and

(ii) efficiency, i.e., using as few messages as possible, and involving as low a time overhead as possible.

We now outline a simple algorithm by Bagrodia that makes the following assumptions:

(i) Receive commands are forever enabled from all processes.

(ii) A send command, once enabled, remains enabled until it completes, i.e., it is not possible that a send command gets disabled before the send is executed.

(iii) To prevent deadlock, process identifiers are used to introduce asymmetry to break potential crowns that arise.

(iv) Each process attempts to schedule only one send event at any time.

The algorithm illustrates how crown-free message scheduling is achieved on-line.

The message types used are:

(i) M,

(ii) *ack*(M),

(iii) *request*(M),

(iv) *permission*(M).

A process blocks when it knows that it can successfully syn- chronize the current message with the partner process. Each process maintains a queue that is processed in FIFO order only when the process is unblocked. When a process is blocked waiting for a particular message that it is currently synchronizing, any other message that arrives is queued up.

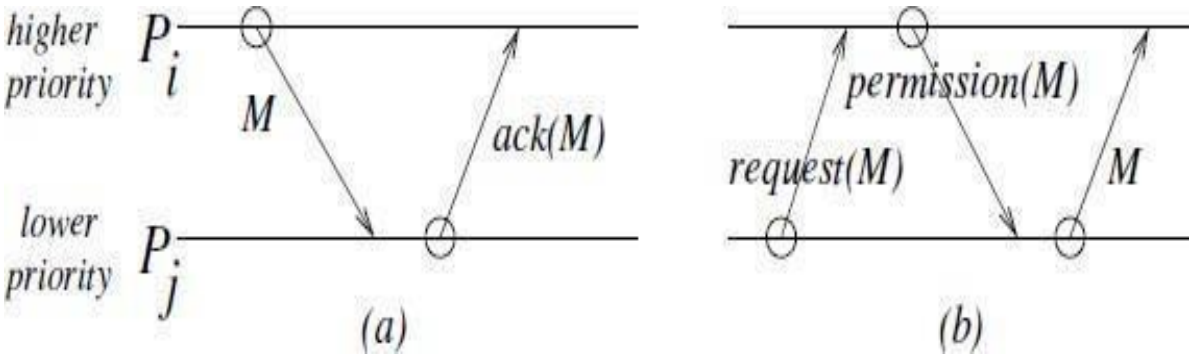
Execution events in the synchronous execution are only the *send* of the message M and *receive* of the message M. The send and receive events for the other message types– *ack*(M), *request*(M), and *permission*(M) which are control messages – are under the covers, and are not included in the synchronous execution.

The messages request(M), ack(M), and permission(M) use M's unique tag; the message M is not included in these messages.

We use capital SEND(M) and RECEIVE(M) to denote the primitives in the application execution, the lower case send and receive are used for the control messages.

The key rules to prevent cycles among the messages are summarized as follows:

- To send to a lower priority process, messages M and *ack*(M) are involved in that order. The sender issues *send*(M) and blocks until *ack*(M) arrives.Thus, when sending to a lower priority process, the sender blocks waiting for the partner process to synchronize and send an acknowledgement.
- To send to a higher priority process, messages *request*(M), *permission*(M),and M are involved, in that order. The sender issues *send*(*request*(M)), does not block, and awaits permission. When *permission*(M) arrives, the sender issues *send*(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 $M^t$ message (from a higher priority processes) and *request*($M^t$ ) 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. $P_i$ seeks permission from $P_j$ by executing *send*(*request*(M)). // to avoid deadlock in which cyclically blocked processes queue

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

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

(ii) If a *request*($M^t$) arrives from a lower priority process $P_k$, $P_i$ exe- cutes *send*(*permission*($M^t$)) to $P_k$ and blocks waiting for the mes- sage $M^t$. When $M^t$ arrives, the RECEIVE($M^t$) event is executed.

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

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

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

*(4) Message* M *arrival at* ***Pi*** *from a higher priority process* ***Pj*** *:*

At the time a message *M* is processed by $P_i$, process $P_i$ executes RECEIVE(M) (which is

assumed to be always enabled) and then *send*(*ack*(M)) to 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).

**Algorithm**

- A simplified implementation of synchronous order. Code shown is for process Pi , $1 \leq i \leq n$.
- Thus, when sending to a higher priority process, the sender asks the higher priority process via the request(M) to give permission to send.
- When the higher priority process gives permission to send, the higher priority process, which is the intended receiver, blocks.
- In either case, a higher priority process blocks on a lower priority process. So cyclic waits are avoided.
- In more detail, a cyclic wait is prevented because before sending a mes- sage 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 $M^t$ from a higher priority process arrives, it is processed by a receive (assuming receives are always enabled) and *ack*($M^t$ ) is returned. Thus, a cyclic wait is prevented.
2. Also, while waiting for this permission, if a *request*($M^t$ ) from a lower priority process arrives, a *permission*($M^t$) is returned and the process blocks until $M^t$ actually arrives.

**6. Explain group communication in detail.**

- A message broadcast is the sending of a message to all members in the distributed system. The notion of a system can be confined only to those sites/processes participating in the joint application.
- Refining the notion of *broadcasting*, there is *multicasting* wherein a message is sent to a certain subset, identified as a *group*, of the processes in the system. At the other extreme is *unicasting*, which is the familiar point-to-point message communication.
- Broadcast and multicast support can be provided by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information.

However, the hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features such as the following:

· 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, 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 several scenarios such as in a large system (e.g., on-line reservation or Internet banking systems) where client processes are short-lived and in large numbers.
- Two popular orders for the delivery of messages were proposed in the context of group communication: *causal order* and *total order*.

## Causal order (CO)

**7. How Casual Order is implemented in Synchronization (or) Illustrate the necessary and sufficient conditions for casual ordering. Elucidate on the Total and Causal order in the Distributed System with a neat diagram. (Nov/Dec 2020) (Apr/May 2021) (Nov/Dec 2022)**

- Causal order has many applications such as updating replicated data, allocating requests in a fair manner, and synchronizing multimedia streams.
- Consider Figure 2.8(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. (In this case, CO is satisfied.) However, this may happen at some, all, or none of the replicas.

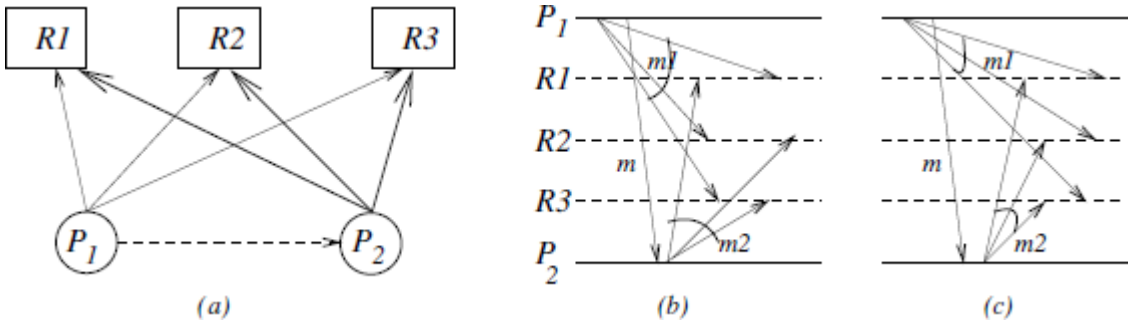



Figure 2.8 Updates to object replicas are issued by two processes

- Figure 2.8 (b) shows that R1 sees P2's update first, while R2 and R3 see P1's update first. Here, CO is violated.
- Figure 2.8 (c) shows that all replicas see P2's update first. However, CO is still violated. If message m did not exist as shown, then the executions shown in Figure 6.11(b) and (c) would satisfy CO.
- Given a system with FIFO channels, causal order needs to be explicitly enforced by a protocol.

The following two criteria must be met by a causal ordering protocol:

**Safety**

- In order to prevent causal order from being violated, a message M that arrives at a process may need to be buffered until all system wide messages sent in the causal past of the send(M) event to that same destination have already arrived.

- The arrival of a message is transparent to the application process. The delivery event corresponds to the *receive* event in the execution model.

**Liveness**

A message that arrives at a process must eventually be delivered to the process.

## The Raynal–Schiper–Toueg algorithm

- Each message M should carry a log of all other messages, or their identifiers, sent causally before M's send event, and sent to the same destination dest(M).
- This log can then be examined to ensure whether it is safe to deliver a message.
- All algorithms aim to reduce this log overhead, and the space and time overhead of maintaining the log information at the processes.
- Consider Figure 6.11(a), which shows two processes $P_1$ and $P_2$ 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 $P_2$ issues its update causally after $P_1$ issued its update, then $P_2$'s update should be seen by the replicas after they see $P_1$'s update, in order to preserve the semantics
- An optimal CO algorithm stores in local message logs and propagates on messages, information of the form "d is a destination of M" about a message M sent in the causal past, *as long as* and *only as long as*:

- (*Propagation Constraint I*) it is not known that the message M is delivered to d, and (*Propagation Constraint II*) it is not known that a message has been sent to d in the causal future of Send(M), and hence it is not guaranteed using a reasoning based on transitivity that the message M will be delivered to d in CO.
- The Propagation Constraints also imply that if either (I) or (II) is false, the information "$d \in M.Dests$" must not be stored or propagated, even to remember that (I) or (II) has been falsified. Stated differently, the information "$d \in M_{i,a}.Dests$" must be available in the causal future of event $e_{i,a}$, but:

. not in the causal future of $Deliver_d(M_{i,a})$, and

. not in the causal future of $e_{k,c}$, where $d \in M_{k,c}$.

. Dests and there is no other message sent causally between $M_{i,a}$ and $M_{k,c}$ to the same destination d.

- In the causal future of $Deliver_d(M_{i,a})$, and $Send(M_{k,c})$, the information is redundant; elsewhere, it is necessary.
- Additionally, to maintain optimality, no other information should be stored, including information about what messages have been delivered.
- As information about what messages have been delivered is necessary for the Delivery Condition, this information is inferred using a set-operation based logic.

The Propagation Constraints are illustrated with the help of Figure 6.12. The message M is sent by process i at event e to process d. The information"d € M.Dests":

- must exist at e1 and e2 because (I) and (II) are true;
- must not exist at e3 because (I) is false;
- must not exist at e4, e5, e6 because (II) is false;
- must not exist at e7, e8 because (I) and (II) are false.

Information about messages

(i) not known to be delivered and

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

- The information must be deleted as soon as either (i) or (ii) becomes false.
- The key problem in designing an optimal CO algorithm is to identify the events at which (i) or (ii) becomes false. Information about messages already delivered and messages guaranteed to be delivered in CO is *implicitly* tracked without storing or propagating it, and is derived from the explicit information. Such implicit information is used for determining when (i) or (ii) becomes false for the explicit information being stored or carried in messages.

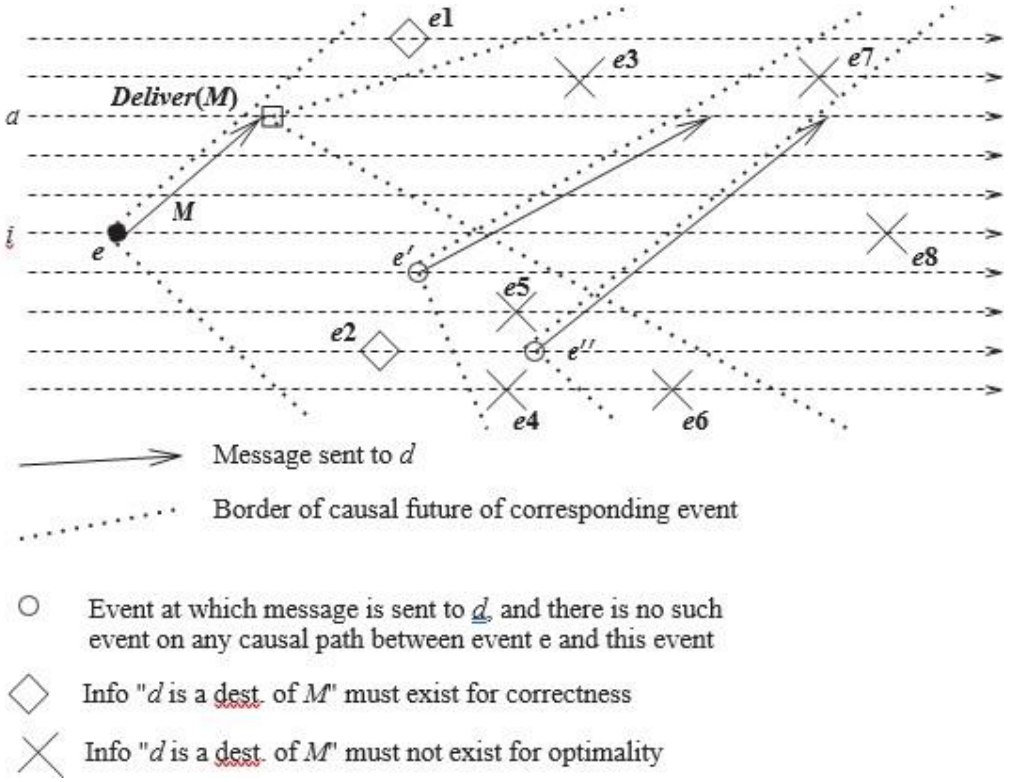



**Figure 6.12** Illustrating the necessary and sufficient conditions for causal ordering [21].

- Procedure SND is executed atomically. Procedure RCV is executed atomically except for a possible interruption in line 2a where a non-blocking wait is required to meet the Delivery Condition.
- pseudo-code can be restructured to complete the processing of each invocation of SND and RCV procedures in a single pass of the data structures, by always maintaining the data structures sorted row–major and then column–major.

### 1. Explicit tracking

Tracking of (source, timestamp, destination) information for messages

(i) not known to be delivered and

(ii) not guaranteed to be delivered in CO, is done explicitly using the l.Dests field of entries in local logs at nodes and o.Dests field of entries in messages.

(iii) Sets $l_{i,a}$.Dests and $o_{i,a}$.Dests contain explicit information of destinations to which $M_{i,a}$ is not guaranteed to be delivered in CO and is not known to be delivered.

The information about "d € $M_{i,a}$.Dests" is propagated up to the earliest events on all causal paths from (i, a) at which it is known that $M_{i,a}$ is delivered to d or is guaranteed to be delivered to d in CO.

**2. Implicit tracking**

Tracking of messages that are either

(i) already delivered, or

(ii) guaranteed to be delivered in CO

The information about messages

(i) already delivered or

(ii) guaranteed to be delivered in CO is deleted and not propagated because it is redundant as far as enforcing CO is concerned

(iii) . However, it is useful in determining what information that is being carried in other messages and is being stored in logs at other nodes has become redundant and thus can be purged.

(iv) The semantics are implicitly stored and propagated. This information about messages that are

(i) already delivered or

(ii) guaranteed to be delivered in

**(local variables)**

$clock_j \longleftarrow 0$; // local counter clock at node $j$

$SR_j[1 \ldots n] \longleftarrow \overline{0}$; // $SR_j[i]$ is the timestamp of last msg. from $i$ delivered to $j$

$LOG_j = \{(i, clock_j, Dests)\} \longleftarrow \{\forall i, (i, 0, \emptyset)\}$;

// Each entry denotes a message sent in the causal past, by $i$ at $clock_j$. *Dests* is the set of remaining destinations
// for which it is not known that $M_{i,clock_j}$ (i) has been delivered, or (ii) is guaranteed to be delivered in CO.

**SND: $j$ sends a message M to Dests:**

1. $clock_j \longleftarrow clock_j + 1$;
2. **for all** $d \in M.Dests$ **do**:
   $O_M \longleftarrow LOG_j$; // $O_M$ denotes $O_{M_{j,clock_j}}$
   **for all** $o \in O_M$, modify $o.Dests$ as follows:
   **if** $d \notin o.Dests$ **then** $o.Dests \longleftarrow (o.Dests \setminus M.Dests)$;
   **if** $d \in o.Dests$ **then** $o.Dests \longleftarrow (o.Dests \setminus M.Dests) \bigcup \{d\}$;
   // Do not propagate information about indirect dependencies that are
   // guaranteed to be transitively satisfied when dependencies of $M$ are satisfied.
   **for all** $o_{s,t} \in O_M$ **do**
   **if** $o_{s,t}.Dests = \emptyset \bigwedge (\exists o'_{s,t'} \in O_M \mid t < t')$ **then** $O_M \longleftarrow O_M \setminus \{o_{s,t}\}$;
   // do not propagate older entries for which *Dests* field is $\emptyset$
   **send** $(j, clock_j, M, Dests, O_M)$ to $d$;
3. **for all** $l \in LOG_j$ **do** $l.Dests \longleftarrow l.Dests \setminus Dests$;
   // Do not store information about indirect dependencies that are guaranteed
   // to be transitively satisfied when dependencies of $M$ are satisfied.
   Execute *PURGE_NULL_ENTRIES*($LOG_j$); // purge $l \in LOG_j$ if $l.Dests = \emptyset$

(iii)

4. $LOG_i \longleftarrow LOG_i \bigcup \{(j, clock_j, Dests)\}$.

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

1. // Delivery Condition; ensure that messages sent causally before M are delivered.
   **for all** $o_{m,t_m} \in O_M$ **do**
   **if** $j \in o_{m.t_m}.Dests$ **wait until** $t_m \leq SR_j[m]$;
2. Deliver M; $SR_j[k] \longleftarrow t_k$;
3. $O_M \longleftarrow \{(k, t_k, Dests)\} \bigcup O_M$;
   **for all** $o_{m,t_m} \in O_M$ **do** $o_{m,t_m}.Dests \longleftarrow o_{m,t_m}.Dests \setminus \{j\}$;
   // delete the now redundant dependency of message represented by $o_{m,t_m}$ sent to $j$
4. // Merge $O_M$ and $LOG_j$ by eliminating all redundant entries.
   // Implicitly track "already delivered" & "guaranteed to be delivered in CO" messages.
   **for all** $o_{m,t} \in O_M$ **and** $l_{s,t'} \in LOG_j$ **such that** $s = m$ **do**
   **if** $t < t' \bigwedge l_{s,t} \notin LOG_j$ **then** mark $o_{m,t}$;
   // $l_{s,t}$ had been deleted or never inserted, as $l_{s,t}.Dests = \emptyset$ in the causal past
   **if** $t' < t \bigwedge o_{m,t'} \notin O_M$ **then** mark $l_{s,t'}$;
   // $o_{m,t'} \notin O_M$ because $l_{s,t'}$ had become $\emptyset$ at another process in the causal past
   Delete all marked elements in $O_M$ and $LOG_j$ ; // delete entries about redundant information
   **for all** $l_{s,t'} \in LOG_j$ **and** $o_{m,t} \in O_M$, **such that** $s = m \bigwedge t' = t$ **do**
   $l_{s,t'}.Dests \longleftarrow l_{s,t'}.Dests \bigcap o_{m,t}.Dests$; // delete destinations for which Delivery
   // Condition is satisfied or guaranteed to be satisfied as per $o_{m,t}$
   Delete $o_{m,t}$ from $O_M$; // information has been incorporated in $l_{s,t'}$
   $LOG_j \longleftarrow LOG_j \bigcup O_M$; // merge nonredundant information of $O_M$ into $LOG_j$
5. $PURGE\_NULL\_ENTRIES(LOG_j)$. // Purge older entries $l$ for which $l.Dests = \emptyset$

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

**for all** $l_{s,t} \in Log_j$ **do**

**if** $l_{s,t}.Dests = \emptyset \bigwedge (\exists l'_{s,t'} \in Log_j \mid t < t')$ **then** $Log_j \longleftarrow Log_j \setminus \{l_{s,t}\}$.

**Multicasts $M_{5,1}$ and $M_{4,2}$**

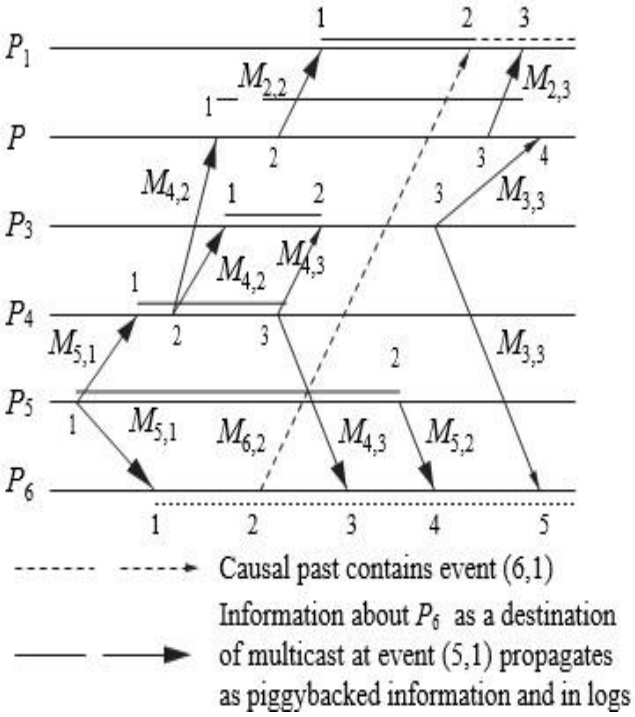
- Message $M_{5,1}$ sent to processes P4 and P6 contains the piggybacked infor- mation "$M_{5,1}$.Dests = {P4, P6}." Additionally, at the send event (5, 1), the information "$M_{5,1}$.Dests = {P4, P6}" is also inserted in the local log Log5.
- When $M_{5,1}$ is delivered to P6, the (new) piggybacked information "P4 € $M_{5,1}$.Dests" is stored in Log6 as
- "$M_{5,1}$.Dests = {P4}"; information about "P6€ $M_{5,1}$.Dests," which was needed for routing, must *not* be stored in Log6 because of constraint I.
- Symmetrically, when $M_{5,1}$ is delivered to process P4 at event (4, 1), *only* the new piggybacked information
- "P6 € $M_{5,1}$.Dests" is inserted in Log4 as "$M_{5,1}$.Dests = {P6}," which is later propagated

during multicast M4,2.

**Multicast *M*4,3**

At event (4, 3), the information "P6 € M5,1.Dests" in Log4 is propagated on multicast M4,3 only to process P6 to ensure causal delivery using the Delivery Condition. The piggybacked information on message M4,3 sent to process P3 must not contain this information because of constraint II.

**Figure 6.13** An example to illustrate the propagation constraints [6].




| Message to dest. | Piggybacked $M_{5,1}$.Dests |
|---|---|
| $M_{5,1}$ to $P_2$ $P_4$ $P_6$ | {$P_2$ $P_4$ $P_6$} |
| $M_{4,2}$ $P_2$, $P_3$ | {$P_6$} |
| $M_{2,2}$ to $P_1$ | {$P_6$} |
| $M_{6,2}$ to $P_1$ | {$P_6$} |
| $M_{4,3}$ to $P_6$ | {$P_6$} |
| $M_{4,3}$ to $P_3$ | {} |
| $M_{5,2}$ to $P_6$ | {$P_4$,$P_6$} |
| $M_{2,3}$ to $P_1$ | {$P_6$} |
| $M_{3,3}$ to $P_{2,6}$ | {} |

**Learning implicit information at P2 and P3**

- When message M4,2 is received by processes P2 and P3, they insert the (new) piggybacked information in their local logs, as information "M5,1.Dests ={P6}."
- They both continue to store this in Log2 and Log3 and propagate this information on multicasts until they "learn" at events (2, 4) and (3, 2) on receipt of messages M3,3 and M4,3, respectively, that any future message is guaranteed to be delivered in causal order to process P6, w.r.t. M5,1 sent to P6. Hence by constraint II, this information must be deleted from Log2 and Log3.
- The logic by which this "learning" occurs is as follows:
  When M4,3 with piggybacked information "M5,1.Dests = φ" is received by P3 at (3, 2), this is inferred to be valid current implicit information about multicast. M5,1 because the log Log3 already contains explicit information"P6 €M5,1.Dests" about that multicast. Therefore, the explicit information in Log3 is inferred to be old and must be deleted to achieve optimality.
  M5,1.Dests is set to φ . Log3
  The logic by which P2 learns this implicit knowledge on the arrival of M3,3 is identical.

**Processing at P6**

- Recall that when message M5,1 is delivered to P6, only "M5,1.Dests = {P4}" is added to Log6. Further, P6 propagates only "M5,1.Dests = {P4}" (from Log6) on message M6,2, and this conveys the current implicit information "M5,1 has been delivered to P6," by its very

absence in the explicit information.

- When the information "P6 € M5,1.Dests" arrives on M4,3, piggybacked as "M5,1.Dests = {P6}," it is used only to ensure causal delivery of M4,3 using the Delivery Condition, and is not inserted in Log6 (constraint I) – further, the presence of "M5,1.Dests = {P4}" in Log6 implies the implicit information that M5,1 has already been delivered to P6. Also, the absence of P4 in M5,1.Dests in the explicit piggybacked information implies the implicit information that M5,1 has been delivered or is guaranteed to be delivered

**Processing at P1**

We have the following processing:

- When M2,2 arrives carrying piggybacked information "M5,1.Dests = {P6}," this (new) information is inserted in Log1.
- When M6,2 arrives with piggybacked information "M5,1.Dests = {P4}," P1 "learns" implicit information "M5,1 has been delivered to P6" by the very absence of explicit information "P6 ∈ M5,1.Dests" in the piggybacked information, and hence marks information "P6 € M5,1.Dests" for dele- tion from Log1.
- Simultaneously, "M5,1.Dests = {P6}" in Log1 implies the implicit information that M5,1 has been delivered or is guaranteed to be delivered in causal order to P4. Analogously, the information"P6€M5,1.Dests" piggybacked on M2,3,which arrives at P1, is inferred to be outdated (and hence ignored) using the implicit knowledge derived from "M5,1.Dests = €" in Log1.

## Total order

### 8. How Total order is implemented in synchronization.

**Definition**

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

**Example**

The execution in Figure 2.11(b) does not satisfy total order. Even if the message m did not exist, total order would not be satisfied. The execution in Figure 2.11(c) satisfies total order.

**Centralized algorithm for total order**

- Assuming all processes broadcast messages, the centralized solution shown in Algorithm 6.4 enforces total order in a system with FIFO channels. Each process sends the message it wants to broadcast to a centralized process, which simply relays all the messages it receives to every other process over FIFO channels.
- It is straightforward to see that total order is satisfied. Furthermore, this algorithm also satisfies causal message order.

(1) When process $P_i$ wants to multicast a message M to group G: (1a) **send** M(i, G) to central

coordinator.

(2) When M(i, G) arrives from $P_i$ at the central coordinator:

(2a) **send** M(i, G) to all members of the group G.

(3) When M(i, G) arrives at $P_j$ from the central coordinator: (3a) **deliver** M(i, G) to the application.

```
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 P_i, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts)                               // Phase 2 message sent by P_j, with revised timestamp, to P_i
FINAL_TS(i, tag, ts)                                     // Phase 3 message sent by P_i, with final timestamp

(1) When process P_i wants to multicast a message M with a tag tag:
(1a) clock = clock + 1;
(1b) send REVISE_TS(M, i, tag, clock) to all processes;
(1c) temp_ts = 0;
(1d) await PROPOSED_TS(j, i, tag, ts_j) from each process P_j;
(1e) ∀j ∈ N, do temp_ts = max(temp_ts, ts_j);
(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 P_j:
(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 P_j.
(3) When FINAL_TS(j, tag, clk) arrives from P_j:
(3a) Identify entry Q_entry(tag) in temp_Q, corresponding to tag;
(3b) mark q_tag as deliverable;
(3c) Update Q_entry.timestamp to clk and re-sort temp_Q based on the timestamp field;
(3d) if head(temp_Q) = Q_entry(tag) then
(3e)     move Q_entry(tag) from temp_Q to delivery_Q;
(3f)     while head(temp_Q) is deliverable do
(3g)         move head(temp_Q) from temp_Q to delivery_Q.
(4) When P_i removes a message (M, tag, j, ts, deliverable) from head(delivery_Q_i):
(4a) clock = max(clock, ts) + 1.
```

**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.

**Three-phase distributed algorithm**

A distributed algorithm that enforces total and causal order for closed groups is given in Algorithm.The three phases of the algorithm are first described from the viewpoint of the sender, and then from the viewpoint of the receiver.

**Global State and Snapshot Recording Algorithms.**

**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 characterized by the state of its local memory and a history of its activity.
- The state of a channel is characterized by the set of messages sent along the channel less the messages received along the channel.
- The global state of a distributed system is a collection of the local states of its components.

**Example**

Let S1 and S2 be two distinct sites of a distributed system which maintain bank accounts A and B, respectively. A site refers to a process in this example. Let the communication channels from site S1 to site S2 and from site S2 to site S1 be denoted by $C_{12}$ and $C_{21}$, respectively. Consider the following sequence of actions, which are also illustrated in the timing diagram .

- Time $t_0$: Initially, Account A =\$600, Account B =\$200, $C_{12}$ =\$0,$C_{21}$ =\$0.
- Time $t_1$: Site S1 initiates a transfer of \$50 from Account A to Account B.
  Account A is decremented by \$50 to \$550 and a request for \$50 credit to Account B is sent on Channel $C_{12}$ to site S2. Account A =\$550, Account B =\$200, $C_{12}$ =\$50, $C_{21}$ =\$0.
- Time $t_2$: Site S2 initiates a transfer of \$80 from Account B to Account A. Account B is decremented by \$80 to \$120 and a request for \$80 credit to Account A is sent on Channel $C_{21}$ to site S1. Account A =\$550, Account B =\$120, $C_{12}$ =\$50, $C_{21}$ =\$80.
- Time $t_3$: Site S1 receives the message for a \$80 credit to Account A and updates Account A. Account A =\$630, Account B =\$120, $C_{12}$ =\$50, $C_{21}$ =\$0.
- Time $t_4$: Site S2 receives the message for a \$50 credit to Account B and updates Account B. Account A =\$630, Account B =\$170, $C_{12}$ =\$0, $C_{21}$ =\$0.
  Suppose the local state of Account A is recorded at time $t_0$ to show \$600 and the local state of Account B and channels $C_{12}$ and $C_{21}$ are recorded at time $t_2$ to show \$120, \$50, and \$80,

respectively. Then the recorded global state shows $850 in the system. An extra $50 appears in the system. The reason for the inconsistency is that Account A's state was recorded before the $50 transfer to Account B using channel $C_{12}$ was initiated, whereas channel $C_{12}$'s state was recorded after the $50 transfer was initiated.

## System model and definitions

**9. Discuss Global State System Model in detail.**

- The system consists of a collection of n processes, $p_1$, $p_2$, .. . , $p_n$, that are connected by channels
- There is no globally shared memory and processes communicate solely by passing messages. There is no physical global clock in the system.
- Message send and receive is asynchronous. Messages are delivered reliably with finite but arbitrary time delay.
- The system can be described as a directed graph in which vertices represent the processes and edges represent unidirectional communication 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 Cij , denoted by SCij , is given by the set of messages in transit in the channel.
- The actions performed by a process are modeled as three types of events, namely, internal events, message send events, and message receive events.
- For a message $m_{ij}$ that is sent by process $p_i$ to process $p_j$, let send($m_{ij}$) and rec($m_{ij}$) denote its send and receive events, respectively.
- Occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state.

**Transit**:

$transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$

- Thus, if a snapshot recording algorithm records the state of processes pi and pj as LSi and LSj, respectively, then it must record the state of channel Cij as transitLSiLS j. There are several models of communication among processes and different snapshot algorithms have assumed different models of communication.
- In the FIFO model, each channel acts as a first-in first-out message queue and, thus, message ordering is preserved by a channel. In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- A system that supports causal delivery of messages satisfies the following property: "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. The causal ordering model is useful in developing distributed algorithms and may simplify the design of algorithms.

**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

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

- A global state GS is a *consistent global state* iff it satisfies the following two conditions
  **C1:** *send*($m_{ij}$) $\in LS_i \Rightarrow m_{ij} \in SC_{ij}$ *rec*($m_{ij}$) $\in LS_j$ ($\oplus$ is the Ex-OR operator).
  **C2:** *send*($m_{ij}$) $\in LS_i \Rightarrow m_{ij} \in SC_{ij} \wedge$ *rec*($m_{ij}$) $\in LS_j$.
- Condition **C1** states the law of conservation of messages. Every message $m_{ij}$ that is recorded as sent in the local state of a process $p_i$ must be captured in the state of the channel $C_{ij}$ or in the collected local state of the receiver process $p_j$.
- Condition **C2** states that in the collected global state, for every effect, its cause must be present. If a message $m_{ij}$ is not recorded as sent in the local state of process $p_i$, then it must neither be present in the state of the channel $C_{ij}$ nor in the collected local state of the receiver process $p_j$.
- In a consistent global state, every message that is recorded as received is also recorded as sent. Such a global state captures the notion of causality that a message cannot be received if it was not sent.
- Consistent global states are meaningful global states and inconsistent global states are not meaning- ful in the sense that a distributed system can never be in an inconsistent state.

**Interpretation in terms of cuts**

- Cuts in a space–time diagram provide a powerful graphical aid in representing and reasoning about the global states of a computation.
- A cut is a line joining an arbitrary point on each process line that slices the space–time diagram into a PAST and a FUTURE. Recall that every cut corresponds to a global state and every global state can be graphically represented by a cut in the computation's space time diagram
- 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*.
- All the messages that cross the cut from the PAST to the FUTURE are captured in the corresponding channel state. For example, consider the space–time diagram for the computation illustrated in Figure 4.2.
- Cut $C_1$ is inconsistent because message $m_1$ is flowing from the FUTURE to the PAST. Cut $C_2$ is consistent and message $m_4$ must be captured in the state of channel $C_{21}$. in a consistent snapshot, all the recorded local states of processes are concurrent; that is, the recorded local state of no process casually affects the recorded local state of any other process.

**Issues in recording a global state**

- If a global physical clock were available, the following simple procedure could be used to record a consistent global snapshot of a distributed system.
- In this, the initiator of the snapshot collection 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 $C_{ij}$ includes all the messages that process $p_j$ receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot.
- Clearly, if channels are not FIFO, a termination detection scheme will be needed to determine when to stop waiting for messages on channels.
- However, a global physical clock is not available in a distributed system and the following two issues need to be addressed in recording of a consistent global snapshot of a distributed system

**I1:** distinguish between the messages to be recorded in the snapshot from those not to be recorded. The answer to this comes from conditions **C1** and **C2** as follows:
C1:Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot
C2:Any message that is sent by a process after recording its snapshot, must not be recorded I n the global snapshot .
**I2:** How to determine the instant when a process takes its snapshot. The answer to this comes from condition **C2** as follows:
A process $p_j$ must record its snapshot before processing a message $m_{ij}$ that was sent by process $p_i$ after recording its snapshot.

**Snapshot algorithms for FIFO channels**

10. **Explain Different Snapshort algorithms for FIFO channels in detail. (or) Describe the snapshot algorithms which could be applied for FIFO channels with diagrammatic representation. Discuss in detail about Snapshot algorithm for FIFO channels. Identify how individual local checkpoints can be combined with those from other processes to form global snapshots that are consistent. (Nov/Dec 2020/Apr/May 2021)** (NOV/DEC 2021) (APR/MAY 2023)

**Chandy-Lamport algorithm**

- The Chandy-Lamport algorithm uses a control message, called a *marker* whose role in a FIFO system is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a *marker*, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those notto be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incomingchannels.
- The algorithm can be initiated by any process by executing the "Marker Sending Rule" by which it records its local state and sends a marker oneach outgoing channel.
- A process executes the "Marker Receiving Rule" on receiving a marker. If the process has not yetrecorded its local state, it records the state of the channel on which them arker is received as emptyand executes the "Marker Sending Rule" to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes candetermine the global state.

Marker Sending Rule for process i
Process i records its state.
For each outgoing channel C on which a marker has not been sent, i sends a marker along C before i sends further messages along C.
Marker Receiving Rule for process j
On receiving a marker along channel C:
if j has not recorded its state then
Record the state of C as the empty set Follow the "Marker Sending Rule"
else
Record the state of C as the set of messages received along C after j 's state was recorded and before j received the marker along C

**Correctness**

- To prove the correctness of the algorithm, we show 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.

**Variations of the Chandy–Lamport algorithmSpezialetti–**

**Kearns algorithm**

There are two phases in obtaining a global snapshot:

- First locally recording the snapshot at every process
- Second distributing the resultant global snapshot to all the initiators.

**Two optimizations to the Chandy–Lamport algorithm.**

- The first optimization combines snap- shots concurrently initiated by multiple processes into a single snapshot.
- Optimization is linked with the second optimization, which deals with the efficient distribution of the global snapshot. A process needs to take only one snapshot, irrespective of the number of concurrent initiators and all pro- cesses are not sent the global snapshot. This algorithm assumes bi-directional channels in the system.

**Efficient snapshot recording**

- In the Spezialetti–Kearns algorithm, a marker carries the identifier of the initiator of the algorithm.
- Each process has a variable *master* to keep track of the initiator of the algorithm.
- When a process executes the "marker sending rule" on the receipt of its first marker, it records the initiator's identifier carried in the received marker in the *master* variable.
- A process that initiates the algorithm records its own identifier in the *master* variable.
- A key notion used by the optimizations is that of a *region* in the system.
- A region encompasses all the processes whose *master* field contains the iden- tifier of the same initiator. A region is identified by the initiator's identifier.
- When there are multiple concurrent initiators, the system gets partitioned into multiple regions.
- When the initiator's identifier in a marker received along a channel is different from the value in the *master* variable, a concurrent initiation of the algorithm is detected and the sender of the marker lies in a different region.
- The identifier of the concurrent initiator is recorded in a local variable id- border-set.
- The process receiving the marker does not take a snapshot for this marker and does not propagate this marker. Thus, the algorithm efficiently handles concurrent snapshot initiations by suppressing redundant snapshot collections – a process does not take a snapshot or propagate a snapshot request initiated by a process if it has already taken a snapshot in response to some other snapshot initiation.
- The state of the channel is recorded just as in the Chandy–Lamport algorithm (including those that cross a border between regions). This enables the snapshot recorded in one region to be merged with the snapshot recorded in the adjacent region.
- Thus, even though markers arriving at a node contain identifiers of different initiators, they are considered part of the same instance of the algorithm for the purpose of channel state recording.
- Snapshot recording at a process is complete after it has received a marker along each of its channels.
- After every process has recorded its snapshot, the system is partitioned into as many regions as the number of concurrent initiations of the algorithm. The variable id-border-set at a process contains the identifiers of the neighboring regions.

**Efficient dissemination of the recorded snapshot**

- In the snapshot recording phase, a forest of spanning trees is implicitly created in the system.
- The initiator of the algorithm is the root of a spanning tree and all processes in its region belong to its spanning tree.
- If process $p_i$ executed the "marker sending rule" because it received its first marker from process $p_j$, then process $p_j$ is the parent of process $p_i$ in the spanning tree.
- When a leaf process in the spanning tree has recorded the states of all incoming channels, the process sends the locally recorded state (local snap- shot, id-border-set) to its parent in the spanning tree.
- After an intermediate process in a spanning tree has received the recorded states from all its child processes and has recorded the states of all incoming channels, it forwards its locally recorded state and the locally recorded states of all its descendent processes to its parent.
- When the initiator receives the locally recorded states of all its descendents from its children

processes, it assembles the snapshot for all the processes in its region and the channels incident on these processes.

- The initiator knows the identifiers of initiators in adjacent regions using id-border-set information it receives from processes in its region.
- The initiator exchanges the snapshot of its region with the initiators in adjacent regions in rounds.
- In each round, an initiator sends to initiators in adjacent regions, any new information obtained from the initiator in the adjacent region during the previous round of message exchange.
- A round is complete when an initiator receives information, or the *blank* message (signifying no new information will be forthcoming) from all initiators of adjacent regions from which it has not already received a *blank* message.
- The message complexity of snapshot recording is O(e) irrespective of the number of concurrent initiations of the algorithm. The message complexity of assembling and disseminating the snapshot is $O(rn^2)$ where r is the number of concurrent initiations.

**Venkatesan's incremental snapshot algorithm**

- Many applications require repeated collection of global snapshots of the sys- tem. For example, recovery algorithms with synchronous checkpointing need to advance their checkpoints periodically.
- This can be achieved by repeated invocations of the Chandy–Lamport algorithm.
- Venkatesan proposed the following efficient approach: execute an algorithm to record an incremental snapshot since the most recent
- snapshot was taken and combine it with the most recent snapshot to obtain the latest snapshot of the system.
- The incremental snapshot algorithm of Venkatesan modifies the global snapshot algorithm of Chandy–Lamport to save on messages when computation messages are sent only on a few of the network channels, between the recording of two successive snapshots.
- The incremental snapshot algorithm assumes bidirectional FIFO channels, the presence of a single initiator, a fixed spanning tree in the network, and four types of control messages: init_snap, regular, and ack. init_snap, and snap completed messages traverse the spanning tree edges. regular and ack messages, which serve to record the state of non-spanning edges, are not sent on those edges on which no computation message has been sent since the previous snapshot.
- snapshot algorithm is K(u + n), where u is the number of edges on which a computation message has been sent since the previous snapshot. Venkatesan's algorithm achieves this lower bound in message complexity.
- The algorithm works as follows: snapshots are assigned version numbers and all algorithm messages carry this version number. The initiator notifies all the processes the version number of the new snapshot by sending init snap messages along the spanning tree edges. A process follows the "marker sending rule" when it receives this notification or when it receives a regular message with a new version number. The "marker sending rule" is modified so that the process sends regular messages along only those channels on which it has sent computation messages since the previous snapshot, and the process waits for ack messages in response to these regular messages. When a leaf process in the spanning tree receives all the ack messages it expects, it sends a snap completed message to its parent process. When a non-leaf process in the spanning

tree receives all the ack messages it expects, as well as a snap completed message from each of its child processes, it sends a snap completed message to its parent process.

- The algorithm terminates when the initiator has received all the ack messages it expects, as well as a snap completed message from each of its child processes.
- The selective manner in which regular messages are sent has the effect that a process does not know whether to expect a regular message on an incoming channel.
- A process can be sure that no such message will be received and that the snapshot is complete only when it executes the "marker sending rule" for the next initiation of the algorithm.

**Helary's wave synchronization method**

- Helary's snapshot algorithm incorporates the concept of message waves in the Chandy–Lamport algorithm. A wave is a flow of control messages such that every process in the system is visited exactly once by a wave control message, and at least one process in the system can determine when this flow of control messages terminates. A wave is initiated after the previous wave terminates. Wave sequences may be implemented by various traversal structures such as a ring. A process begins recording the local snapshot when it is visited by the wave control message.
- In Helary's algorithm, the "marker sending rule" is executed when a control message belonging to the wave flow visits the process. The process then for- wards a control message to other processes, depending on the wave traversal structure, to continue the wave's progression. The "marker receiving rule"
- Is modified so that if the process has not recorded its state when a marker is received on some channel, the "marker receiving rule" is not executed and no messages received after the marker on this channel are processed until the control message belonging to the wave flow visits the process. Thus, each process follows the "marker receiving rule" only after it is visited by a control message belonging to the wave.
- Note that in this algorithm, the primary function of wave synchronization is to evaluate functions over the recorded global snapshot. This algorithm has a message complexity of O(e) to record a snapshot (because all channels need to be traversed to implement the wave).
- An example of this function is the number of messages in transit to each process in a global snapshot, and whether the global snapshot is strongly consistent. For this function, each process maintains two vectors, SENT and RECD.
- The ith elements of these vectors indicate the number of messages sent to/received from process i, respectively, since the previous visit of a wave control message.
- The wave control messages carry a global abstract counter vector whose $i^{th}$ entry indicates the number of messages in transit to process i.
- These entries in the vector are updated using the SENT and RECD vectors at each node visited. When the control wave terminates, the number of messages in transit to each process as recorded in the snapshot is known.

| UNIT III DISTRIBUTED MUTEX & DEADLOCK |
| --- |
| Distributed Mutual exclusion Algorithms: Introduction – Preliminaries – Lamport's algorithm – Ricart-Agrawala's algorithm – Token-Based Algorithm – Suzuki–Kasami's Broadcast Algorithm. Deadlock Detection in Distributed Systems: Introduction – System Model – Preliminaries – Models of Deadlocks – Chandy-Misra-Haas Algorithms for the AND model and the OR model. |

**1. Explain in detailed about distributed mutual exclusion algorithms. (or) Discuss in detail the requirement that mutual exclusion algorithms should satisfy and discuss what metric we use to measure the performance of mutual exclusion algorithm. Recognize a distributed mutual exclusion algorithm as an illustration of the clock synchronization scheme. (Nov/Dec 2020/Apr/May 2021) (Apr/May 2023)**

**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 resource or data is serialized, that is, executed in a mutually exclusive manner.
- 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 (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion. The decision as to which process is allowed access to the CS next is arrived at by message passing, in which each process learns about the state of all other processes in some consistent way.
- The design of distributed mutual exclusion algorithms is complex because these algorithms have to deal with unpredictable message delays and incomplete knowledge of the system state.
- There are three basic approaches for implementing distributed mutual exclusion:
  - ✓ Token-based approach.
  - ✓ Non-token-based approach.
  - ✓ Quorum-based approach.

- 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 is ensured because the token is unique. The algorithms based on this approach essentially differ in the way a site carries out the search for the token.
- 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. 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, at least one site receives both the requests and this site is responsible to make sure that only one request executes the CS at any time.

**Preliminaries**

- In this section, we describe the underlying system model, discuss the requirements that mutual exclusion algorithms should satisfy, and discuss what metrics we use to measure the performance of mutual exclusion algorithm.

**System Model**

- The system consists of $N$ sites, $S1, S2, \ldots, SN$. Without loss of generality, we assume that a single process is running on each site.
- The process at site $Si$ is denoted by $pi$. All these processes communicate asynchronously over an underlying communication network.
- 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 the CS.

- 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: requesting the CS, executing the CS, or 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.
- We do not make any assumption regarding communication channels if they are FIFO or not. This is algorithm specific.
- 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. The general rule followed is that the smaller the timestamp of a request, the higher its priority to execute the CS.

We use the following notation: $N$ denotes the number of processes or sites involved in invoking the critical section, $T$ denotes the average message delay, and $E$ denotes the average critical section execution time.

**Requirements of mutual exclusion algorithms**

A mutual exclusion algorithm should satisfy the following properties:

i. **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.

ii. **Liveness property** This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages that will never arrive. 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.

iii. **Fairness** Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS. In mutual exclusion algorithms, the fairness property generally means that the CS execution requests are executed in order of their arrival in the system (the time is determined by a logical clock).

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

**Performance metrics**

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

- **Message complexity** This is the number of messages that are required per CS execution by a site.
- **Synchronization delay** After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 9.1). Note that normally one or more sequential message exchanges may be required after a site exits the CS and before the next site can enter the CS.
- **Response time** This is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 9.2). Thus, response time does not include the time a request waits at a site before its request messages have been sent out.

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

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

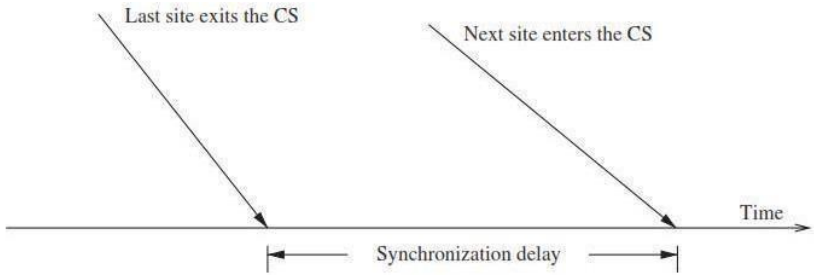



**Figure 9.1** Synchronization delay.

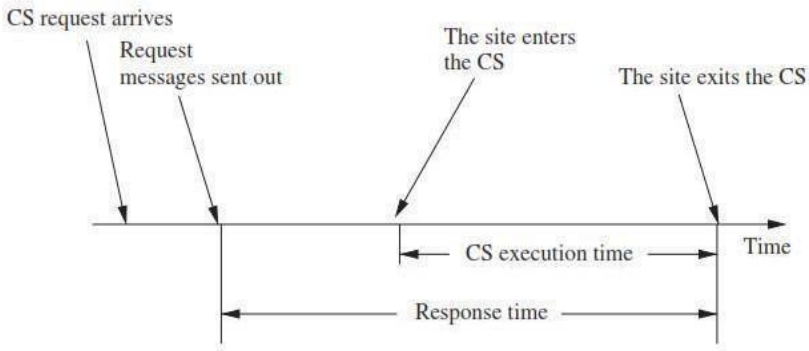



**Figure 9.2** Response time.

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. Performance of a mutual exclusion algorithm depends upon the load and we often study the performance of mutual exclusion algorithms under 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 at a site.
- Thus, in heavy load conditions, after having executed a request, a site immediately initiates activities to execute its next CS request. A site is seldom in the idle state in heavy load conditions.
- For many mutual exclusion algorithms, the performance metrics can be computed easily under low and heavy loads through a simple mathematical reasoning.

**Best and worst case performance**

- Generally, mutual exclusion algorithms have best and worst cases for the performance metrics. In the best case, prevailing conditions are such that a performance metric attains the best possible value.
- For example, in most mutual exclusion algorithms the best value of the response time is a round- trip message delay plus the CS execution time, $2T+E$.
- Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively.
- For examples, the best and worst values of the response time are achieved when load is, respectively, low and high; in some mutual exclusion algorithms the best and the worse message traffic is generated at low and heavy load conditions, respectively.

2. **Discuss in detail about Lamport's algorithm.**
   **External synchronization ensures internal synchronization. But the vice versa does not stand true. Justify. Explain Lamport's algorithm in brief. Outline Lamport's algorithm with an example. (NOV/DEC 2020/APR/MAY 2021) (NOV/DEC 2021)**

**Lamport's algorithm**

- Lamport developed a distributed mutual exclusion algorithm s anillustration of his clock synchronization scheme.
- The algorithm is fair in the sense that a request for CS are executed in theorder of their timestamps and time is determined by logical clocks.
- When a site processes a request for the CS, it updates its local clock andassigns the request a timestamp.
- The algorithm executes CS requests in the increasing order of timestamps.

  Every site $Si$ keeps a queue, *request_queue*$i$, which contains mutual exclusion requests ordered by their timestamps. (Note that this queue is different from the queue that contains local requests for CS execution awaiting their turn.)
- This algorithm requires communication channels to deliver messages in FIFO order.

**Requesting the critical section**

• When a site $Si$ wants to enter the CS, it broadcasts a REQUEST($ts_i$, $i$) message to all other sites and places the request on *request_queue*$i$. (($ts_i$,$i$) denotes the timestamp of the request.)

• When a site $Sj$ receives the REQUEST($ts_i$, $i$) message from site $Si$, it places site $Si$"s request on *request_queue*$j$ and returns a timestamped REPLY message to $Si$.

**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 ($ts_i$, $i$) fromall other sites.

**L2:** $Si$"s request is at the top of *request_queue*$i$.

**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.

**Algorithm 9.1** Lamport"s algorithm.

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 9.1** *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 holds 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 $request\_queue_j$ 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 the $request\_queue_j$ – a contradiction! Hence, Lamport"s algorithm achieves mutual exclusion.

**Theorem 9.2** *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 has a 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 that at some instant in time $Sj$ 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_queue* at 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_queue*$j$. This is a contradiction. Hence Lamport"s algorithm is a fair mutual exclusion algorithm.

**Example** In Figures 9.3 to 9.6, we illustrate the operation of Lamport"s algorithm. In Figure 9.3, sites $S1$ and $S2$ are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (1,1) and (1,2), respectively. In Figure 9.4, both the sites $S1$ and $S2$ have received REPLY messages from all other sites. $S1$ has its request at the top of its *request_queue* but site $S2$ does not have its request at the top of its *request_queue*. Consequently, site $S1$ enters the CS. In Figure 9.5, $S1$ exits and sends RELEASE mesages to all other sites. In Figure 9.6, site $S2$ has received REPLY from all other sites and also received a RELEASE message

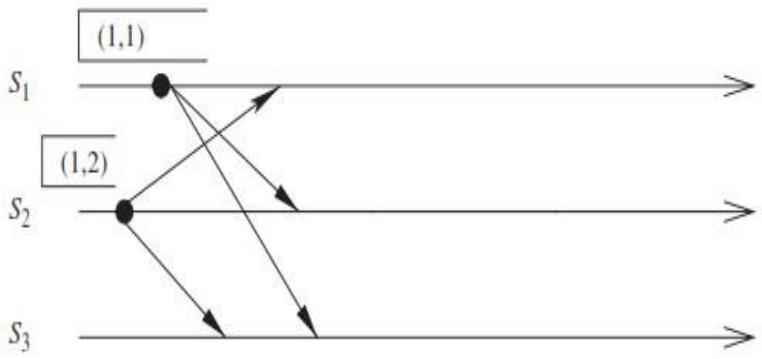



**Figure 9.3** Sites $S_1$ and $S_2$ are Making Requests for the CS.

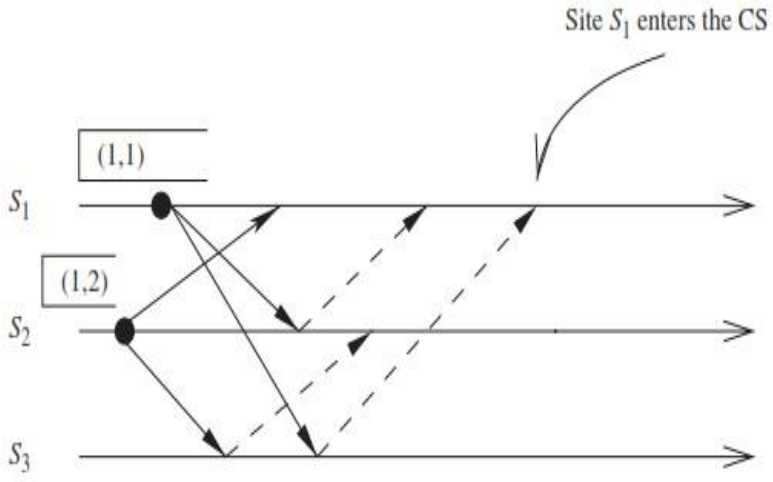



**Figure 9.4** Site $S_1$ enters the CS.

**Figure 9.5** Site $S_1$ exits the CS and sends RELEASE messages.

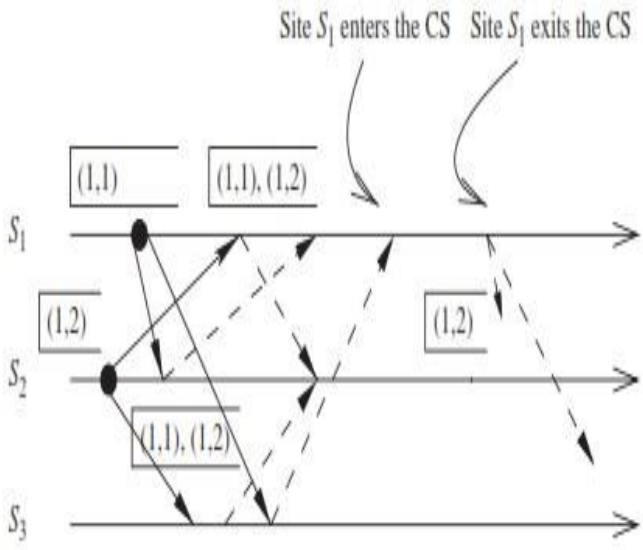
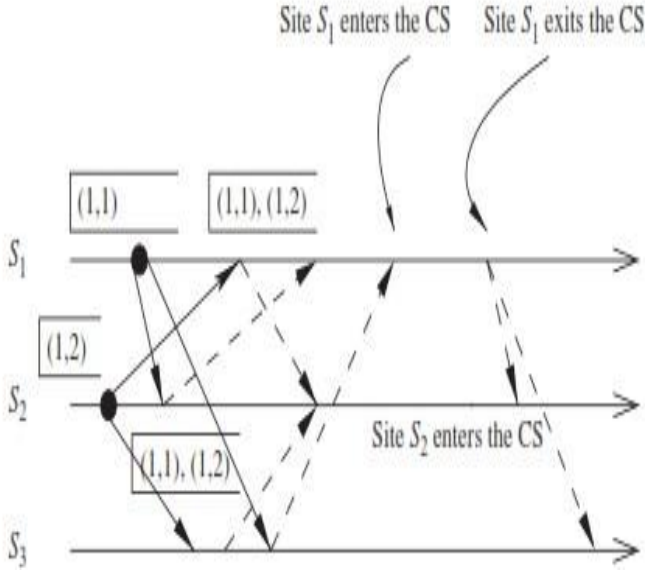



**Figure 9.6** Site $S_2$ enters the CS.



from site $S1$. Site $S2$ updates its *request_queue* and its request is now at the top of its *request_queue*. Consequently, it enters the CS next.

**Performance**

For each CS execution, Lamport‟s algorithm requires $(N - 1)$ REQUEST messages, $(N - 1)$ REPLY messages, and $(N - 1)$ RELEASE messages. Thus, Lamport‟s algorithm requires $3(N - 1)$ messages per CS invocation. The synchronization delay in the algorithm is $T$ .

**An optimization**

- In Lamport‟s algorithm, REPLY messages can be omitted in certain situations. For example, if site $Sj$ receives a REQUEST message from site $Si$ after it has sent its own REQUEST message with a timestamp higher than the timestamp of site $Si$‟s request, then site $Sj$ need not send a REPLY message to site $Si$.
- This is because when site $Si$ receives site $Sj$‟s request with a timestamp higher than its own, it can conclude that site $Sj$ does not have any smaller timestamp request which is still pending (because communication channels preserves FIFO ordering).
- With this optimization, Lamport‟s algorithm requires between $3(N - 1)$ and $2(N - 1)$ messages per CS execution.

**3. Write in detail about Ricart-Agarwala algorithm.( Nov/Dec 2022)**

**Ricart–Agrawala algorithm**

- The Ricart–Agrawala algorithm assumes that the com-munication 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 $pj$ andsends a REPLY message to $pj$ only after executing the CS for its pending 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 $p_i$ maintains the request-deferred array, $RD_i$, the size of which is the same as the number of processes in the system. Initially, $\forall i \; \forall j$:

$RDi[j] = 0$. Whenever $pi$ defers the request sent by $pj$, it sets $RDi[j] = 1$, and after it has sent a REPLY message to $pj$, it sets $RDi[j] = 0$.

**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 requesting and $Si$"s request"s timestamp is smaller than site $Sj$"s own request"s timestamp. Otherwise, the reply is deferred and $Sj$ sets $RDj[i] := 1$.

**Executing the critical section**

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

**Releasing the critical section**

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

**Algorithm 9.2** The Ricart–Agrawala algorithm.

- 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 that are requesting the CS with higher priority (i.e., smaller timestamp).
- Thus, when a site sends out deferred REPLY messages, the 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.

**Correctness**

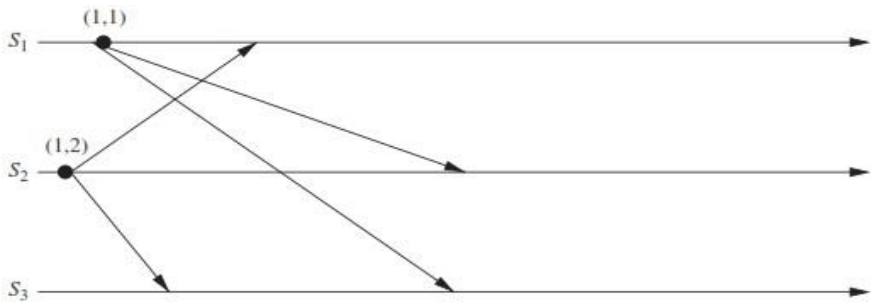
**Theorem 9.3** *Ricart–Agrawala algorithm achieves mutual exclusion.*

**Proof** Proof is by contradiction. Suppose two sites $Si$ and $Sj$ are executing the CS concurrently and $Si$‟s request has higher priority (i.e., smaller times- tamp) than the request of $Sj$. Clearly, $Si$ received $Sj$‟s request after it has made its own request. (Otherwise, $Si$‟s request will have lower priority.) Thus, $Sj$ can concurrently execute the CS with $Si$ only if $Si$ returns a REPLY to $Sj$ (in response to $Sj$‟s request) before $Si$ exits the CS. However, this is impossible because $Sj$‟s request has lower priority.

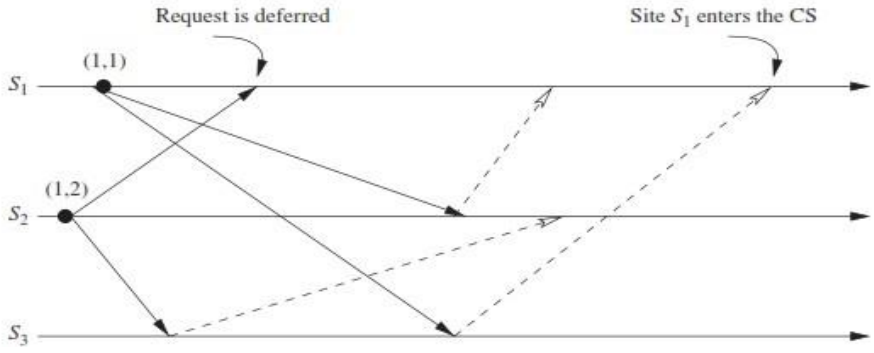
Therefore, the Ricart–Agrawala algorithm achieves mutual exclusion.

In the Ricart–Agrawala algorithm, for every requesting pair of sites, the site with higher priority request will always defer the request of the lower priority site. At any time only the highest priority request succeeds in getting all the needed REPLY messages.

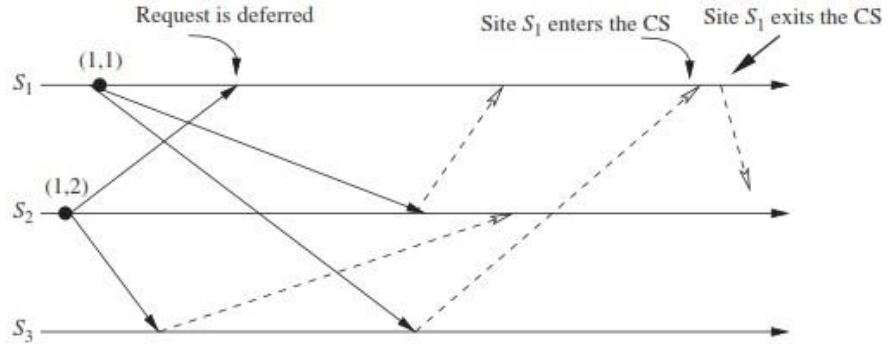
**Example** Figures illustrate the operation of the Ricart–Agrawala algorithm. In Figure 9.7, sites $S_1$ and $S_2$ are each making requests for the



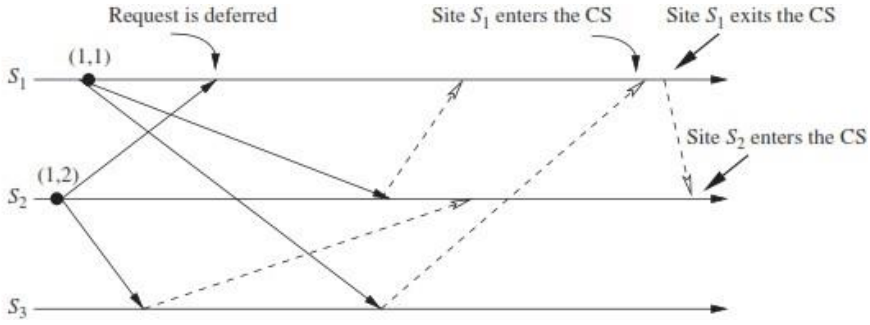

**igure 9.7** Sites $S_1$ and $S_2$ ach make a request for the CS.




**igure 9.8** Site $S_1$ enters the CS.




**igure 9.9** Site $S_1$ exits the CS nd sends a REPLY message to $_2$'s deferred request.




**igure 9.10** Site $S_2$ enters he CS.

CS and sending out REQUEST messages to other sites. The timestamps of the requests are (2,1) and (1,2), respectively. In Figure 9.8, $S2$ has received REPLY messages from all other sites and, consequently, 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.

**Performance**

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

**4. Explain in detail about Maekawa's algorithm.**

**Maekawa's algorithm**

Maekawa"s algorithm [14] was the first quorum-based mutual exclusion algorithm. The request sets for sites (i.e., quorums) in Maekawa"s algorithm are constructed to satisfy the following conditions:
M1 $(\forall i \forall j : i \bullet= j, 1 \leq i, j \leq N :: R_i \cap R_j \bullet= \emptyset)$.M2 $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$.
M3 $(\forall i : 1 \leq i \leq N :: |R_i| = K)$.

M4 Any site S$j$ is contained in $K$ number of $Ri$s, $1 \leq i, j \leq N$ .

Maekawa used the theory of projective√ planes and showed that

$N = K(K - 1) + 1$. This relation gives $|Ri| = N$ .

- Since there is at least one common site between the request sets of any two sites (condition M1), every pair of sites has a common site which mediates conflicts between the pair.
- A site can have only one outstanding REPLY message at any time; that is, it grants permission to an incoming request if it has not granted permission to some other site.
- Therefore, mutual exclusion is guaranteed. This algorithm requires delivery of messages to be in the order they are sent between every pair of sites.
- Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm. Condition M3 states that the size of the requests sets of all sites must be equal, which implies that all sites should have to do an equal amount of work to invoke mutual exclusion. Condition M4 enforces that exactly the same number of sites should request permission from any site, which implies that all sites have "equal responsibility" in granting permission to other sites.
- In Maekawa"s algorithm, a site $Si$ executes the steps shown in Algorithm9.5 to execute the CS.

**Requesting the critical section**:

(a) A site $Si$ requests access to the CS by sending REQUEST($i$) messages to all sites in its request set $Ri$.

(b) When a site $Sj$ receives the REQUEST($i$) message, it sends a REPLY($j$) message to $Si$ provided it hasn"t sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST($i$) for later consideration.

**Executing the critical section:**

(c) Site $Si$ executes the CS only after it has received a REPLY message from every site in $Ri$.

**Releasing the critical section:**

(d) After the execution of the CS is over, site $Si$ sends a RELEASE($i$)

message to every site in $Ri$.

(e) When a site $Sj$ receives a RELEASE($i$) message from site $Si$, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

**Algorithm 9.5** Maekawa"s algorithm.

**Correctness**

**Theorem 9.3** *Maekawa's algorithm achieves mutual exclusion.*

**Proof** Proof is by contradiction. Suppose two sites $Si$ and $Sj$ are concurrently executing the CS. This means site $Si$ received a REPLY message from all sites in $Ri$ and concurrently site $Sj$ was able to receive a REPLY message from all sites in $Rj$. If $Ri \cap Rj = \{Sk\}$, then site $Sk$ must have sent REPLY messages to both $Si$ and $Sj$ concurrently, which is a contradiction.

**Problem of deadlocks**

- Maekawa"s algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps [14,22].
- Thus, a site may send a REPLY message to a site and later force a higher priority request from another site to wait.
- Without loss of generality, assume three sites $Si$, $Sj$, and $Sk$ simultaneously invoke mutual exclusion.
- Suppose $Ri \cap Rj = \{Sij\}$, $Rj \cap Rk = \{Sjk\}$, and $Rk \cap Ri = \{Ski\}$.
- Since sites do not send REQUEST messages to the sites in their request sets in any particular order and message delays are arbitrary, the following scenario is possible: $Sij$ has been locked by $Si$ (forcing $Sj$ to wait at $Sij$), $Sjk$ has been locked by $Sj$ (forcing $Sk$ to wait at $Sjk$), and $Ski$ has been locked by $Sk$ (forcing $Si$ to wait at $Ski$).
- This state represents a deadlock involving sites $Si$, $Sj$, and $Sk$.

**Handling deadlocks**

- Maekawa"s algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock (unless the former has succeeded in acquiring locks on all the needed sites) [14, 22].
- A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.

Deadlock handling requires the following three types of messages:

**FAILED** A FAILED message from site $Si$ to site $Sj$ indicates that $Si$ cannot grant $Sj$"s request because it has currently granted permission to a site with a higher priority request.

**INQUIRE** An INQUIRE message from $Si$ to $Sj$ indicates that $Si$ would like to find out from $Sj$ if it has succeeded in locking all the sites in its request set.

**YIELD** A YIELD message from site $Si$ to $Sj$ indicates that $Si$ is returning the permission to $Sj$ (to yield to a higher priority request at $Sj$).

Details of how Maekawa"s algorithm handles deadlocks are as follows:

- When a REQUEST(*ts*, *i*) from site $Si$ blocks at site $Sj$ because $Sj$ has currently granted permission to site $Sk$, then $Sj$ sends a FAILED(*j*) message to $Si$ if $Si$"s request has lower priority. Otherwise, $Sj$ sends an INQUIRE(*j*) message to site $Sk$.
- In response to an INQUIRE(*j*) message from site $Sj$, site $Sk$ sends a YIELD(*k*) message to $Sj$ provided $Sk$ has received a FAILED message from a site in its request set and if it sent a YIELD to any of these sites, but has not received a new REPLY from it.
- In response to a YIELD(*k*) message from site $Sk$, site $Sj$ assumes as if it has been released by $Sk$, places the request of $Sk$ at appropriate location in the request queue, and sends a REPLY (*j*) to the top request's site in the queue.

**5. Explain about Suzuki-Kasami's broastcast algorithm. (Nov/Dec 2022)**

**Suzuki–Kasami's broadcast algorithm**

- 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 that 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.

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

- **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, but it 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 out dated.

- **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 $S_i$ receives a token request message from a site $S_j$, site $S_j$ 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 $Sj$ has the form REQUEST($j$, $n$) where $n$ ($n$ = 1, 2,... ) is a sequence number that indicates that site $Sj$ is requesting its $n$th CS execution.
- A site $Si$ keeps an array of integers $RNi[1,\ldots,N]$ 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 $LN[1,\ldots,N]$, 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 $LN[1,\ldots,N]$ 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 a token.
- After executing the CS, a site checks this condition for all the $j$"s to determine all the sites that are requesting the token and places their i.d."s in queue $Q$ if these i.d."s are not already present in $Q$.
- Finally, the site sends the token to the site whose i.d. is at the head of $Q$.

**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 $RN_i[i]$.)

(b) When a site $S_j$ receives this message, it sets $RN_j[i]$ to $max(RN_j[i], sn)$. If $S_j$ has the idle token, then it sends the token to $S_i$ if $RN_j[i] = LN[i] + 1$.

**Executing the critical section**:

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

**Releasing the critical section**: Having finished the execution of the CS, site

$S_i$ takes the following actions:

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

(e) For every site $S_j$ whose i.d. is not in the token queue, it appends its i.d. to the token queue if $RN_i[j] = LN[j] + 1$.

(f) If the token queue is nonempty after the above update, $S_i$ deletes the top site i.d. from the token queue and sends the token to the site indicated by the i.d.

**Algorithm 9.7** Suzuki–Kasami"s broadcast algorithm.

- Thus, as shown in Algorithm9.7, after executing the CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS).

- 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 9.3** *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.

**Performance**

- The beauty of the 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. The synchronization delay in this algorithm is 0 or *T*.

**6. Write in detail about deadlock detection in distributed.**

**How we can achieve deadlock detection in distributed systems? Provide various models to carry out the same systems. Analyze the Correctness criteria of the deadlock detection algorithm. (NOV/DEC 2021) (APR/MAY 2023)**

**Deadlock detection in distributed systems**

**Introduction**

- Deadlocks are a fundamental problem in distributed systems and deadlock detection in distributed systems has received considerable attention in the past.
- In distributed systems, a process may request resources in any order, which may not be known a priori, and a process can request a resource while holding others.
- If the allocation sequence of process resources is not controlled in such environments, deadlocks can occur.
- A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set.
- Deadlocks can be dealt with using any one of the following three strategies: deadlock prevention, deadlock avoidance, and 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 holds the needed resource.
- In the deadlock avoidance approach to distributed systems, 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.
- In this chapter, we study several distributed deadlock detection techniques based on various strategies.

**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 $P1, P2, \ldots, Pi, \ldots, Pn$ that communicate by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor. The processors do not share a common global memory and communicate solely by passing messages over the communication network.
- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost, garbled, or duplicated due to timeout and retransmission, processors may fail, and communication links may go down.
- The system can be modeled as a directed graph in which vertices represent the processes and edges 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 modeled 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 an edge to process $P32$ of site 2.
- Process $P32$ of site 2 is waiting for a resource that is currently held by process $P33$ of site 3.
- At the same time process $P21$ at site 1 is waiting on process $P24$ at site 4 to release a resource, and so on. If $P33$ starts waiting on process $P24$, then processes in the WFG are involved in a deadlock depending upon the request model.

## Preliminaries

## Deadlock handling strategies

- There are three strategies for handling deadlocks, *viz.*, deadlock prevention, deadlock avoidance, and deadlock detection.
- Handling of deadlocks 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 that holds the needed resource.

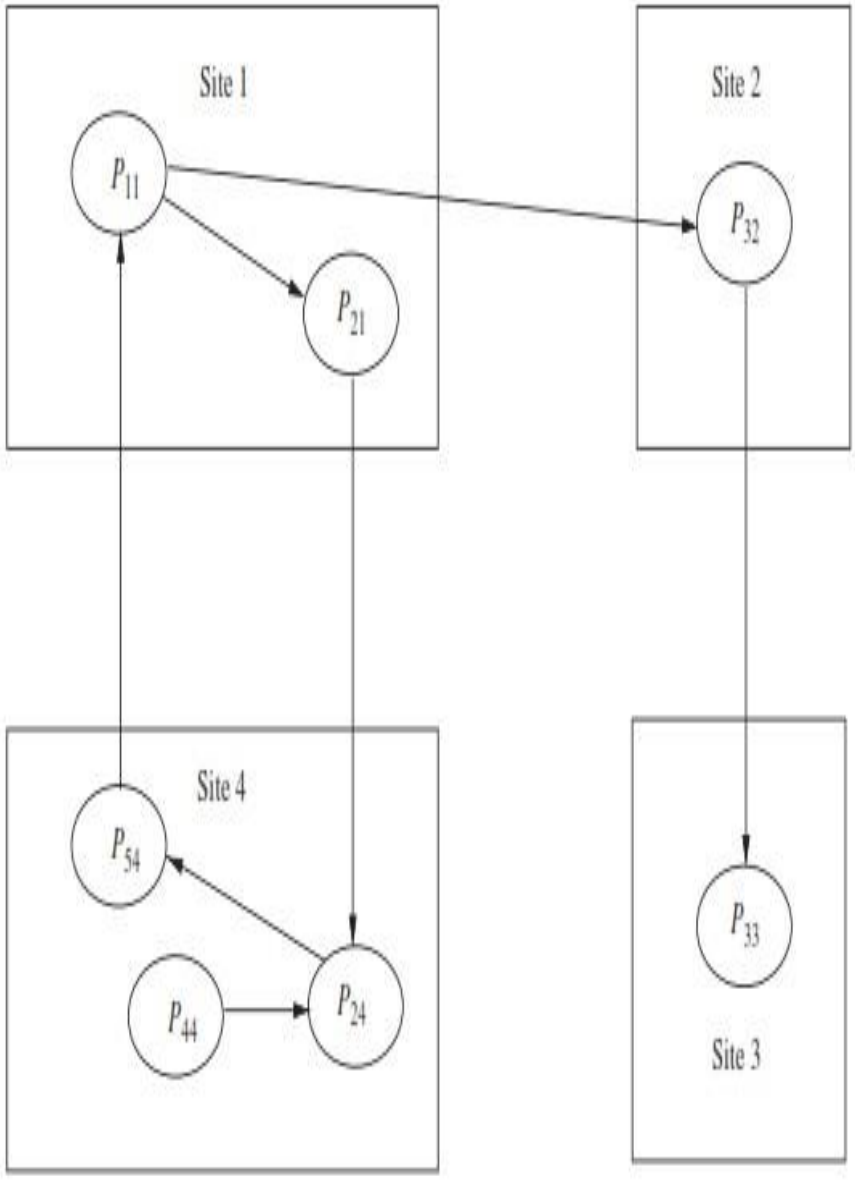



**Figure 10.1** Example of a WFG.

- 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).
- Due to several problems, however, deadlock avoidance is impractical in distributed systems.
- Deadlock detection requires an examination of the status of process– resource interactions for the presence of cyclic wait.
- Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.

**Issues in deadlock detection**

- Deadlock handling using the approach of deadlock detection entails addressing two basic issues: first, detection of existing deadlocks and, second, resolution of detected deadlocks.

**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 the search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems.

***Correctness criteria***

A deadlock detection algorithm must satisfy the following two conditions:

- **Progress (no undetected deadlocks)** The algorithm must detect all existing deadlocks in a 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.
- **Safety (no false deadlocks)** The algorithm should not report deadlocks that 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 an out-of-date and inconsistent WFG of the system.
- As a result, sites may detect a cycle that never existed but whose different segments existed in the system at different times. This is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

**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 a timely manner, it may result in detection of phantom deadlocks.
- Untimely and inappropriate cleaning of bro- ken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

**7. Explain in detail about the models of dead-locks.**

**Name and explain the different types of deadlock models in distributed system with the commonly used strategies to handle deadlocks with a neat diagram. (Nov/Dec 2022)**

**Models of deadlocks**

- Distributed systems allow many kinds of resource requests. A process might require a 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 theypermit.

**The single-resource model**

- The single-resource model is the simplest resource model in a distributed system, where 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 more than one resource simultaneously and the 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 anAND node.
- Consider the example WFG described in the Figure 10.1. Process $P11$ has two outstanding resource requests. In case of the AND model, $P11$ shall become active from idle state only after both the resources are granted.

- There is a cycle $P11 \rightarrow P21 \rightarrow P24 \rightarrow P54 \rightarrow P11$, whichcorresponds to a deadlock situation.
- In the AND model, if a cycle is detected in the WFG, it implies adeadlock but not vice versa.
- That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process $P44$ in Figure 10.1. It is not a part of any cycle but is still deadlocked as it is dependent on $P24$, which isdeadlocked.
- 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 ORnodes.
- Presence of a cycle in the WFG of an OR model does not imply adeadlock in the OR model.
- If all nodes are OR nodes, then process $P11$ is not deadlocked because once process $P33$ releases its resources, $P32$ shall becomeactive 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 it has a pending OR requests to be satisfied.

- With every blocked process, there is an associated set of processes called dependent set.
- A process shall move from an *idle* to an *active* state on receiving a grant message from any of the processes in its dependent set.
- A process is permanently blocked if it never receives a grant message from any of the processes in its dependent set.
- 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*.
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 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 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 processes in 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 deadlocked process that is not a part of a knot.

Consider Figure 10.1, where *P*44 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.

**8. Explain Chandy–Misra–Haas algorithm for the AND model.**

- Chandy–Misra–Haas‟s distributed deadlock detection algorithm for the AND model [6], which 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 dead-
- lock 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*.

**The algorithm**

Algorithm is executed to determine if a blocked process is deadlocked. 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.

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

On the receipt of a probe (*i*, *j*, *k*), the sitetakes 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.

**Algorithm:** Chandy–Misra–Haas algorithm for the AND model.

## 9. Explain Chandy–Misra–Haas algorithm for the OR model

- Chandy–Misra–Haas"s distributed deadlock detection algorithm for the OR model, which 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 $P_i$ and are being sent from process $P_j$ to process $P_k$.

**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 $P_k$ receives a *query*(*i*, *j*, *k*) message, it takes the following actions:

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

ii. If this is not the engaging *query*, then $P_k$ returns a *reply* message to it immediately provided $P_k$ has been continuously blocked since it received the corresponding engaging *query*. Otherwise, it discards the *query*.

- Process $P_k$ maintains a boolean variable $wait_k(i)$ that denotes the fact that it has been continuously blocked since it received the last engaging *query* from process $P_i$.
- When a blocked process $P_k$ receives a *reply*(*i*, *j*, *k*) message, it decrements $num_k(i)$ only if $wait_k(i)$ holds.
- A process sends a reply message in response to an engaging *query* only after it has received a *reply* to every *query* message it has sent out for this engaging *query*.
- The initiator process detects a deadlock when it has received *reply* messages to all the *query* messages it has sent out.

**The algorithm**

- The algorithm works as shown in Algorithm. For ease of presentation, we have 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.

**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:** Chandy–Misra–Haas algorithm for the OR model.

**PART A:**

**1. What is Mutual exclusion? (or) Explain the term Mutual Exclusion. (NOV/DEC 2022)**

- 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 (semaphores) or a local kernel cannot be used to implement mutual exclusion.

**2. What are the types of mutual exclusion? (OR) Identify the three basic approaches for implementing distributed mutual exclusion. (Apr/May 2023)**

- There are three basic approaches for implementing distributed mutual exclusion:
  a. Token-based approach.
  b. Non-token-based approach.
  c. Quorum-based approach.

**3. What is token based approach?**

- In the token-based approach, a unique token (also known as thePRIVILEGE message) is shared among the sites.
- A site is allowed to enter its CS if it possesses the token and it continues tohold the token until the execution of the CS is over.

**4. Define Non-Token based approach**?

- 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.

**5. What is quorum-based approach?**

- 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, at least one site receives both the requests and this site is responsible to make sure that only one request executes the CS at any time.

**6. Define idle state.**

- 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.

**7. What are the requirements of mutual exclusion algorithms**

A mutual exclusion algorithm should satisfy the following properties:

a. Safety property
b. Liveness property
c. Fairness

**8. Define 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.

**9. Define Liveness property.**

This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages that will never arrive. 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.

**10. Define Fairness.**

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

**11. How to measure the Performance of metrics in mutual exclusion?**

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

- Message complexity
- Synchronization delay
- Response time
- System throughput

**12. Define Message complexity**.

This is the number of messages that are required per CS execution by a site.

**13. What is Synchronization delay?**

**Synchronization delay** After a site leaves the CS, it is the time required and before the next site enters the CS. Normally one or more sequential message exchanges may be required after a site exits the CS and before the next site can enter the CS.

**14. Define Response time.**

**Response time** This is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 9.2). Thus, response time does not include the time a request waits at a site before its request messages have been sent out.

**15. Define System throughput.**

**System throughput** This 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)

**16. Define Lamport's algorithm**

- Lamport developed a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme.

- The algorithm is fair in the sense that a request for CS are executed in the order of their timestamps and time is determined by logical clocks.

**17. What is Ricart-Agarwala algorithm.**

- The Ricart–Agrawala algorithm assumes that 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 requesttheir permission to enter the critical section.
- A process sends a REPLY message to a process to give its permission to that process.

**18. Define Maekawa's algorithm.**

**Maekawa's algorithm**

Maekawa's algorithm was the first quorum-based mutual exclusion algorithm. The request sets for sites (i.e., quorums) in Maekawa's algorithm are constructed to satisfy the following conditions:

M1 ($\forall i \forall j : i \bullet= j, 1 \leq i, j \leq N :: R_i \cap R_j \bullet= \emptyset$).

M2 ($\forall i : 1 \leq i \leq N :: S_i \in R_i$).

M3 ($\forall i : 1 \leq i \leq N :: |R_i| = K$).

M4 Any site S$j$ is contained in $K$ number of $Ri$s, $1 \leq i, j \leq N$.

Maekawa used the theory of projective√ planes and showed

that

$N = K(K - 1) + 1$. This relation gives $|Ri| = N$.

**19. What are the types of message handling in deadlock?**

Deadlock handling requires the following three types of messages:

**a.** Failed

**b.** Inquire

**c.** Yield

**20. Define FAILED.**

A FAILED message from site *Si* to site *Sj* indicates that *Si* cannot grant *Sj*'s request because it has currently granted permission to a site with a higher priority request.

**21. Define INQUIRE.**

An INQUIRE message from *Si* to *Sj* indicates that *Si* would like to find out from *Sj* if it has succeeded in locking all the sites in its request set.

**22. What is YIELD?**

A YIELD message from site *Si* to *Sj* indicates that *Si* is returning the permission to *Sj* (toyield to a higher priority request at *Sj*).

**23. Define Suzuki-Kasami's broadcast algorithm.**

**Suzuki–Kasami's broadcast algorithm**

α. In Suzuki–Kasami's algorithm, if a site that wants to enter the CS does nothave the token, it broadcasts a REQUEST message for the token to all other sites.

β. A site that possesses the token sends it to the requesting site upon thereceipt of its REQUEST message.

χ. If a site receives a REQUEST message when it is executing the CS, itsends the token only after it has completed the execution of the CS.

**24. What is Deadlock detection in distributed systems?**

Deadlocks are a fundamental problem in distributed systems and deadlock detection in distributed systems has received considerable attention in the past.

In distributed systems, a process may request resources in any order, which may not be known a priori, and a process can request a resource while holding others.

**25. What is deadlock? (or) Define Deadlock. (NOV/DEC 2021) (Nov/Dec 2022)**

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

**26. What are the types of deadlocks strategies?**

Deadlocks can be dealt with using any one of the following three strategies: deadlock prevention, deadlock avoidance, and deadlock detection.

**27. Define deadlock prevention.**

Deadlock prevention is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre-emptying a process that holds the needed resource.

**28. What is Deadlock avoidance?**

In the deadlock avoidance approach to distributed systems, a resource isgranted to a process if the resulting global system is safe.

**29. What is Deadlock detection?**

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.

**30. Define Wait-for graph (WFG) (or) What is the purpose of wait-for- graph (WFG)? Give an example for WFG. Outline the wait-for-graph. (Nov/Dec 2020) (Apr/May 2021) (Apr/May 2023)**

α. In distributed systems, the state of the system can be modeled by directed graph, called a *wait-for graph* (WFG).

β. 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.

**31. What are the conditions of deadlock detection algorithm must satisfy?**

- **Progress (no undetected deadlocks)** The algorithm must detect all existing deadlocks in a finite time. Once a deadlock has occurred, the deadlock detection activity should continuously progress until the deadlock is detected.
- **Safety (no false deadlocks)** The algorithm should not report deadlocks that do not exist (called *phantom or false* deadlocks).

**32. What is the single-resource model?**

α. The single-resource model is the simplest resource model in a distributed system, where a process can have at most one outstanding request for only one unit of a resource.

**33. Define the AND model.**

α. In the AND model, a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.

**34. What is 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.

**35. What is 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.

36. **Define Mitchell and Merritt's algorithm for the single-resource model.**

i. Mitchell and Merritt's algorithm belongs to the class of edge- chasing algorithms where probes are sent in the opposite direction to the edges of the WFG.

ιι. When a probe initiated by a process comes back to it, the process declares deadlock.

**37. Define Chandy–Misra–Haas algorithm for the OR model.**

a. Chandy–Misra–Haas's distributed deadlock detection algorithm for the OR model, which is based on the approach of diffusion-computation.

β. 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*.

**38. What is the purpose of associating timestamp with events in Lamport's algorithm? (NOV/DEC 2021)**

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.

**39. What are the different models of deadlocks? (Nov/Dec 2020) (Apr/May 2021)**

i. Single model
ii. AND model
iii. OR model
iv. AND-OR model

## UNIT IV RECOVERY & CONSENSUS

Consensus and Agreement Algorithms: Problem Definition – Overview of Results – Agreement in a Failure – Free System – 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.

### 1. Explain in detail about Checkpoint and Rollback Recovery.

✓ Distributed systems today are ubiquitous and enable many applications, including client–server systems, transaction processing, the World Wide Web, and scientific computing, among many others.

✓ Distributed systems are not fault-tolerant and the vast computing potential of these systems is often hampered by their susceptibility to failures.

✓ Many techniques have been developed to add reliability and high availability to distributed systems.

✓ These techniques include transactions, group communication, and rollback recovery.

✓ These techniques have different tradeoffs and focus.

✓ Rollback recovery treats a distributed system application as a collection of processes that communicate over a network.

✓ It 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*, and the procedure of restarting from a previously check pointed state is called *rollback recovery.*

✓ A checkpoint can be saved on either the stable storage or the volatile storage depending on the failure scenarios to be tolerated.

✓ In distributed systems, rollback recovery is complicated because messages induce inter-process dependencies during failure-free operation.

✓ Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called rollback *propagation*.

✓ To see why rollback propagation occurs, consider the situation where 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* because they would show that Message *m* was received without being sent,

✓ It is impossible in any correct failure-free execution. This phenomenon of cascaded rollback is called the domino effect.

✓ 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 check pointing.*

✓ It is obviously desirable to avoid the domino effect and therefore several techniques have been developed to prevent it.

✓ One such technique is *coordinated check- pointing* where processes coordinate their checkpoints to form a system-wide consistent state.

✓ In case of a process failure, the system state can be restored to such a consistent set of checkpoints, preventing the rollback propagation.

✓ Alternatively, *communication-induced check pointing* forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes.

✓ Checkpoints are taken such that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect.

✓ The approaches discussed so far implement *checkpoint-based rollback* recovery, which relies only on checkpoints to achieve fault-tolerance.

✓ *Log- based* rollback recovery combines check pointing with logging of non-deterministic 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.*

✓ By logging and replaying the non- deterministic events in their exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed.

✓ Log-based rollback recovery in general enables a system to recover beyond the most recent set of consistent checkpoints.

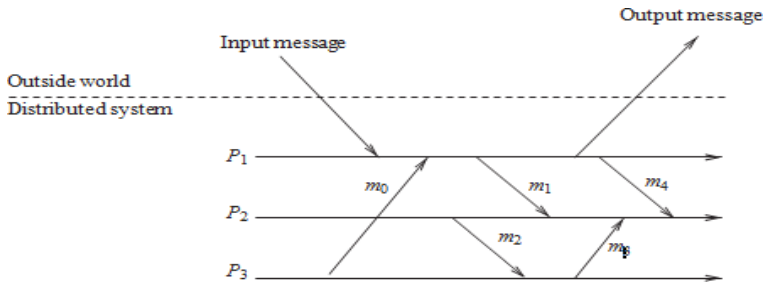
✓ It is therefore particularly attractive for applications that frequently interact with the *outside world*, which consists of input and output devices that cannot rollback.

## 2. Explain distributed system with an example of three processes.

- A distributed system consists of a fixed number of processes, $P_1$, $P_2$….$P_N$, which communicate only through messages.
- Processes cooperate to execute a distributed application and interact with the outside world by receiving and sending input and output messages, respectively.
- Figure 4.1 shows a system consisting of three processes and interactions with the outside world.
- Rollback-recovery protocols generally make assumptions about 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.

**Figure 4.1** An example of a distributed system with three processes.




- The choice between these two assumptions usually affects the complexity of check pointing and failure recovery.
- A generic correctness condition for rollback-recovery can be defined as follows "a system recovers correctly if its internal state is consistent with the observable behavior of the system before the failure."
- Rollback-recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world.

**A local checkpoint**

- In distributed systems, 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 and the event of recording the state of a process is called local check pointing.
- The contents of a checkpoint depend upon the application context and the check

pointing method being used.

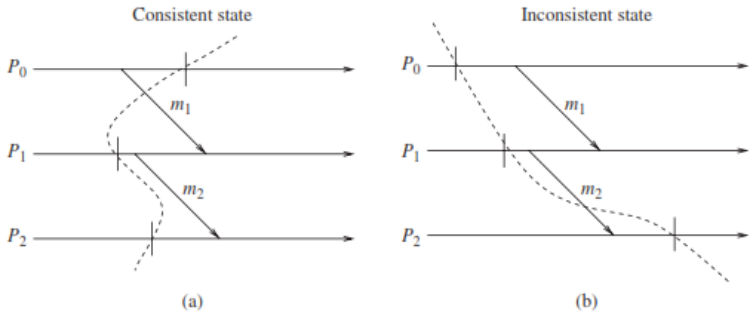
- Depending upon the check pointing method used, a process may keep several local checkpoints or just a single checkpoint at any time.
- We assume that a process stores all local checkpoints on the stable storage so that they are available even if the process crashes.
- We also assume that a process is able to roll back to any of its existing local checkpoints and thus restore to and restart from the corresponding state.
- Let $C_{i,k}$ denote the kth local check point at process Pi.
- Generally, it is assumed that a process $P_i$ takes a check point $C_i$, before it starts Execution. A local checkpoint is shown in the process-line by the symbol "|".

**Consistent system states**

- A global state of a distributed system is a collection of the individual states of all participating processes and the states of the communication channels.
- Intuitively, a consistent global state is one that may occur during a failure-free execution of a distributed computation.
- More precisely, a *consistent system state is* one in which a process's state reflects a message receipt, then the state of the corresponding sender must reflect the sending of that message.

**Figure 4.2** Examples of consistent and inconsistent states

For instance, Figure shows two examples of global states.



- The state in Figure (a) is consistent and the state in Figure(b) is inconsistent. Note that the consistent state in Figure (a) shows message m1 to have been sent but not yet received, but that is alright.
- The state in Figure (a) is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event.
- The state in Figure b) is inconsistent because process $P2$ is shown to have received $m2$ but the state of process $P1$ does not reflect having sent it.
- Such a state is impossible in any failure-free, correct computation. Inconsistent states occur because of failures.
- For instance, the situation shown in Figure (b) may occur if process $P1$ fails after sending message $m2$ to process $P2$ and then restarts at the state shown in Figure (b).
- Thus, a local checkpoint is a snapshot of a local state of a process and a global checkpoint is a set of local checkpoints, one from each process.
- A consistent global checkpoint is a global checkpoint such that no message is sent by a process after taking its local checkpoint that is received byanother process before taking its local checkpoint.
- The consistency of global checkpoints strongly depends on the flow of messages exchanged by processes and an arbitrary set of local check points at processes may not format consistent global check point.

The fundamental goal of any rollback-recovery protocol is to bring the system to a consistent state after a failure. The reconstructed consistent state is not necessarily one that occurred before the failure.

**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.

- If a failure occurs, the outside world cannot be expected to roll back.
- For example, a printer cannot roll back the effects of printing a character, and an automatic teller machine cannot recover the money that it dispensed to a customer.
- To simplify the presentation of how rollback-recovery protocols interact with the outside world, we model the latter as a *special* process that interacts with the rest of the system through message passing. We call this special process the "outside world process" (OWP).

**3. Explain Different types of messages in Distributed systems in detail.**

- A process failure and subsequent recovery may leave messages that were perfectly received (and processed) before the failure in abnormal states.
- This is because a rollback of processes for recovery may have to roll back the send and receive operations of several messages.
- We identify several types such messages using the example shown in Figure. shows an example consisting of four processes.

□ Process P1 fails at the point indicated and the whole system recovers to the state indicated by the recovery line; that is, to global state {C1, 8, C2, 9, C3, 8, C4, 8}.

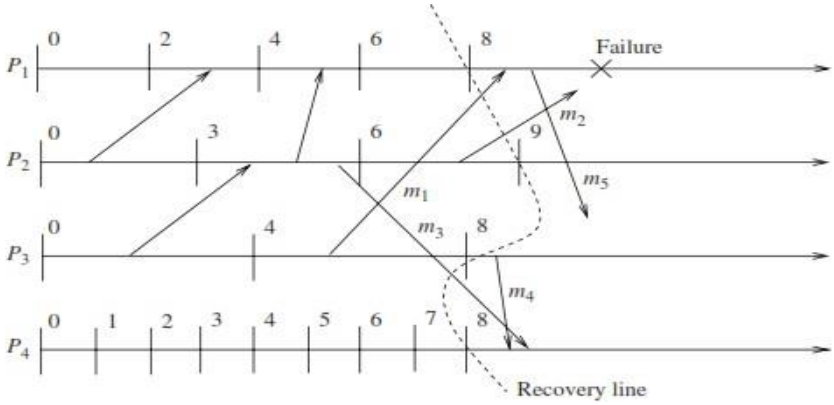



Figure 4.3 Different types of Messages

**In-transit messages**

- In Figure 4.3, the global state {$C_1$, 8, $C_2$, 9, $C_3$, 8, $C_4$, 8} shows that message $m_1$ has been sent but not yet received.
- We call such a message an in-*transit message*. Message $m_2$ is also an in-transit message.

**Lost messages**

- Messages whose send is not undone but receive is undone due to rollback are called *lost* messages.
- This type of messages occurs when the process rolls back to a checkpoint prior to reception of the message while the sender does not rollback beyond the send operation of the message.
- In Figure 13.3, message $m_1$ is a lost message.

**Delayed messages**

- Messages whose receive are not recorded because the receiving process was either down or the message arrived after the rollback of the receiving process, are called *delayed* messages.
- For example, messages $m_2$ and $m_5$ in Figure 13.3 are delayed messages.

**Orphan messages**

- Messages with receive recorded but message send not recorded are called *orphan* messages.
- For example, a rollback might have undone the send of such messages, leaving the receive event intact at the receiving process.
- Orphan messages do not arise if processes roll back to a consistent global state.

**Duplicate messages**

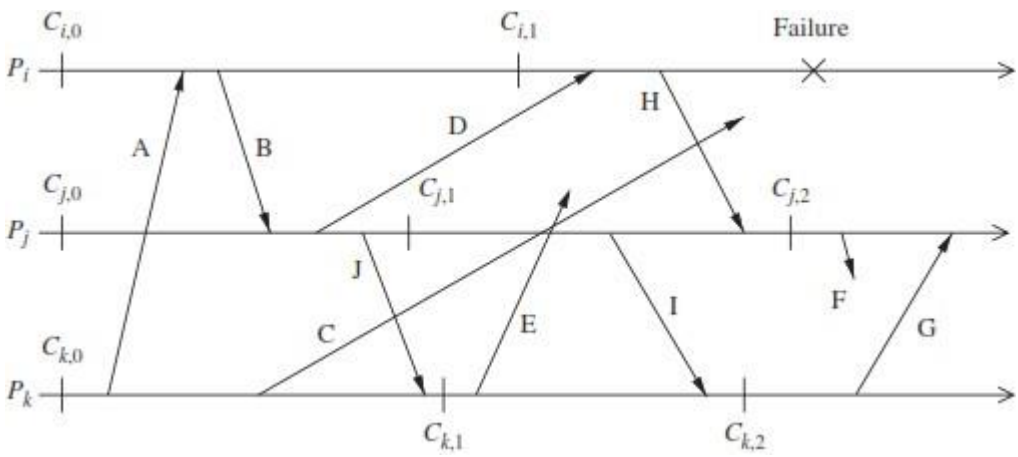
- Duplicate messages arise due to message logging and replaying during process recovery.
- For example, in Figure message m4 was sent and received before the rollback. However, due to the rollback of process P4 to C4, 8 and process P3 to C3, 8, both send and receipt of message m4 are undone.
- When process P3 restarts from C3,8, it will resend message m4. Therefore, P4 should not replay message m4 from its log.
- If P4 replays message m4, then message m4 is called a *duplicate* message. Message m5 is an excellent example of a duplicate message.
- No matter what, the receiver of m5 will receive a duplicate m5message.

**4. What are the Issues in failure recovery? (or) Write about the issues in failure recovery and discuss about any two recovery mechanisms. (NOV/DEC 2020/APR.MAY 2021) (NOV/DEC 2021)**

- In a failure recovery, we must not only restore the system to a consistent state, but also appropriately handle messages that are left in an abnormal state due to the failure and recovery.
- We now describe the issues involved in a failure recovery with the help of a distributed computation shown in Figure 4.5.
- The computation comprises of three processes $P_i$, $P_j$, and $P_k$, connected through a communication network.
- The processes communicate solely by exchanging messages over fault-free,
- FIFO communication channels.
- Processes $P_i$, $P_j$, and $P_k$ have taken check- points $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}\}$, respectively, and these processes have exchanged messages A to J as shown in Figure4.5.
- Suppose process $P_i$ fails at the instance indicated in the figure.
- All the contents of the volatile memory of $P_i$ are lost and, after $P_i$ has recovered from the failure, the system needs to be restored to a consistent global state from where the processes can resume their execution.

- Process Pi's state is restored to a valid state by rolling it back to its most recent checkpoint Ci, 1.
- To restore the system to a consistent state, the process Pj rolls back to checkpoint Cj,1 because the rollback of process Pi to checkpoint Ci,1 created an orphan message H (the receive event of H is recorded at process Pj while the send event of H has been undone at process Pi).
- Note that process Pj does not roll back to checkpoint Cj,2 but to checkpoint Cj,1, because rolling back to checkpoint Cj,2does not eliminate the orphan message
- Even this resulting state is not a consistent global state, as an orphan message I is created due to the rollback of process Pjto check point Cj,1.To eliminate this orphan message, process Pk rolls back to checkpoint Ck, 1.

**Figure 4.4 Illustration of issues in failure recovery.**




- Lost messages like D can be handled by having processes keep a message logof all the sent messages.
- So when a process restores to a checkpoint, it replays the messages from itslog to handle the lost message problem.

- However, message logging and message replaying during recovery can result in duplicate messages.
- In the example shown restored global state {Ci, 1, Cj,1, Ck,1} is a consistent state as it is free from orphan messages.
- Although the system state has been restored to a consistent state,

Several messages are left in an erroneous state which must be handled correctly.

- Messages A, B, D, G, H, I, and J had been received at the points indicated in the figure and messages C, E, and F were in transit when the failure occurred.
- Restoration of system state to checkpoints {Ci, 1, Cj, 1, Ck, 1} automatically handles messages A, B, and J because the send and receive events of
- Messages A, B, and J have been recorded, and both the events for G, H, and I have been completely undone.
- These messages cause no problem and we call messages A, B, and J normal messages and messages G, H, and I vanished messages.
- Messages C, D, E, and F are potentially problematic. Message C is in transit during the failure and it is a delayed message.
- The delayed message C has several possibilities: C might arrive at process $P_i$ before it recovers, it might arrive while $P_i$ is recovering, or it might arrive after $P_i$ has completed recovery.
- Each of these cases must be dealt with correctly.
- Message D is a lost message since the send event for D is recorded in the restored state for process $P_j$, but the receive event has been undone at process$P_I$.
- Process $P_j$ will not resend D without an additional mechanism, since the send D at $P_j$ occurred before the checkpoint and the communication system Successfully delivered D.
- Messages E and F are delayed orphan messages and pose perhaps the most serious problem of all the messages.

- When messages E and F arrive at their respective destinations, they must be discarded since their send events have been undone.
- Processes, after resuming execution from their checkpoints, will generate both of these messages, and recovery techniques must be able to distinguish between messages like C and those like E and F.
- When process $P_j$ replays messages from its log, it will regenerate message J. Process $P_k$, which has already received message J, will receive it again, Thereby causing inconsistency in the system state.
- Therefore, these duplicate messages must be handled properly. Overlapping failures further complicate the recovery process.
- A process $P_j$ that begins rollback/recovery in response to the failure of a process $P_i$ can itself fail and develop amnesia with respect process $P_i$'s failure; that is, process $P_j$ can act in a fashion that exhibits ignorance ofprocess $P_i$'s failure.
- If overlapping failures are to be tolerated, a mechanism must be introduced to deal with amnesia and the resulting inconsistencies.

## 4. Elaborate Checkpoint-based recovery in detail.

- In the checkpoint-based recovery approach, the state of each process and the communication channel is check pointed 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.
- Checkpoint-based protocols are therefore less restrictive and simpler to implement than log-based rollback recovery.
- However, checkpoint-based rollback recovery does not guarantee that pre-failure execution can be deterministically regenerated after a rollback.
- Therefore, checkpoint-based rollback recovery may not be suitable for applications

that require frequent interactions with the outside world.

- Checkpoint-based rollback-recovery techniques can be classified into three categories: *uncoordinated check pointing*, *coordinated check pointing*, and *communication-induced check pointing*.

**Uncoordinated check pointing**

- In uncoordinated check pointing, each process has autonomy in deciding when to take checkpoints.
- This eliminates the synchronization overhead as there is no need for coordination between processes and it allows processes to take checkpoints when it is most convenient or efficient.
- The main advantage is the lower runtime overhead during normal execution, because no coordination among processes is necessary.
- Autonomy in taking checkpoints also allows each process to select appropriate checkpoints positions.
- However, uncoordinated check pointing has several shortcomings.
- First, there is the possibility of the domino effect during a recovery, which may cause the loss of a large amount of useful work.
- Second, recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints.

- Since no coordination is done at the time the checkpoint is taken, checkpoints taken by a process may be *useless* checkpoints. (A useless checkpoint is never a part of any global consistent state.) Useless checkpoints are undesirable because they incur overhead and do not contribute to advancing the recovery line.

 Third, uncoordinated check pointing forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the checkpoints that are no longer required.

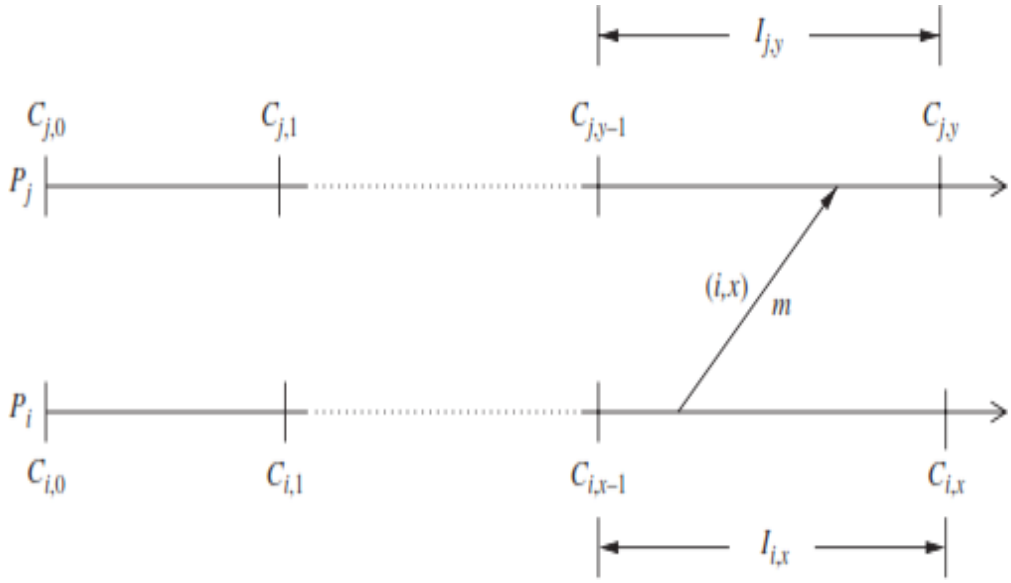
- Fourth, it is not suitable for applications with frequent output commits because

these require global coordination to compute the recovery line, negating much of the advantage of autonomy.

As each process takes checkpoints independently, we need to determine a consistent global checkpoint to rollback to, when a failure occurs.

- The main disadvantage of this method is that large latency is involved in committing output, as a global checkpoint is needed before a message is sent to the OWP. Also, delays and overhead are involved every time a new global checkpoint is taken.

**Figure 4.5** Checkpoint index and checkpoint interval



.

**Blocking coordinated check pointing**

- A straightforward approach to coordinated check pointing is to block communications while the check pointing protocol executes.
- After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire check pointing 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.
- After the coordinator receives acknowledgments from all processes, it broadcasts a commit message that completes the two-phase check pointing protocol.
- 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.

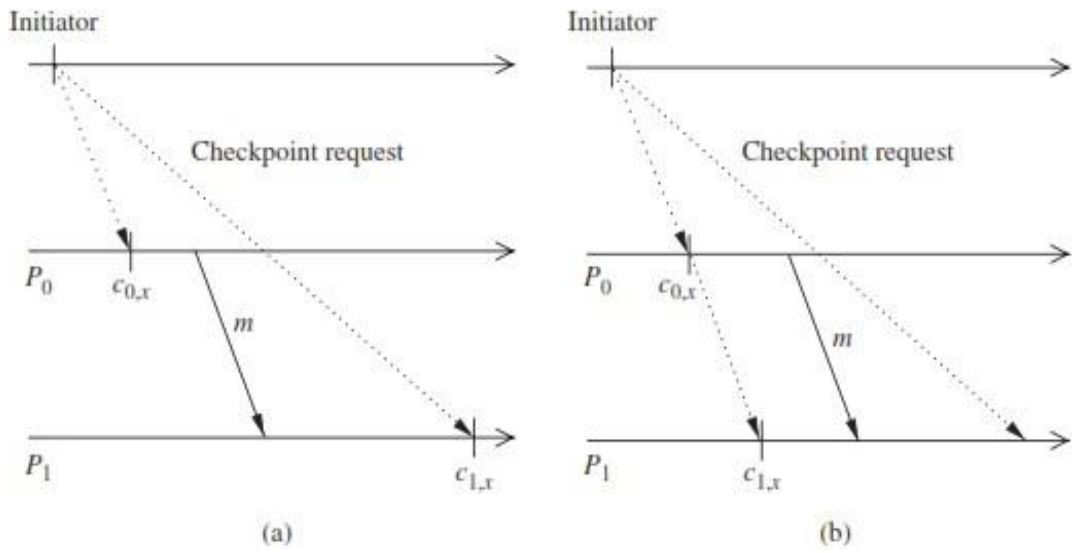
**Non-blocking checkpoint coordination**

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

**Figure: 4.6 Non-**blocking coordinated check pointing:

(a) checkpoint inconsistency;

(b) a solution with FIFO channels




In this algorithm, the initiator takes a checkpoint and sends a marker (acheckpoint request) on all outgoing channels.

- Each process takes a checkpoint upon receiving the first marker and sends the marker on all outgoing channels before sending any application message.
- The protocol works assuming the channels are reliable and FIFO.
- If the channels are non-FIFO, the following two approaches can be used: first, the marker can be piggybacked on every post-checkpoint message.
- When a process receives an application message with a marker, it treats it as if it has received a marker message, followed by the application message.
- Alternatively, checkpoint indices can serve the same role as markers, where a checkpoint is triggered when the receiver's local checkpoint index is lower than the piggybacked check point index.
- Coordinated check pointing requires all processes to participate in every checkpoint. This requirement generates valid concerns about its scalability.
- It is desirable to reduce the number of processes involved in a coordinated check pointing session.
- This can be done since only those processes that have communicated with the

checkpoint initiator either directly or indirectly since the last check point need to take new checkpoints.

**Impossibility of min-process non-blocking check pointing**

- A min-process, non-blocking check pointing 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.
- Clearly, such check pointing algorithms will be very attractive. Cao and Singhal showed that it is impossible to design a min-process, non-blocking check pointing algorithm.
- Of course, the following type of min-process check pointing algorithms is possible.
- The algorithm consists of two phases. During the first phase, the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request.
- Upon receiving the request, each process in turn identifies all processes it has communicated with since the last checkpoint and sends them a request, and so on, until no more processes can be identified.
- During the second phase, all processes identified in the first phase take a checkpoint.
- The result is a consistent checkpoint that involves only the participating processes.
- In this protocol, after a process takes a checkpoint, it cannot send any message until the second phase terminates successfully, although receiving a message after the checkpoint has been taken is allowable.

**Communication-induced check pointing**

- *Communication-induced check pointing* is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently.

- Processes may be forced to take additional checkpoints (over and above their autonomous checkpoints), and thus process independence is constrained to guarantee the eventual progress of the recovery line.
- Communication-induced check pointing reduces or completely eliminates the useless checkpoints. In communication-induced check pointing, processes take two types of checkpoints, namely, autonomous and forced checkpoints.
- The checkpoints that a process takes independently are called *local* checkpoints, while those that a process is forced to take are called *forced* checkpoints.
- Communication-induced check pointing piggybacks protocol-related information on each application message.
- The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line.
- The forced checkpoint must be taken before the application may process the contents of the mess- sage, possibly incurring some latency and overhead.
- It is therefore desirable in these systems to minimize the number of forced checkpoints. In contrast with coordinated check pointing, no special coordination messages are exchanged.

**Model-based check pointing**

- Model-based check pointing prevents patterns of communications and check points that could result in inconsistent states among the existing checkpoints.
- A process detects the potential for inconsistent checkpoints and independently forces local checkpoints to prevent the formation of undesirable patterns.
- A forced checkpoint is generally used to prevent the undesirable patterns from occurring.
- No control messages are exchanged among the processes during normal

operation.

- All information necessary to execute the protocol is piggy- backed on application messages.
- The decision to take a forced checkpoint is done locally using the information available.

**Index-based check pointing**

- Index-based communication-induced check pointing assigns monotonically increasing indexes to checkpoints, such that the checkpoints having the same index at different processes form a consistent state.
- Inconsistency between checkpoints of the same index can be avoided in a lazy fashion if indexes are piggybacked on application messages to help receivers decide when they should take a forced a checkpoint.

**5. Explain Koo–Tough coordinated check pointing algorithm in detail. Illustrate briefly the two kinds of Check points for Check point algorithm. (Nov/Dec 2022)**

- Koo and Toque's coordinated check pointing and recovery technique takes a consistent set of checkpoints and avoids the domino effect and livelock problems during the recovery.
- Processes coordinate their local checkpointing actions such that the set of all checkpoints in the system is consistent.

**The checkpointing algorithm**

- The checkpoint algorithm makes the following assumptions about the distributed system: processes communicate by exchanging messages through communication channels. Communication channels are FIFO.
- It is assumed that end- to-end protocols (such as the sliding window protocol) exist to cope with message loss due to rollback recovery and communication failure. Communication failures do not partition the network.
- The checkpoint algorithm takes two kinds of checkpoints on the stable storage: permanent and tentative.

- A permanent checkpoint is a local check- point at a process and is a part of a consistent global checkpoint.
- A tentative checkpoint is a temporary checkpoint that is made a permanent checkpoint on the successful termination of the checkpoint algorithm. In case of a failure, processes roll back only to their permanent checkpoints for recovery.

- The check pointing algorithm assumes that a single process invokes the algorithm at any time to take permanent checkpoints.
- The algorithm also assumes that no process fails during the execution of the algorithm.
- The algorithm consists 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 $P_i$ whether it succeeded in taking a tentative checkpoint.
- A process says "no" to a request if it fails to take a tentative checkpoint, which could be due to several reasons, depending upon the underlying application.
- If $P_i$ learns that all the processes have successfully taken tentative checkpoints, $P_i$ decides that all tentative checkpoints should be made permanent; otherwise, $P_i$ decides that all the tentative checkpoints should be discarded.

***Second phase***

- $P_i$ informs all the processes of the decision it reached at the end of the first phase.
- A process, on receiving the message from $P_i$, will act accordingly. Therefore, either all or none of the processes advance the checkpoint by taking permanent checkpoints.
- The algorithm requires that after a process has taken a tentative checkpoint, it cannot send messages related to the underlying computation until it is

informed of $P_i$'s decision.

**Correctness**

- A set of permanent checkpoints taken by this algorithm is
- consistent because of the following two reasons: first, either all or none of the processes take permanent checkpoints; second, no process sends a message after taking a tentative checkpoint until the receipt of the initiating process's decision, as by then all processes would have taken checkpoints.
- Thus, a situation will not arise where there is a record of a message being received but there is no record of sending it. Thus, a set of checkpoints taken will always be inconsistent.

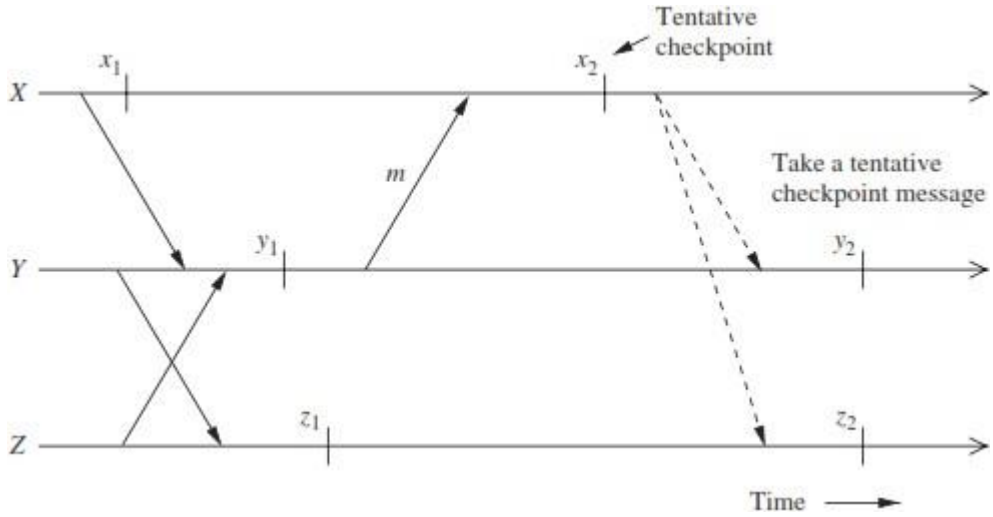
**An optimization**

- Note that the above protocol may cause a process to take a checkpoint even when it is not necessary for consistency.
- Since taking a checkpoint is an expensive operation, we would like to avoid taking checkpoints if it is not necessary.
- Consider the example shown in Figure The set $\{x_1, y_1, z_1\}$ is a consistent set of checkpoints.
- Suppose process X decides to initiate the check- pointing algorithm after receiving message m.
- It takes a tentative checkpoint $x_2$ and sends "take tentative checkpoint" messages to processes Y and Z, causing Y and Z to take checkpoints $y_2$ and $z_2$, respectively.
- Clearly, $\{x_2, y_2, z_2\}$ forms a consistent set of checkpoints. Note, however, that $\{x_2, y_2, z_1\}$ also forms a consistent set of checkpoints.
- In this example, there is no need for process Z to take checkpoint $z_2$ because Z has not sent any message since its last checkpoint.
- However, process Y must take a checkpoint since it has sent messages since its last checkpoint.

**The rollback recovery algorithm**

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

**Figure : 4.7 Example of Checkpoints taken unnecessarily**




***First phase***

- An initiating process $P_i$ sends a message to all other processes to Check if they all are willing to restart from their previous checkpoints.

  A process may reply "no" to a restart request due to any reason (e.g., it is already participating in a checkpoint or recovery process initiated by some other process).
- If $P_i$ learns that all processes are willing to restart from their previous

checkpoints, $P_i$ decides that all processes should roll back to their previous checkpoints.

- Otherwise, $P_i$ aborts the rollback attempt and it may attempt a recovery at a later time.

***Second phase***

$P_i$ propagates its decision to all the processes. On receiving

- $P_i$'s decision, a process acts accordingly.
- During the execution of the recovery algorithm, a process cannot send messages related to the underlying computation while it is waiting for $P_i$'s decision.

**Correctness**

All processes restart from an appropriate state because, if they decide to restart, they resume execution from a consistent state (the check pointing algorithm takes a consistent set of checkpoints).

**An optimization**

The above recovery protocol causes all processes to roll back.Irrespective of whether a process needs to roll back or not. Consider the example shown in Figure In the event of failure of process X, the above protocol will require processes X, Y, and Z to restart from checkpoints $x_2$, $y_2$, and $z_2$, respectively. However, note that process Z need not roll back because there has been no interaction between process Z and the other two processes since the last checkpoint at Z.

**6. Explain–Venkatesan algorithm for asynchronous check pointing and recovery Algorithm in detail.**

We now describe the algorithm of Juan and Venkatesan for recovery in asystem that employs asynchronous check pointing.

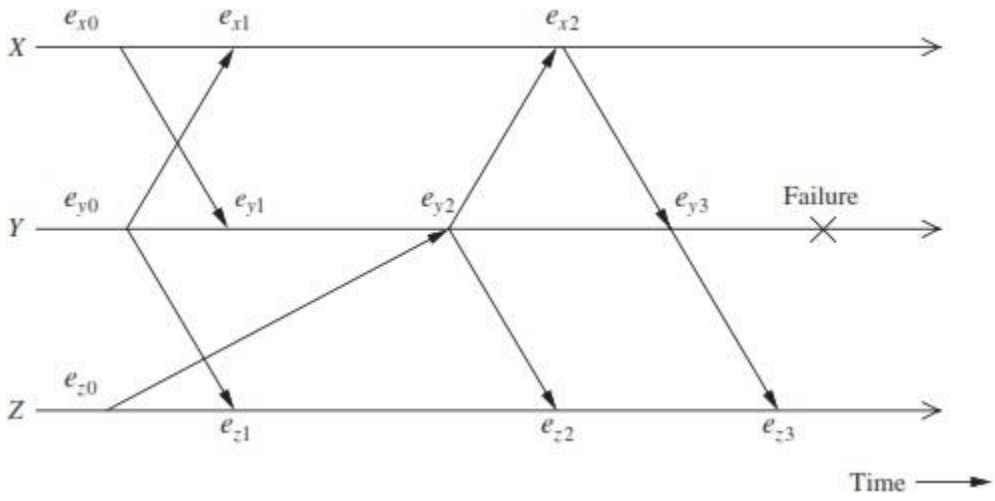
**System model and assumptions**

- The algorithm makes the following assumptions about the underlying system: the communication channels are reliable, deliver the messages in FIFO order,and

have infinite buffers.

- The message transmission delay is arbitrary, but finite. The processors directly connected to a processor via communication channels are called its neighbors.
- The underlying computation or application is assumed to be event-driven: a processor P waits until a message m is received, it processes theMessage m, changes its state from s to $s^t$, and sends zero or more messages to someOf its neighbors.
- Then the processor remains idle until the receipt of the next message.
- The new state $s^t$ and the contents of messages sent to its neighbors depend on state s and the contents of message m. The events at a processor are identified by unique monotonically increasing numbers, ex0, ex1, ex2,
- To facilitate recovery after a process failure and restore the system to aconsistent
- State, two types of log storage are maintained, volatile log and stable log. Accessing the volatile log takes less time than accessing the stable

**Figure: 4.8** An event-driven computation



Asynchronous check pointing

- After executing an event, a processor records a triplet {s, m, msgs sent} in its volatile storage, where s is the state of the processor.
- Before the event, m is the message (including the identity of the sender of m, denoted as messenger) whose arrival caused the event, and msqs sent is the set of messages that were sent by the processor during the event.

**The recovery algorithm**

**Notation and data structure**

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

- RCVDi←j(CkPti)represents the number of messages received by processor $p_i$ from processor $p_j$ , from the beginning of the computation until the checkpoint $CkPt_i$.
- $SENT_{i\rightarrow j}(Copt_i)$ represents the number of messages sent by processor $p_i$ to processor $p_j$, from the beginning of the computation until the checkpoint $CkPt_i$.

**Description of the algorithm**

- When a processor restarts after a failure, it broadcasts a ROLLBACK message that it has failed.[1] the recovery algorithm at a processor is initiated when it restarts after a failure or when it learns of a failure at another processor.
- Because of the broadcast of ROLLBACK messages, the recovery algorithm is initiated at all processors. The algorithm is shown in Algorithm
- 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 $p_i$ does the following:

(i) based on the state Ck Pti it was rolled back in the(k−1)th iteration, it computes SENTi→j (CkPti) for each neighbor pj and sends this value in a ROLLBACK message to that neighbor; and

(ii) pi waits for and processes ROLLBACK messages that it receives from its neighbors in kth iteration and determines a new recovery point CkPtI for pi based on information in these messages.

- At the end of each iteration, at least one processor will roll back to its final recovery point, unless the current recovery points are already consistent.

Procedure M+gIbackRecover : processory, executes
thefulTuu'ing: STEP(a)
If {xDC8ssor /ii L recovering after a failure then
*C.km,*— latest event logged in the stable storage
Else
*Copy,*— latest e endThai took place inpt (The laziest event at y, can be either in stable or in s volatile storage.}

STEP (b)
fur 1 = I to .N' *N* is the number of processors in thesystem) do
For etch neighboring processor p, do
Compute IE *NT* _,(Chi )
Send*u* *ROLLBACK (I, SE 4T, (CkPt, j)*run*asa e* to p end

for
Forever j *ROLLBACK [j,*c) nzss%e received. Tram a neighbor J du
if R.CVD *I CA Pt, )* >c (1 implied the presence of orphanmexsapex) then
Find the latest extent e such Thai*DC\'Dip(e)* *c* (such an event emay be ii\ the volatile proratenr ratable storage.}
End if

Endfor (for i)

**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.

**Synchronous/asynchronous communication**

If a failure-prone process chooses to send a message to process $P_i$ but fails, then $P_i$ cannot detect the non-arrival of the message in an asynchronous system because this scenario containing some default data, and then proceeding with the next round of the algorithm.

**9. Explain Consensus and agreement algorithms in detail. (or) Describe the role of consensus and agreement algorithms in distributed, system along with its underlying structure, benefits and an example. (NOV/DEC2020) (APR.MAY 2021) (NOV/DEC 2021) Illustrate different types of problems in distributed system and how to prevent them. (Nov/Dec 2022) Compare the results and lower bounds on agreement on solving the consensus problem under different assumptions. (Apr/May 2023)**

**Problem definition**

- Agreement among the processes in a distributed system is a fundamental requirement for a wide range of applications.
  - Many forms of coordination require the processes to exchange information to negotiate with one another and eventually reach a common understanding or agreement, before taking application-specific actions.

- A classical example is that of the *commit* decision in database systems, wherein the processes collectively decide whether to *commit* or *abort* a transaction that they participate in.
- We first state some assumptions underlying our study of agreement algorithms:

**Sender identification**

- A process that receives a message always knows the identity of the sender process.
- This assumption is important– because even with Byzantine behavior, eventhough the payload of the message can contain fictitious data sent by a malicious sender, the underlying network layer protocol scan reveal the true identity of the sender process.
- When multiple messages are expected from the same sender in a single round
- Implicitly assume a scheduling algorithm that sends these messages in sub-rounds, so that each message sent within the round can be uniquely identified.

**Channel reliability**

- The channels are reliable, and only the process say fail (less than one of various failure models).This is a simplifying assumption in our study. As we will see even with this simplifying assumption, the agreement problem is either unsolvable, or solvable in a complex manner.

**Authenticated vs. non-authenticated messages**

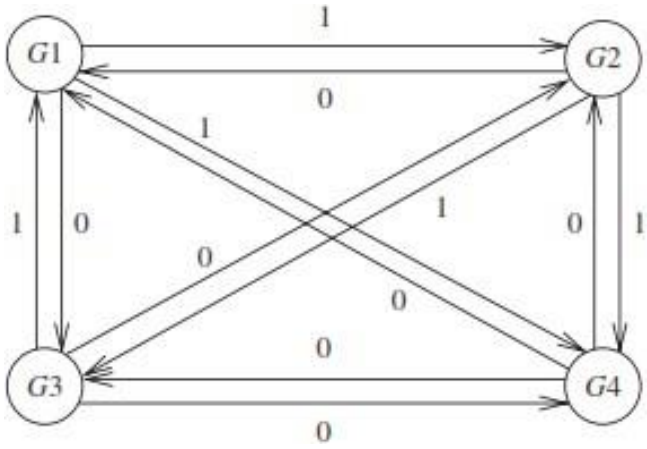
- In our study, we will be dealing only with *unauthenticated* messages. With unauthenticated messages, when a faulty process relays a message to other processes,
- It can forge the message and claim that it was received from another process, and it can also tamper with the content so far received message before relaying it. When a process receives a message, it has no way to verify its authenticity. An unauthenticated message is also called an *oral* message or an *unsigned* message.
- Using authentication via techniques such as digital signatures, it is easier to solve the agreement problem because, if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the for gery or tampering. Thus, faulty processes can inflict less damage.

**Agreement variable**

The agreement variable may be Boolean or multi valued, and need not be an integer. When studying some of the more complex algorithms, we will use a Boolean variable. This simplifying assumption does not affect the results for other data types, but

helps in the abstraction while presenting the algorithms.

**Figure: 4.9 Byzantine generals sending confusing message**




- Four camps of the attacking army, each commanded by a general, are camped around the fort of Byzantium. they can succeed in attacking only if they attack simultaneously.
- Hence, they need to reach agreement on the time of attack. The only way they can communicate is to send messengers among themselves. The messengers model the messages. An asynchronous system is modeled by messengers taking an unbounded time to travel between two camps.

- A lost message is modeled by a messenger being captured by the enemy. A Byzantine process is modeled by a general being a traitor.
- The traitor will attempt to subvert the agreement- reaching mechanism, by giving misleading information to the other generals.
- For example, a traitor may inform one general to attackat10a.m., and inform the other.
- Generals to attack at noon. Or he may not send a message at all to some general. Likewise, he may tamper with the messages he gets from other generals, before relaying those messages.

**The Byzantine agreement and other problems**

- **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-faultyprocesses.
- **Termination:** Each non-faulty process must eventually decide on a value.

**The consensus problem**

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

**The interactive consistency problem**

- **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 $v_i$, then all non- faulty processes agree on $v_i$ as the i th element of the array A. If process j is faulty, then the non-faulty processes can agree on any value for A[j].
- **Termination** Each non-faulty process must eventually decide on the array A.

**Equivalence of the problems and notation**

The three problems defined above are equivalent in the sense that a solution to any one of them can be used as a solution to the other two problems. This equivalence can be shown using a reduction of each problem to the other two problems. If problem A is reduced to problem B, then a solution to problem B can be used as a solution to problem in conjunction with the reduction. Exercise asks the reader to show these reductions.

**Overview of results**

- ✓ Table gives an overview of the results and lower bounds on solving the consensus problem under different assumptions.
- ✓ It is worth understanding the relation between the consensus problem and the problem of attaining common knowledge of the agreement value. For the "no failure" case, consensus is attainable. Further, in a synchronous system, common knowledge of the consensus value is also attainable, whereas in the asynchronous
- ✓ case, concurrent common knowledge of the consensus value is attainable.
- ✓ Consensus is not solvable in asynchronous systems even if one process can fail by crashing. To circumvent this impossibility result, weaker variants

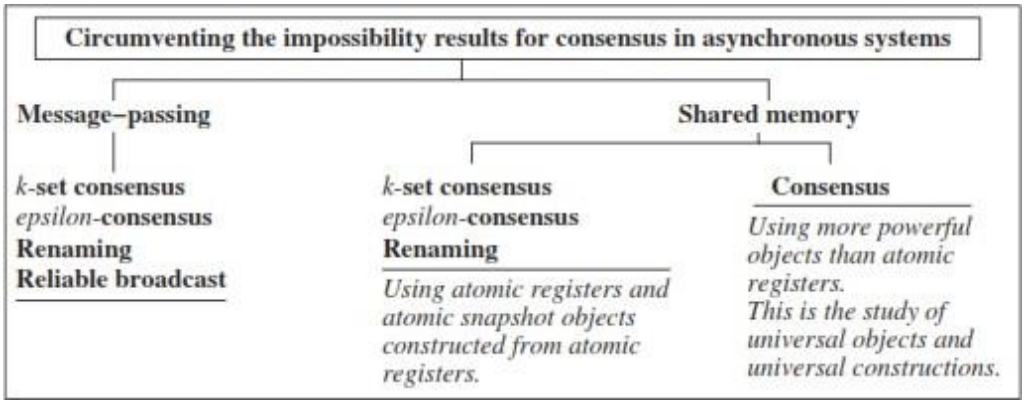
**Table 14.1** Overview of results on agreement. $f$ denotes number of failure-prone processes. $n$ is the total number of processes.

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

**Table 14.2** Some solvable variants of the agreement problem in an asynchronous system. The overhead bounds are for the given algorithms, and are not necessarily tight bounds for the problem.

| Solvable variants | Failure model and overhead | Definition |
|---|---|---|
| Reliable broadcast | Crash failures, $n > f$ (MP) | Validity, agreement, integrity conditions (Section 14.5.7) |
| $k$-set consensus | Crash failures, $f < k < n$ (MP and SM) | Size of the set of values agreed upon must be at most $k$ (Section 14.5.4) |
| $\epsilon$-agreement | Crash failures, $n \geq 5f + 1$ (MP) | Values agreed upon are within $\epsilon$ of each other (Section 14.5.5) |
| Renaming | Up to $f$ fail-stop processes, $n \geq 2f + 1$ (MP)<br>Crash failures, $f \leq n - 1$ (SM) | Select a unique name from a set of names (Section 14.5.6) |

**Figure: 4.9 Circumventing the impossibility results for Consensus in Asynchronous system**



**8. What are the Agreement carried out 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 – some simple examples being the majority, max, and min functions.
- ✓ Algorithms to collect the initial values and then distribute the decision may be based on the token circulation on a logical ring, or the three-phase tree-based broadcast– converge cast–broadcast, or direct communication with all nodes.
- ✓ In a synchronous system, this can be done simply in a constant number of rounds Further, common knowledge of the decision value can be obtained using an additional round.

✓ In an asynchronous system, consensus can similarly be reached in a conStint number of message hops. Further, concurrent common knowledge of the consensus value can also be attained, using any of the algorithms. Agreement in (message-passing) synchronous systems with failures

**Consensus algorithm for crash failures (synchronous system)**

✓ Consensus algorithm for n processes, where up to f processes, where f < n, may fail in the fail-stop model. Here, the consensus variable x is integer-valued. Each process has an initial value $x_i$.

✓ If up to f failures are to be tolerated, then the algorithm has f+1round.In each round,a process i sends the value of its variable $x_i$ to all other processes if that value has not been sent before. Of all the values received within the round and its own value $x_i$ at the start of the round, the process takes the minimum, and updates

$x_i$. After f + 1 round, the local value $x_i$ is guaranteed.

✓ To be the consensus value.

- 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

```
(global constants)
integer: f;                          // maximum number of crash failures tolerated
(local variables)
integer: x ⟵ local value;

(1) Process P_i (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)     y_j ⟵ value (if any) received from process j in this round;
(1e)     x ⟵ min(x, y_j);
(1f) output x as the consensus value.
```

**Algorithm:** Consensus with upto f fail-stop processes in a system of n processes, n>f

Code shown is for process $P_I$, $1 \leq i \leq n$.

- The *agreement condition* is satisfied because in the f+1rounds, there must be atleast one round in which no process failed. In this round, say round r, all the processes that have not failed so far succeed in broadcasting the values.

  All these processes take the minimum of the values broadcast and received in that round. Thus, the local values at the end of the round are the same, say $x^r$ for all non-failed processes. In further rounds, only this value may be sent by each Process at most once, and no process i will update its value $x^r$.

- The *validity condition* is satisfied because processes do not send fictitious

Values in this failure model. (Thus, a process that crashes has sent only Correct values until the crash.) 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.

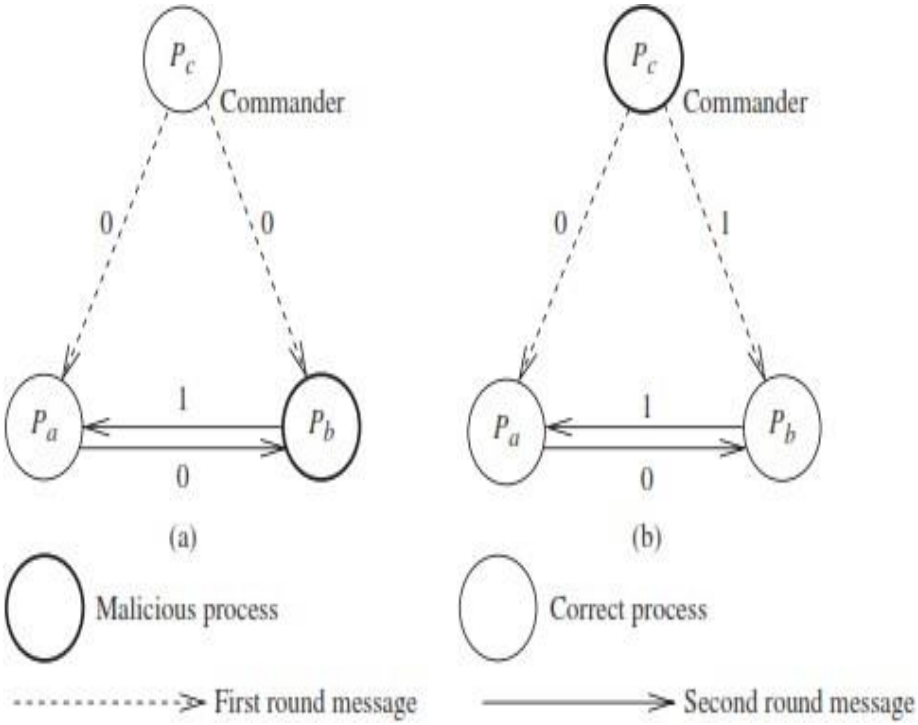



**Figure 4.10: Impossibility of achieving Byzantine agreement with n=3 processes and f=1 malicious process**

**Algorithm : Byzantine general algorithm-exponential number of unsigned messages, n > f Recursive formulation**

---

(variables)

**boolean**: $v \leftarrow$ initial value;

**integer**: $f \leftarrow$ 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.

---

**Table : Relationship between messages and rounds in the oral messages algorithm for the Byzantine agreement**

| Round number | A message has already visited | Aims to tolerate these many failures | Each message gets sent to | Total number of messages in round |
|---|---|---|---|---|
| 1 | 1 | $f$ | $n-1$ | $n-1$ |
| 2 | 2 | $f-1$ | $n-2$ | $(n-1)\cdot(n-2)$ |
| ... | ... | ... | ... | ... |
| $x$ | $x$ | $(f+1)-x$ | $n-x$ | $(n-1)(n-2)\ldots(n-x)$ |
| $x+1$ | $x+1$ | $(f+1)-x-1$ | $n-x-1$ | $(n-1)(n-2)\ldots(n-x-1)$ |
| ... | ... | ... | ... | ... |
| $f+1$ | $f+1$ | 0 | $n-f-1$ | $(n-1)(n-2)\ldots(n-f-1)$ |

(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 $h-1 = sizeof(L)$, 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)

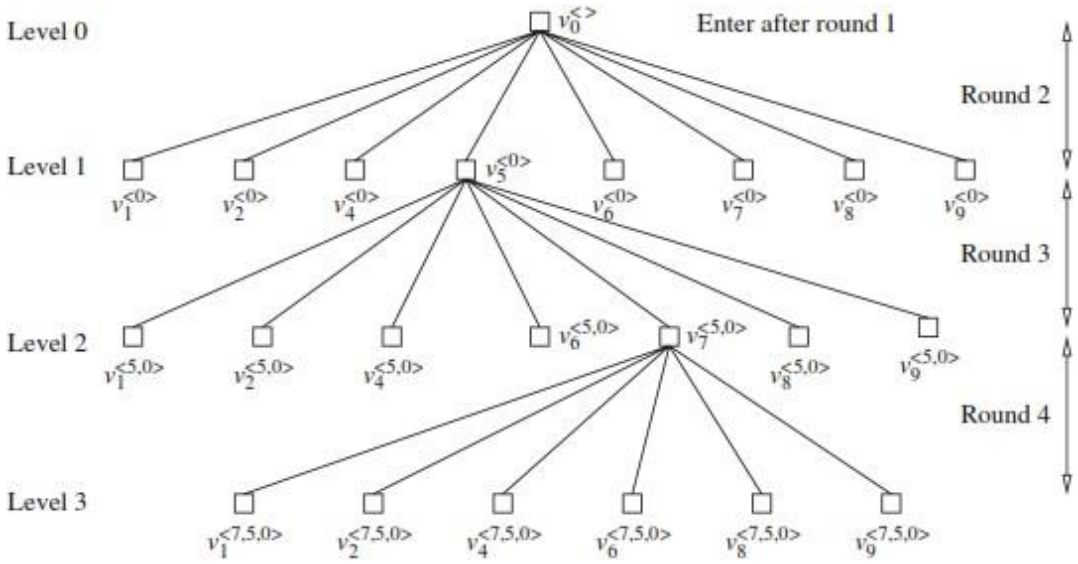
$OM(v, Dests, List, faulty)$, where the parameters are as in the recursive formulation.

(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) $v_{head(L)}^{tail(L)} \longleftarrow v$; // $sizeof(L) + faulty = f + 1$. 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 $v_x^L$ in level $level$, **do**
(2h) $v_x^L(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v_x^L, v_y^{concat(\langle x \rangle, L)})$;

**Algorithm: Byzantine generals algorithm – exponential number of unsigned messages, n > 3f. Iterative formulation. Code for Process Pi**

**Examples:**



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

Round 1: P0 sends its value to all process using Oral_Msg(3), including to P3.
Round 2: P3 sends 8 messages to others (excl. P0 and P3) using Oral_Msg(2). P3 also receives 8 messages.
Round 3: P3 sends 8 X 7 = 56 messages to all others using Oral_Msg(1); P3 also receives 56 messages.
Round 4: P3 sends 56 X 6 = 336 messages to all others using Oral_Msg(0); P3 also receives 336 messages. The received values are used as estimates of the majority function at this level of recursion.

*Complexity*

The algorithm requires $f+1$ rounds, an exponential amount of local memory, and

$$(n-1)+(n-1)(n-2)+\cdots+[(n-1)(n-2)\cdots(n-f-1)] \text{ messages.}$$

**Phase-king algorithm for consensus: polynomial (synchronous system)**

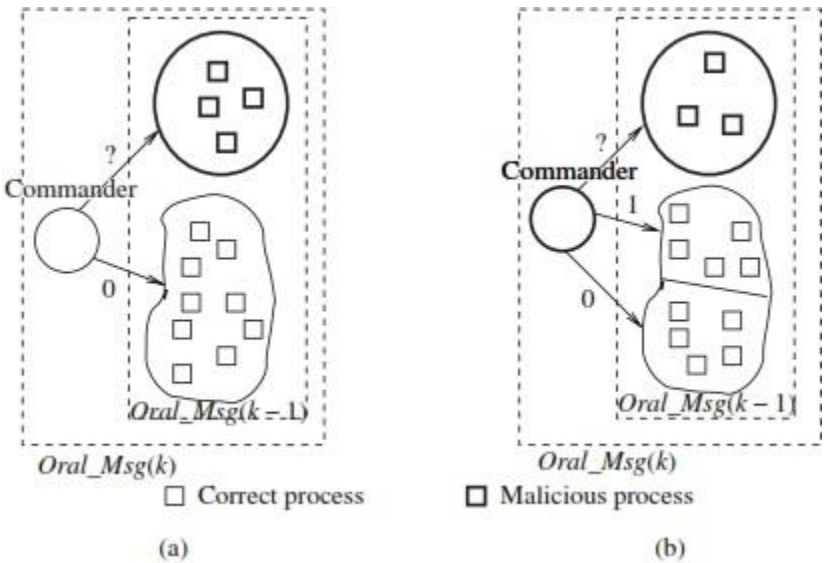



**Figure: 4.11 The effect of a loyal or disloyal commander in a system with n = 14 and f = 4. The sub-system that need to tolerate k and k-1 traitors are shown for two cases.**

**(a) Loyal Commander**

**(b) No assumption for Commander**

```
(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:
(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 value if no majority);
(1f)         mult ⟵ number of times that majority occurs;
(1g)     Execute the following round 2 actions:
(1h)         if i = phase then
(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.
```

**Algorithm: Phase king algorithm – Polynomial number of unsigned messages, n>4f. Code is for process $P_i$ , 1<=i<=n.**

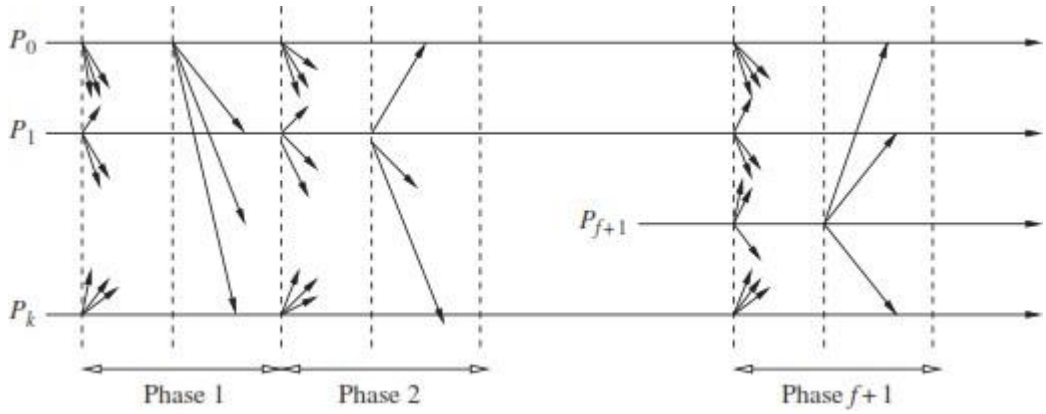



**Figure: 4.12 Message pattern for the phase king algorithm**

Operation

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

- Each phase has two rounds:
  - e in 1st round, each process sends its estimate to all other processes.
  - e 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.

*Complexity*

The algorithm requires $f+1$ phases with two sub-rounds in each phase, and $(f+1)[(n-1)(n+1)]$ messages.

**Agreement in Asynchronous message passing system with failure**

- In a failure-free async MP system, initial state is *monovalent* $\Longrightarrow$ consensus can be reached.
- In the face of failures, initial state is necessarily bivalent
- Transforming the input assignments from the all-0 case to the all-1 case, there must exist input assignments $\vec{I_a}$ and $\vec{I_b}$ that are 0-valent and 1-valent, resp., and that differ in the input value of only one process, say $P_i$. If a 1-failure tolerant consensus protocol exists, then:
  - Starting from $\vec{I_a}$, if $P_i$ fails immediately, the other processes must agree on 0 due to the termination condition.
  - Starting from $\vec{I_b}$, if $P_i$ fails immediately, the other processes must agree on 1 due to the termination condition.

  However, execution (2) looks identical to execution (1), to all processes, and must end with a consensus value of 0, a contradiction. Hence, there must exist at least one bivalent initial state.
- Consensus requires some communication of initial values.

- To transition from bivalent to monovalent step, must exist a critical step which allows the transition by making a decision
- Critical step cannot be local (cannot tell apart between slow and failed process) nor can it be across multiple processes (it would not be well-defined)
- Hence, cannot transit from bivalent to univalent state.

Wider Significance of Impossibility Result

- By showing reduction from consensus to problem X, then X is also not solvable under same model (single crash failure)
- E.g., leader election, terminating reliable broadcast, atomic broadcast, computing a network-wide global function using BC-CC flows, transaction commit.

# PART A

**1. What is check point and rollback recovery? (or) What do you mean by checkpoint? (Apr/May 2023)**

✓ The saved state is called **a checkpoint**, and the procedure of restarting from a previously check pointed state is called **rollback recovery.**

✓ A checkpoint can be saved on either the stable storage or the volatile storage depending on the failure scenarios to betolerated.

✓ In distributed systems, rollback recovery is complicated because messages induce inter-process dependencies during failure-free operation.

**2. Define rollback propagation?**

✓ A failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called a **rollback propagation.**

**3. Different types of messages**

i. In-transit messages
ii. Lost messages
iii. Delayed messages
iv. Orphan messages
v. Duplicate messages

**4. Define In-transit messages and lost messages?**

- **In-transit messages:** the global state $\{C_{1,8}, C_{2,9}, C_{3,8}, C_{4,8}\}$ shows that message m1 has been **sent but not yet received.** We call such a message an in-transit message. Message m2 is also an **in-transit message**.

- **Lost messages: Messages** whose **send is not undone but receive is undone due to rollback** are called **lost messages.**

**5. Define a local checkpoint and global checkpoint**

✓ In distributed systems, 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 and the event of n recording the state of a process is called **local check pointing.**

✓ The contents of a checkpoint depend upon the application context and the check pointing method being used.

✓ **A global checkpoint** is a set of local checkpoints, one from eachprocess.

**6. What are three categories of Checkpoint-based rollback-recovery?**

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

a. Uncoordinated check pointing,
b. coordinated check pointing, and
c. communication-induced check pointing

**11. Define uncoordinated checkpointing.**

✓ In uncoordinated check pointing, each process has autonomy in deciding when to take checkpoints.

✓ This eliminates the synchronization overhead as there is no need for coordination between processes and it allows processes to take checkpoints when it is most convenient or efficient.

**12. Define coordinated checkpointing.**

✓ In coordinated check pointing, processes arrange their check pointing activities so that all local

Checkpoints form a consistent global state.

✓ Coordinated check pointing simplifies recovery and is not susceptible to the domino effect, since

every process always restarts from its most recent checkpoint.

**13. What are the advantages of Uncoordinated check pointing? Advantages:**

✓ The main advantage is the lower runtime overhead during normal execution, because no coordination among processes is necessary.

✓ Autonomy in taking checkpoints also allows each process to select appropriate checkpoints positions.

**14. What are the disadvantages of Uncoordinated check pointing?**

✓ First, there is the possibility of the **domino effect during a recovery**, which may cause the loss of a large amount of usefulwork.

✓ Second, recovery from a failure is slow because processes need to iterate to find a consistent set of checkpoints.

✓ Third, uncoordinated check pointing forces each process to maintain multiple checkpoints, and to periodically invoke a garbage collection algorithm to reclaim the

checkpoints that are no longer required.

✓ Fourth, it is not suitable for applications with frequent output commits because these require global coordination to compute the recovery line, negating much of the advantage of autonomy.

**15. Define Communication-induced checkpointing?**

✓ Communication-induced check pointing is another way to avoid the domino effect, while allowing processes to take some of their checkpoints independently.

✓ Communication-induced check pointing reduces or completely eliminates the useless checkpoints.

✓ In communication-induced check pointing, processes take two types of checkpoints, namely, **autonomous and forced checkpoints**.

**16. What is the problem of coordinated checkpointing?**

**Disadvantages:**

✓ A problem with this approach is that the computation is blocked during the check pointing and therefore, non-blocking check pointing schemes are preferable.

**17. What are the types of communication-induced check pointing?**

There are two types of communication-induced check pointing:

1. Model based check pointing and
2. Index-based check pointing.

**18. Define Consistent global state**

**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 ofthe message.

**19. Define blocking coordinated check pointing Blocking**

**Coordinated check pointing**

✓ A straightforward approach to coordinated check pointing is to block communications while the check pointing protocol executes.

✓ After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire check pointing activity is complete.

**20. Define Non-blocking checkpoint coordination Non- Blocking checkpoint**

**coordination**

✓ In this approach the processes need not stop their execution while taking checkpoints.

✓ A fundamental problem in coordinated check pointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

**21. What is model-based check pointing?**

Model-based check pointing,

- The system maintains checkpoints and communication structures that prevent the domino effect or achieve some even stronger properties.
- Model-based check pointing prevents patterns of communications and checkpoints that could result in inconsistent states among the existing checkpoints.

**22. What is index-based check pointing index- based check pointing?**

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

**23. What is the no-orphans consistency condition?**

**The no-orphans consistency condition**

Let e be a non-deterministic event that occurs at process p. We define then Following :

✓ **Depend(e):** the set of processes that are affected by a non- deterministic event e. This set consists of p, and any process whose state depends o the event e according to Lamport's happened before relation.

✓ **Log(e):** the set of processes that have logged a copy of e's determinant in their volatile memory.

✓ **Stable(e):** a predicate that is true if e's determinant is logged on the stable storage.

$$\forall(e) : \neg Stable(e) \Longrightarrow Depend(e) \subseteq Log(e).$$

This property is called the always-no-orphans condition.

**24. What is Koo–Toueg coordinated check pointing algorithm Koo–Toueg**

**Coordinated check pointing algorithm**

- Koo and Toueg's coordinated check pointing and recovery technique takes a consistent set of checkpoints and avoids the **domino effect and live lock problems** during the recovery.
- Processes coordinate their local check pointing actions such that the set of all checkpoints in the system is consistent.

**25. What is the reason of permanent checkpoint?**

A set of permanent checkpoints taken by this algorithm is consistent because of the following two reasons:

- ✓ first, either all or none ofthe processes take permanent checkpoints;
- ✓ second, no process sends a message after taking a tentative checkpoint until the receipt of the initiating process's decision, as by then all processes would have taken checkpoints.

**26. What is rollback recovery algorithm?**

**The rollback recovery algorithm**

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

**27. What are the two phases of rollback recovery algorithm?**

1. An initiating process $P_i$ sends a message to all other processes to check if they all are willing to restart fromtheir previous checkpoints.
2. The initiating process send the final decision to all processes, all the processes act accordingly after receiving the final decision.

**28. What is Byzantine Agreement (single source has an initial value) ? (Apr/May 2023)**

- **Agreement:** All non-faulty processes must agree on the same value.
- **Validity:** Ifthe 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-faultyprocess must eventually decide on a value.

**29. What is Consensus Problem (all processes have an initial value)? (or) What is Consensus in distributed system. (Nov/Dec 2022) (Apr/May 2023)**

- **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-faultyprocess must eventually decide on a value.

**30. What is Interactive Consistency (all processes have an initial value)?**

- **Agreement** All non-faulty processes must agree on the same array of values $A[v_1 \ldots v_n]$
- **Validity** If process i is non-faulty and its initial value is $v_i$, 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.

**31. List the benefits of recovery. (NOV/DEC 2021)**

- Protect the database
- Associated users from unnecessary problems
- Avoid or reduce the possibility of having to duplicate work manually.

**32. Why coordination is required in distributed systems? (NOV/DEC 2021)**

The coordination part of a distributed system **handles the communication and cooperation between processes**. It forms the glue that binds the activities performed by processes into a whole.

**33. What do you mean by local checkpoints? (Nov/Dec 2020/Apr/May 2021)**

✓ A local checkpoint is **a snapshot of the state** of the process at a given instance and the event of n recording the state of a process is called **local check pointing.**

**34. What is the drawback of a checkpoint based rollback recovery approach? (Nov/Dec 2020/Apr/May 2021)**

Disadvantages

**a) Multiple processes may force independent checkpoints for the same Z-**

**formation**.

b) Checkpoints may be forced for Z-formations that never emerge.

**35. State the use of Roll back recovery. (Nov/Dec 2022)**

End a unit of work and back out all the relational database changes that were made by that unit of work. If relational databases are the only recoverable resources used by the application process.

Back out only the changes made after a savepoint was set within the unit of work without ending the unit of work.

## UNIT V CLOUD COMPUTING

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

## PART – A

**1. Define Cloud Computing.**

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

**2. List the Characteristics of Cloud Computing.**

- ✓ On-demand self service
- ✓ Broad network access
- ✓ Resource pooling
- ✓ Rapid elasticity
- ✓ Measured service
- ✓ Performance
- ✓ Reduced costs
- ✓ Outsourced Management
- ✓ Reliability
- ✓ Multi-tenancy

**3. Express Cloud Deployment Model.**

(i) Public Cloud

- Available for public use or a large industry group

(ii) Private Cloud

- Operated for exclusive use of a single organization

(iii) Community Cloud

- Available for shared use of several organizations supporting a specificcommunity

(iv) Hybrid Cloud

- Combines multiple clouds (public and private) that remain unique but bound together to offer application and data portability

**4. State Cloud Service Model.**

(i) Software as a Service (SaaS)

Applications, management and user interfaces provided over a network

(ii) Platform as a Service (PaaS)

Application development frameworks, operating systems and deploymentframeworks

(iii) Infrastructure as a Service (IaaS)

Virtual computing, storage and network resource that can be provisioned ondemand

**5. List the examples for Cloud Services.**

- IaaS:
  - ✓ Amazon EC2
  - ✓ Google Compute Engine
  - ✓ Windows Azure VMs
- PaaS:
  - ✓ Google App Engine
- SaaS:
  - ✓ Salesforce

**6. Write the applications for Cloud Computing.**

- ✓ Banking & Financial Apps
- ✓ E-Commerce Apps
- ✓ Social Networking
- ✓ Healthcare Systems
- ✓ Energy Systems
- ✓ Intelligent Transportation Systems
- ✓ E-Governance
- ✓ Education
- ✓ Mobile Communications

**7. List the Driving Factors and Challenges of Cloud.**

- ✓ Virtualization
- ✓ Load balancing
- ✓ Scalability & Elasticity
- ✓ Deployment
- ✓ Replication
- ✓ Monitoring

**8. State Cloud Services and Platforms.**

- ✓ Compute Services
- ✓ Storage Services
- ✓ Database Services
- ✓ Application Services
- ✓ Content Delivery Services
- ✓ Analytics Services
- ✓ Deployment & Management Services
- ✓ Identity & Access Management Services

## PART – B

**1. Describe in detail about the Characteristics of Cloud Computing.**

**Characteristics of Cloud Computing:**

i. On-demand self service
ii. Broad network access
iii. Resource pooling
iv. Rapid elasticity
v. Measured service
vi. Performance
vii. Reduced costs
viii. Outsourced Management
ix. Reliability
x. Multi-tenancy

**i. On-demand self service:**

Cloud computing resources can be provisioned on-demand by the users, without requiring interactions with the cloud service provider.

**ii. Broad network access:**

Cloud computing resources can be accessed over the network using standard access mechanisms that provide platform-independent access through the use of heterogeneous client platforms such as workstations, laptops, tablets and smartphones.

**iii. Resource pooling:**

The computing and storage resources provided by cloud service providers are pooled to serve multiple users using multi-tenancy. Multi-tenant aspects of the cloud allow multiple users to be served by the same physical hardware.

**iv. Rapid elasticity:**

Cloud computing resources can be provisioned rapidly and elastically. Cloud resources can be rapidly scaled up or down based on demand.

**v. Measured service:**

Cloud computing resources are provided to users on a pay-per-use model. The usage of the cloud resources is measured and the user is charged basedon some specific metric.

**vi. Performance:**

Cloud computing provides improved performance for applications since the resources available to the applications can be scaled up or down based on thedynamic application workloads.

**vii. Reduced costs:**

Cloud computing provides cost benefits for applications as only as much computing and storage resources as required can be provisioned dynamically, and upfront investment in purchase of computing assets to cover worst case requirements is avoid.

**viii. Outsourced Management**

Cloud computing allows the users (individuals, large organizations, small andmedium enterprises and governments) to outsource the IT infrastructure requirements to external cloud providers.

**ix. Reliability**

Applications deployed in cloud computing environments generally have ahigher reliability since the underlying IT infrastructure is professionally managed by the cloud service.

**x. Multi-tenancy:**

- ✓ The multi-tenanted approach of the cloud allows multiple users to make use of the same shared resources.
- ✓ In virtual multi-tenancy, computing and storage resources are shared amongmultiple users.
- ✓ In organic multi-tenancy every component in the system architecture is sharedamong multiple tenants

## 2. Explain in detail about Cloud Service Models.

(i) Software as a Service (SaaS)

Applications, management and user interfaces provided over a network

(ii) Platform as a Service (PaaS)

Application development frameworks, operating systems and deploymentframeworks

(iii) Infrastructure as a Service (IaaS)

Virtual computing, storage and network resource that can be provisioned ondemand

### (i) Software as a Service (SaaS)

**Software/Interface**

- SaaS provides the users a complete software application or the user interface to the application itself.

**Outsourced Management**

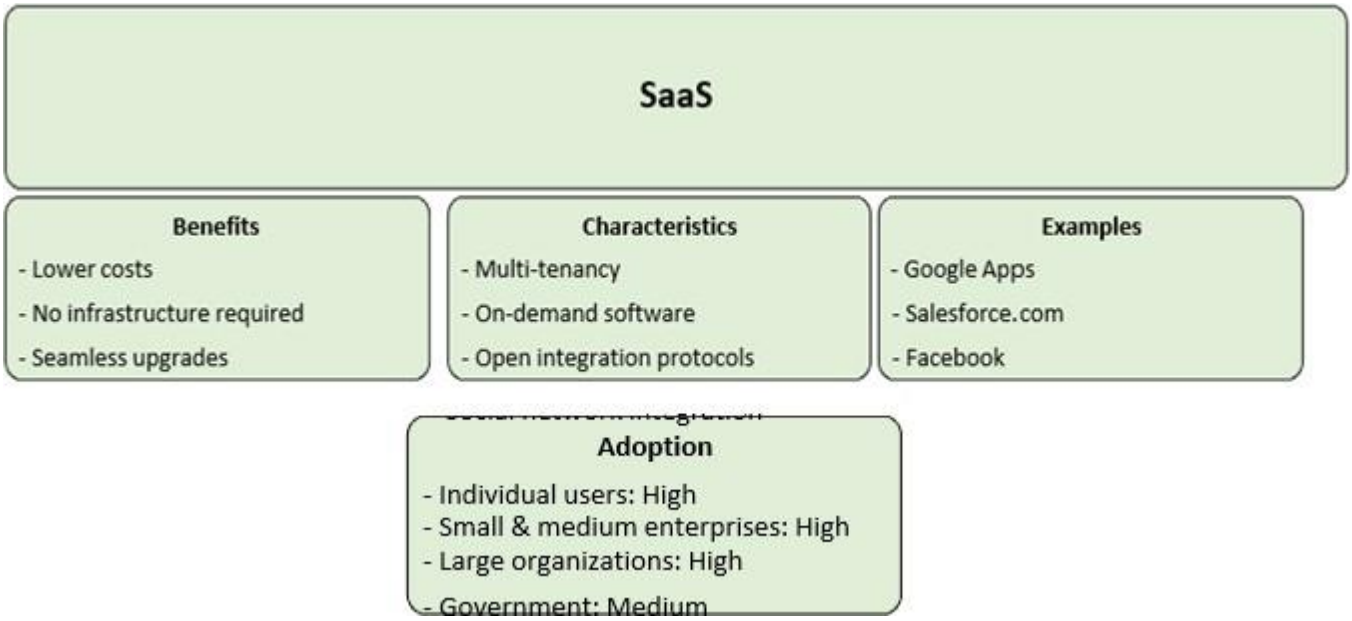
- The cloud service provider manages the underlying cloud infrastructure including servers, network, operating systems, storage and application software, and the user is unaware of the underlying architecture of the cloud.

**Thin client interfaces**

- Applications are provided to the user through a thin client interface (e.g., a browser). SaaS applications are platform independent and can be accessed from various client devices such as workstations, laptop, tablets and smartphones, running different operating systems.

**Ubiquitous Access**

- Since the cloud service provider manages both the application and data, the users are able toaccess the applications from anywhere.




**(ii) Platform as a Service (PaaS)**

**Development & Deployment**

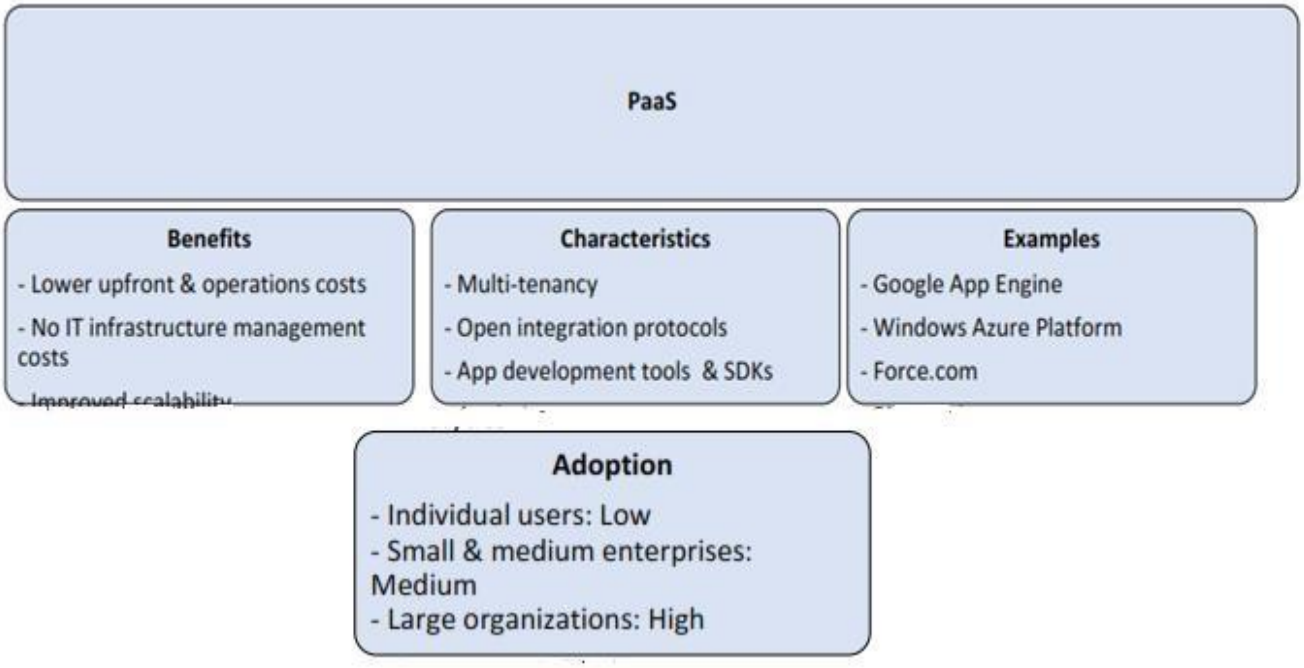
- PaaS provides the users the capability to develop and deploy application in the cloud using the development tools, application programming interfaces (APIs), software libraries and services provided by the cloud service provider.

**Provider Manages Infrastructure**

- The cloud service provider manages the underlying cloud infrastructureincluding servers, network, operating systems and storage.

**User Manages Application**

- The users, themselves, are responsible for developing, deploying, configuringand managing applications on the cloud infrastructure.




## (iii) Infrastructure as a Service (IaaS)

**Resource Provisioning**

- Provides the users the capability to provision computing and storage resources.

**Virtual Machines**

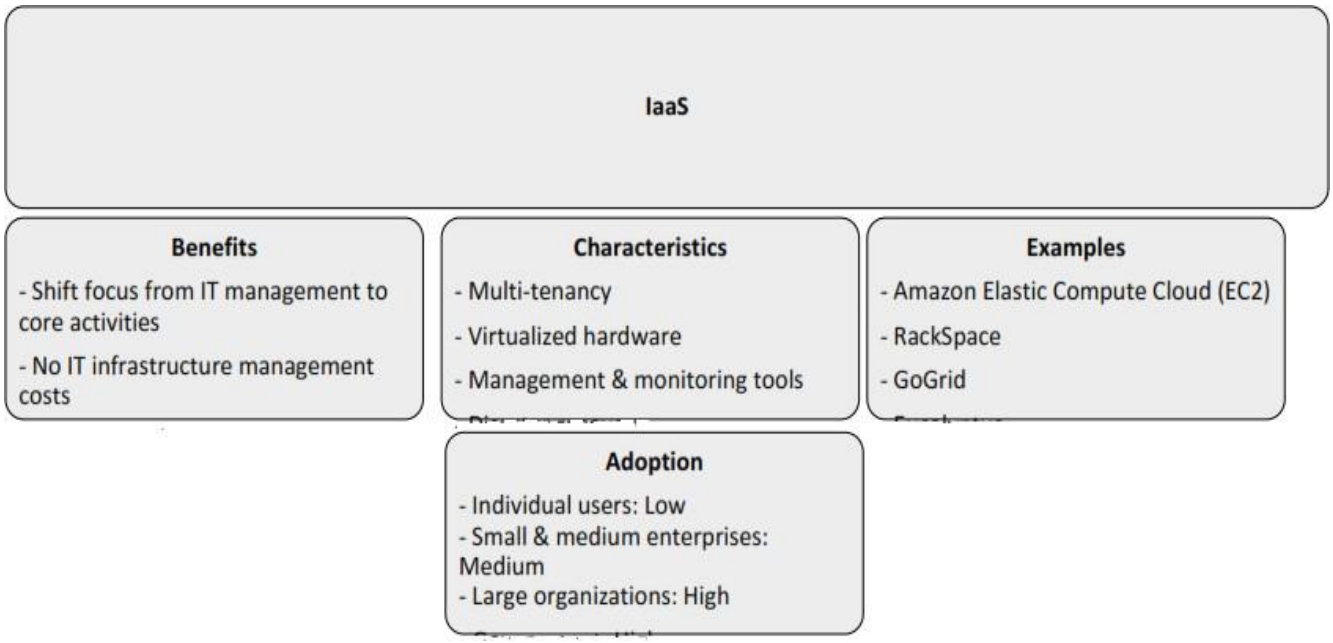
- These resources are provided to the users as virtual machine instances and virtual storage. Users can start, stop, configure and manage the virtual machine instancesand virtual storage.

**Provider Managers Infrastructure:**

- The cloud service provider manages the underlying infrastructure.

**Pay-per-use/Pay-as-you-go:**

- Virtual resources provisioned by the users are billed based on a pay-per-use/pay-as-you-go paradigm.

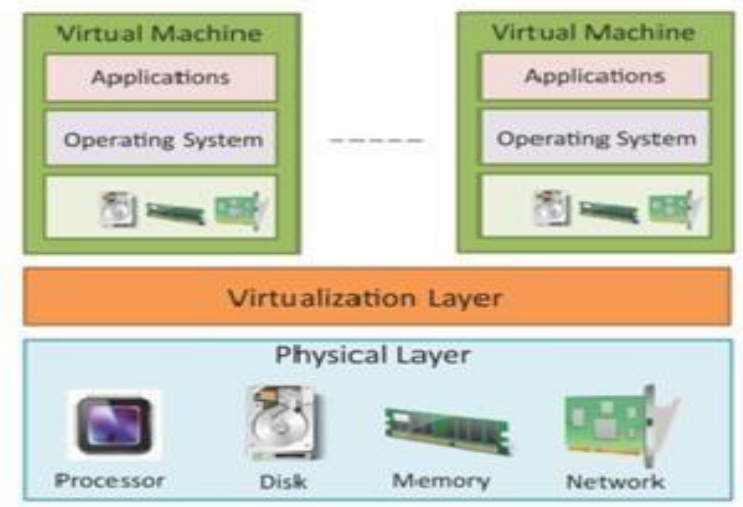



**3. List the Driving Factors and Challenges of Cloud and explain in detail about Virtualization.**

a. Virtualization
b. Load balancing
c. Scalability & Elasticity
d. Deployment
e. Replication
f. Monitoring

**a. Virtualization**

- Virtualization refers to the partitioning theresources of a physical system (such as computing, storage, network and memory)into multiple virtual resources.




- Key enabling technology of cloud computing that allow pooling of resources.
- In cloud computing, resources are pooled to serve multiple users using multi-tenancy.

**Hypervisor**

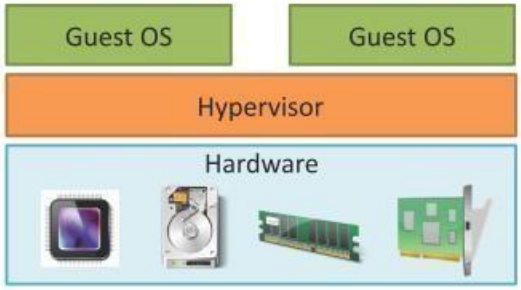
- The virtualization layer consists of a hypervisor or avirtual machine monitor (VMM).
- Hypervisor presents a virtual operating platform to a guest operating system.

✓ Type-1 Hypervisor

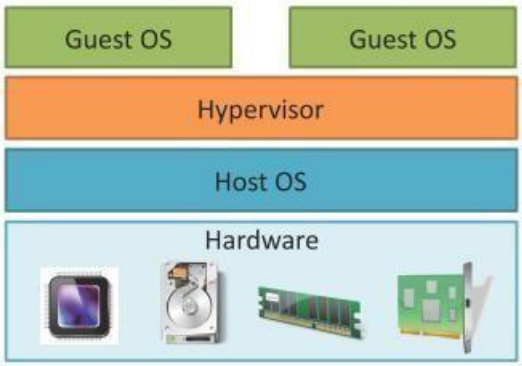
✓ Type-I or the native hypervisors run directly on the host hardware and control the hardware and monitor the guestoperating systems.

✓ Type-2 Hypervisor

✓ Type 2 hypervisors or hosted hypervisors run on top of a conventional (main/host) operating system and monitorthe guest operating systems.




**Type 1 Hypervisor**




**Type 2 Hypervisor**

## Types of the Hypervisor

**(i) Full Virtualization**

In full virtualization, the virtualization layer completely decouples the guest OS from the underlying hardware. The guest OS requires no modification and is not aware that it is being virtualized. Full virtualization is enabled by direct execution of user requests and binary translation of OS requests.

**(ii) Para-Virtualization**

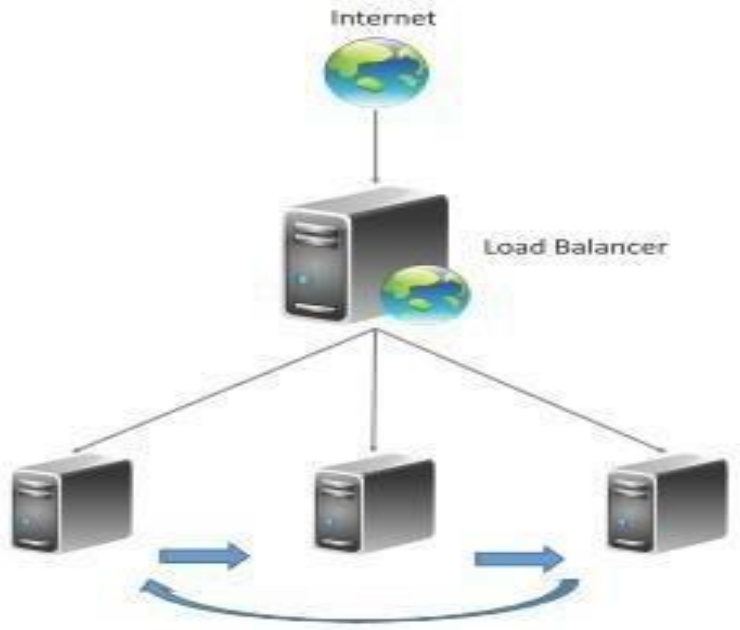
In para-virtualization, the guest OS is modified to enable communication with the hypervisor to improve performance and efficiency. The guest OS kernel is modified to replace non-virtualizableinstructions with hyper-calls that communicate directly with the virtualization layer hypervisor.

**(iii) Hardware Virtualization**

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

## 4. Describe in detail about Load Balancing.

- Cloud computing resources can be scaled up on demand to meet the performance requirements of applications.
- Load balancing distributes workloads across multipleservers to meet the application workloads.




- The goals of load balancing techniques include:
  - ✓ Achieve maximum utilization of resources
  - ✓ Minimizing the response times
  - ✓ Maximizing throughput

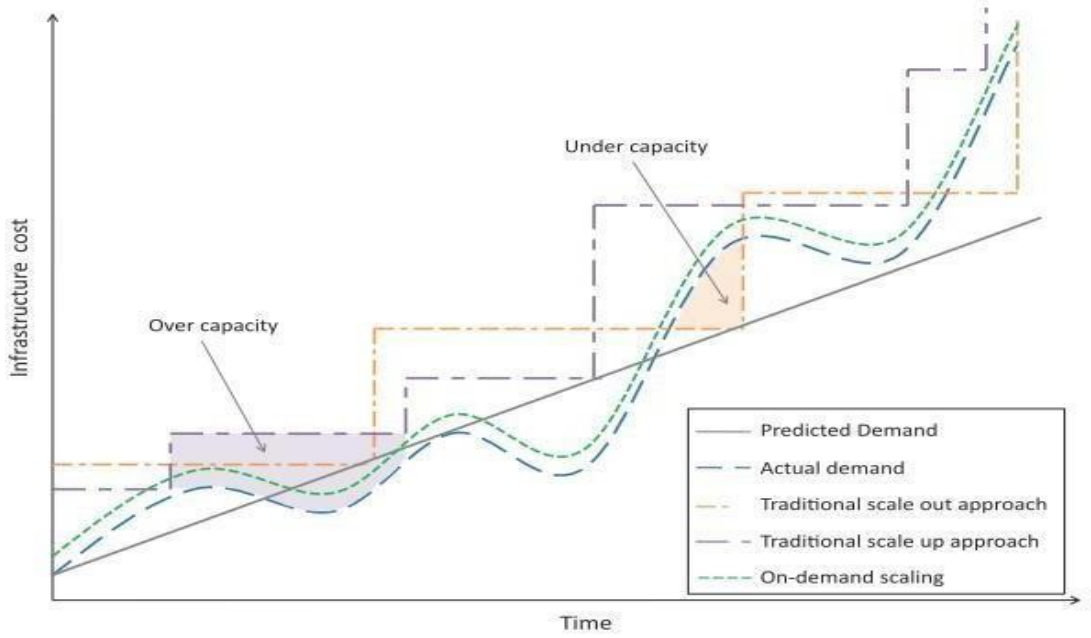
- **Load Balancing Algorithms**
  - ✓ Round Robin load balancing
  - ✓ Weighted Round Robin load balancing
  - ✓ Low Latency load balancing
  - ✓ Least Connections load balancing
  - ✓ Priority load balancing
  - ✓ Overflow load balancing

**Load Balancing – Persistent Approaches**

- Since load balancing can route successive requests from a user session to different servers, maintaining the state or the informationof the session is important.
- Persistence Approaches
  - ✓ Sticky sessions
  - ✓ Session Database
  - ✓ Browser cookies
  - ✓ URL re-writing

## 5. Explain in detail about Scalability and Elasticity.

- Multi-tier applications such as e-Commerce, social networking, business-to-business, etc. can experience rapid changes in their traffic.
- Capacity planning involves determining the right sizing of each tier of the deployment ofan application in terms of the number of resources and the capacity of each resource.
- Capacity planning may be for computing,storage, memory or network resources.




**Scaling Approaches**

- Vertical Scaling/Scaling up:
  - ✓ Involves upgrading the hardware resources (adding additional computing, memory,storage or network resources).

- Horizontal Scaling/Scaling out
  - ✓ Involves addition of more resources of the same type.

**6. Express the concept of Deployment and Replication.**

Cloud application deployment design is an iterative process that involves:

❖ **Deployment Design**

- ✓ The variables in this step include the number of servers in each tier, computing, memory and storage capacities of severs, server interconnection, load balancing and replication strategies.

❖ **Performance Evaluation**

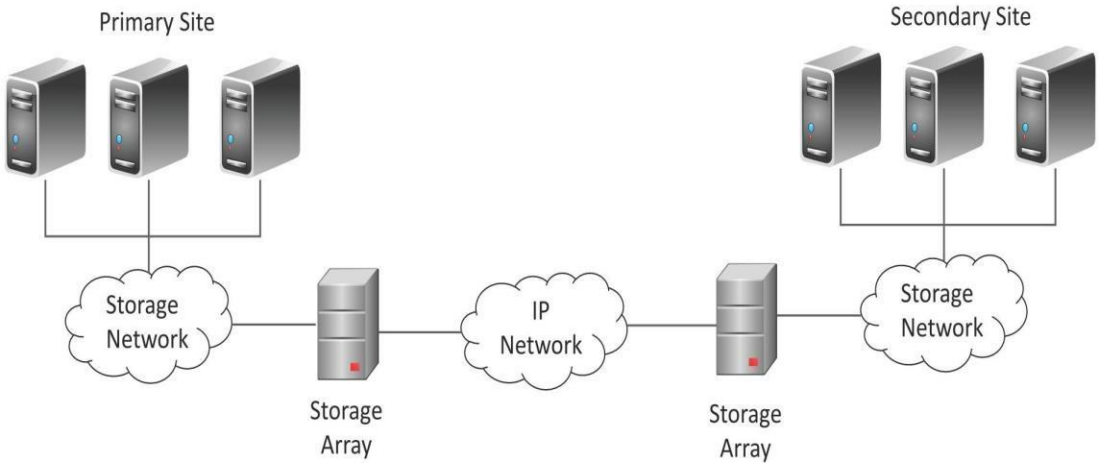
- ✓ To verify whether the application meets the performance requirements with the deployment.
- ✓ Involves monitoring the workload on the application and measuring various workload parameters such asresponse time and throughput.
- ✓ Utilization of servers (CPU, memory, disk, I/O, etc.) in each tier is also monitored.

❖ **Deployment Refinement**

- ✓ Various alternatives can exist in this step such as vertical scaling (or scaling up), horizontal scaling (or scalingout), alternative server interconnections, alternative load balancing and replication strategies, for instance.

**Replication:**

- Replication is used to create and maintain multiple copies of the data in the cloud.
- Cloud enables rapid implementation of replication solutions for disaster recovery fororganizations.
- With cloud-based data replication organizations can plan for disaster recovery without making any capital expenditures on purchasing, configuring and managing secondary sitelocations.




**Example: Array based Replication**

- Types of Replications:
  - ✓ Array-based Replication
  - ✓ Network-based Replication
  - ✓ Host-based Replication

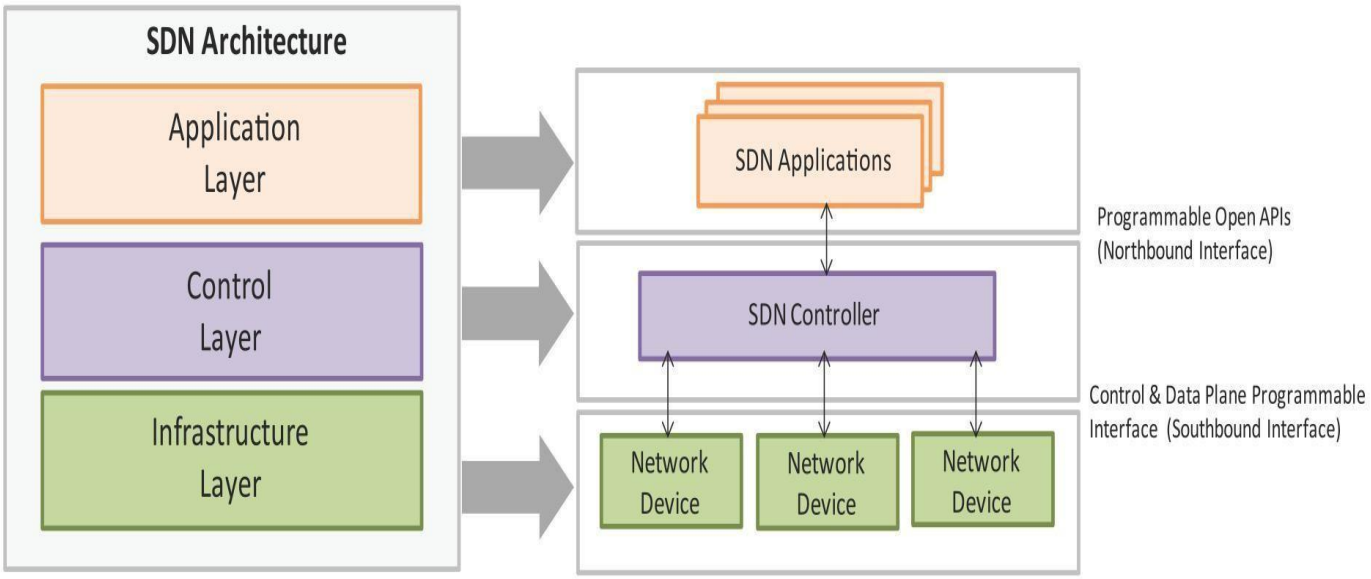
**7. Express the concept of Monitoring and explain about Software defined Networking.**

- Monitoring services allow cloud users tocollect and analyze the data on various monitoring metrics
- A monitoring service collects data on various system and application metrics from the cloud computing instances.
- Monitoring of cloud resources is important because it allows the users to keep track of the health of applications and services deployed in the cloud.

| Type | Metrics |
|---|---|
| CPU | CPU-Usage, CPU-Idle |
| Disk | Disk-Usage, Bytes/sec (read/write), Operations/sec |
| Memory | Memory-Used, Memory-Free, Page-Cache |
| Interface | Packets/sec (incoming/outgoing), Octets/sec(incoming/outgoing) |

**Software defined Networking**

- Software-Defined Networking (SDN) is a networking architecture that separates thecontrol plane from the data plane and centralizes the network controller.
- Conventional network architecture
  - ✓ The control plane and data plane are coupled. Control plane is the part of the network that carries thesignaling and routing message traffic while the data plane is the part of the network that carries the payload data traffic.
- SDN Architecture
  - ✓ The control and data planes are decoupled and the network controller is centralized.




**SDN Architecture**

**SDN – Key Elements**

- **Centralized Network Controller**
  - ✓ With decoupled the control and data planes and centralized network controller, the network administrators can rapidly configure the network.

- **Programmable Open APIs**
  - ✓ SDN architecture supports programmable open APIs for interface between the SDN application and control layers (Northbound interface). These open APIs that allow implementing various network servicessuch as routing, quality of service (QoS), access control, etc.

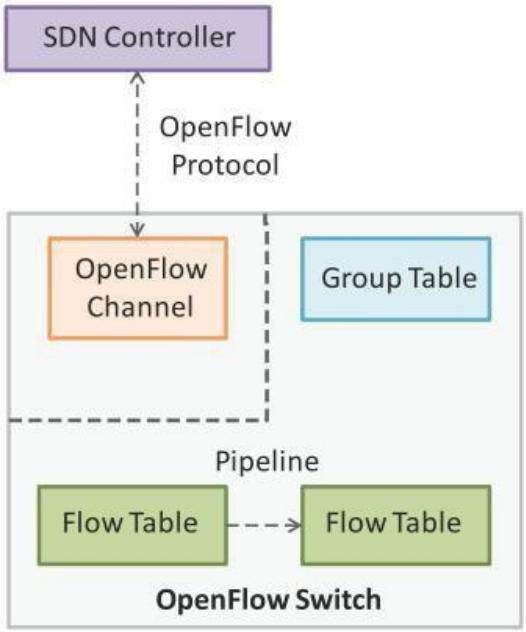
- **Standard Communication Interface (OpenFlow)**
  - ✓ SDN architecture uses a standard communication interface between the control and infrastructure layers (Southbound interface). OpenFlow, which is defined by the Open Networking Foundation (ONF) is the broadly accepted SDN protocol for the Southbound interface.

**About Openflow:**

OpenFlow is the broadly accepted SDN protocol for the Southbound interface

- ✓ With OpenFlow, the forwarding plane of the network devices can be directly accessed and manipulated.
- ✓ OpenFlow uses the concept of flows to identify network traffic basedon pre-defined match rules.

- ✓ Flows can be programmed statically or dynamically by the SDNcontrol software.
- ✓ OpenFlow protocol is implemented on both sides of the interface between the controller and the network devices.



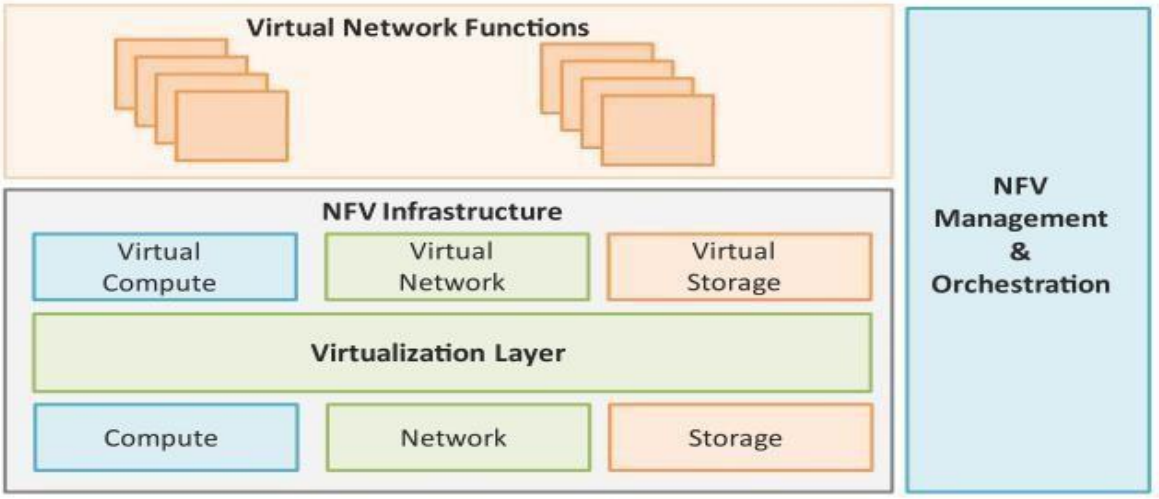

## Openflow

OpenFlow switch comprising of one or more flow tables and a group table, which perform packetlookups and forwarding, and OpenFlow channel to an external controller.

### Network Function Virtualization

- Network Function Virtualization (NFV) is a technology that leverages virtualization to consolidate the heterogeneous network devices onto industry standard high volume servers, switches and storage.
- Relationship to SDN
  - ✓ NFV is complementary to SDN as NFV can provide the infrastructure on which SDN can run.
  - ✓ NFV and SDN are mutually beneficial to each other but not dependent.
  - ✓ Network functions can be virtualized without SDN, similarly, SDN can run without NFV.
- NFV comprises of network functions implemented in software that run on virtualized resources in the cloud.
- NFV enables a separation the network functions which are implemented in software from the underlying hardware.

**NFV Architecture**



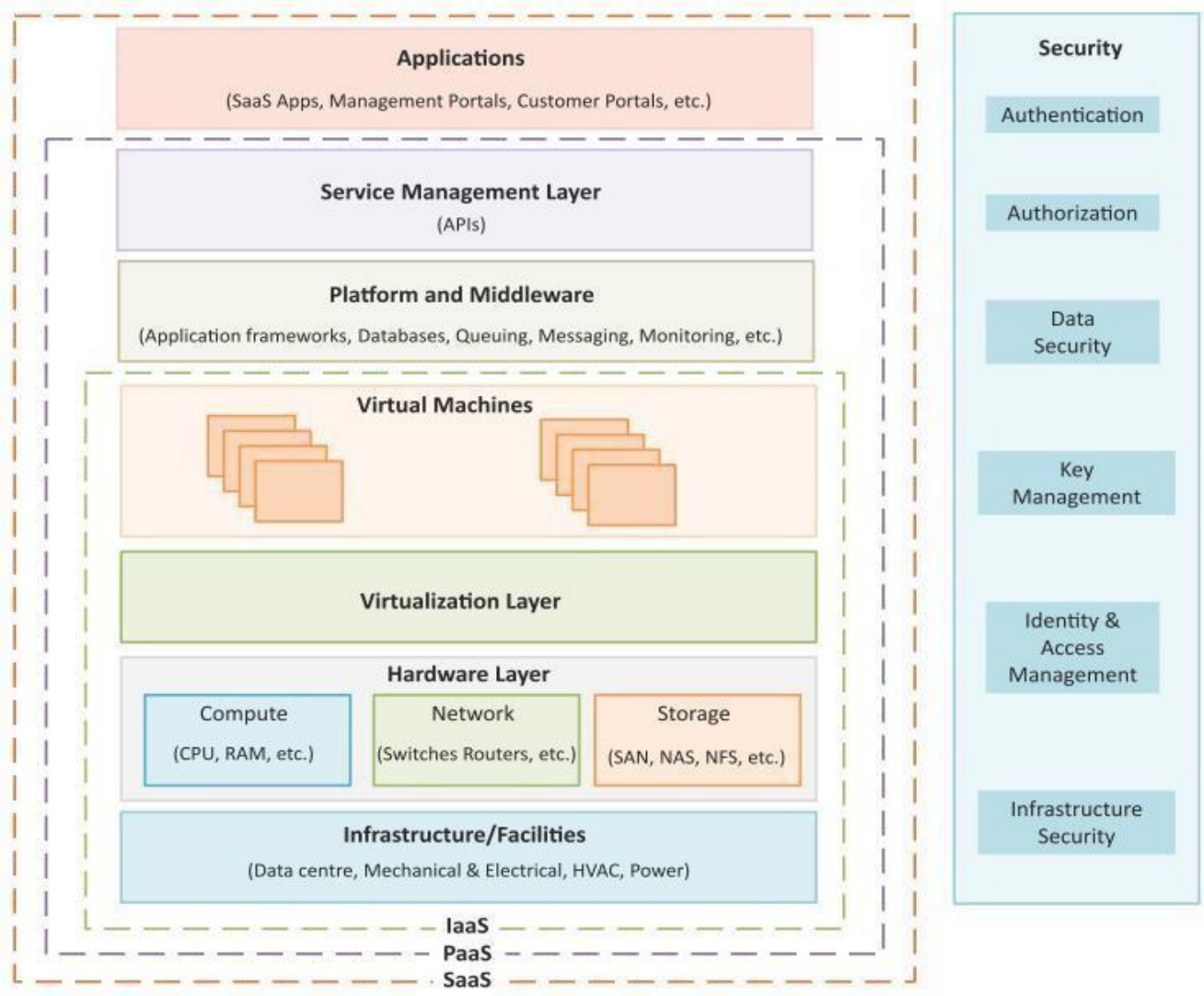

**Architecture of NFV**

- Key elements of the NFV architecture are
  - ✓ Virtualized Network Function (VNF): VNF is a software implementation of a network function which is capable of running over the NFV Infrastructure (NFVI).
  - ✓ NFV Infrastructure (NFVI): NFVI includes compute, network and storage resources that are virtualized.
  - ✓ NFV Management and Orchestration: NFV Management and Orchestration focuses on all virtualization-specific management tasks and coversthe orchestration and lifecycle management of physical and/or software resources that support theinfrastructure virtualization, and the lifecycle management of VNFs.

**8. Express the concept of Cloud reference Model.**

- **Infrastructure & Facilities Layer**
  - ✓ Includes the physical infrastructure such as datacenter facilities, electrical and mechanical equipment, etc.
- **Hardware Layer**
  - ✓ Includes physical compute, network and storage hardware.
- **Virtualization Layer**
  - ✓ Partitions the physical hardware resources into multiple virtual resources that enabling pooling of resources.
- **Platform & Middleware Layer**
  - ✓ Builds upon the IaaS layers below and provides standardized stacks of services such as database

service, queuing service, application frameworks and run-time environments, messaging services, monitoring services, analytics services, etc.




- **Service Management Layer**
  - ✓ Provides APIs for requesting, managing and monitoring cloud resources.
- **Applications Layer**
  - ✓ Includes SaaS applications such as Email, cloud storage application, productivity applications, management portals, customer self-serviceportals, etc.

**9. Explain in detail about Compute Services with examples.**

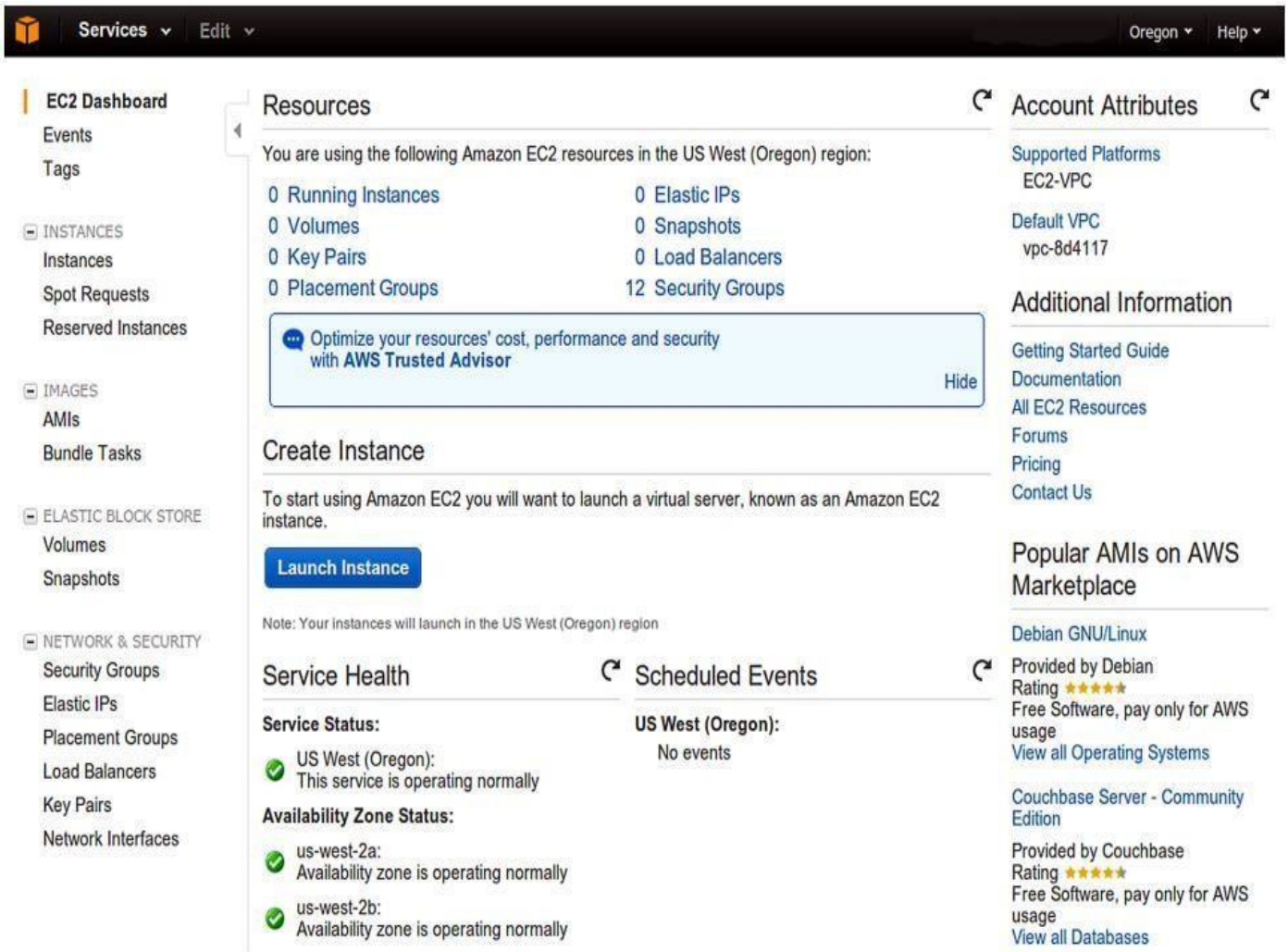
- ✓ Compute services provide dynamically scalable compute capacity in the cloud.
- ✓ Compute resources can be provisioned on-demand in the form of virtual machines. Virtual machines can be created from standard images provided by the cloud service provider or custom images created by the users.

- ✓ Compute services can be accessed from the web consoles of these services that provide graphical user interfaces for provisioning, managing and monitoring theseservices.

- ✓ Cloud service providers also provide APIs for various programming languages thatallow developers to access and manage these services programmatically.

## Compute Services – Amazon EC2

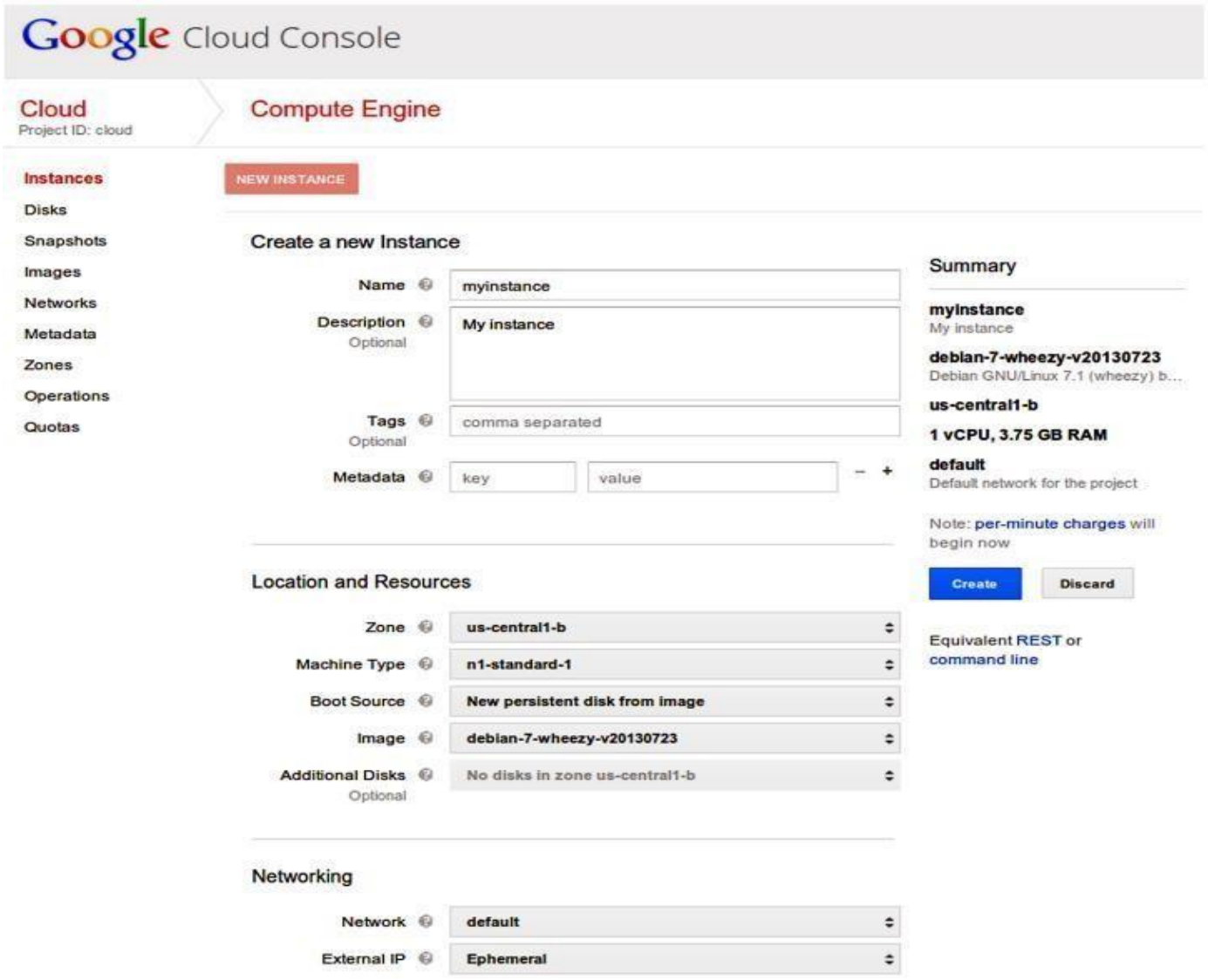
- Amazon Elastic Compute Cloud (EC2) is a compute service provided by Amazon.
- **Launching EC2 Instances**
  - ✓ To launch a new instance click on the launch instance button. This will open a wizard where you can select the Amazon machine image (AMI) with which you want to launchthe instance. You can also create their own AMIs with custom applications, libraries and data. Instances can be launched with a variety of operating systems.
- **Instance Sizes**
  - ✓ When you launch an instance you specify the instance type (micro, small, medium, large, extra-large, etc.), the number of instances to launch based on the selected AMIand availability zones for the instances.



- **Key-pairs**
  - ✓ When launching a new instance, the user selects a key-pair from existing keypairs orcreates a new keypair for the instance. Keypairs are used to securely connect to an instance after it launches.
- **Security Groups**
  - ✓ The security groups to be associated with the instance can be selected from the instance launch wizard. Security groups are used to open or block a specific networkport for the launched instances.

## Compute Services – Google Compute Engine

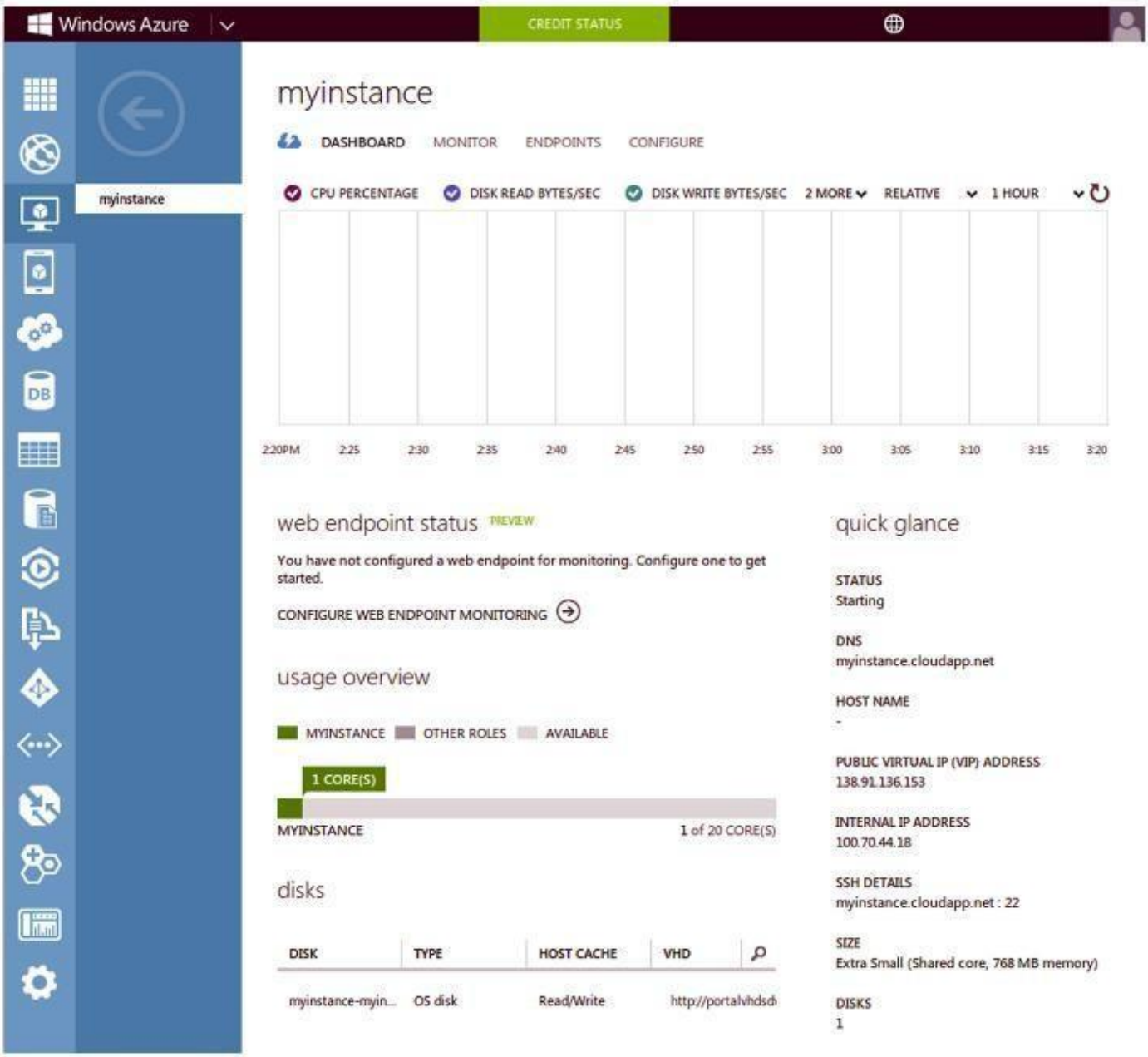
- Google Compute Engine is a compute service provided byGoogle.
- **Launching Instances**
  - ✓ To create a new instance, the user selects an instance machine type, a zone in which the instance will be launched, a machine image for the instance and provides an instance name, instance tags and meta-data.



- **Disk Resources**
  - ✓ Every instance is launched with a disk resource. Depending on the instance type, the disk resource can be a scratch disk space or persistent disk space. The scratch disk space is deleted when the instance terminates. Whereas, persistent disks live beyond the life of an instance.
- **Network Options**
  - ✓ Network option allows you to control the traffic to and from the instances. By default, traffic between instances in the same network, over any port and any protocol and incoming SSH connections from anywhere are enabled.

## Windows Azure VMs

Windows Azure Virtual Machines is the compute service from Microsoft



- Launching Instances:
  - ✓ To create a new instance, you select the instance type and the machine image.
  - ✓ You can either provide a user name and password or upload a certificate file for securely connecting to the instance.
  - ✓ Any changes made to the VM are persistently stored and new VMs can be created from the previously stored machine images.

**10. Explain in detail about Storage Services with examples.**

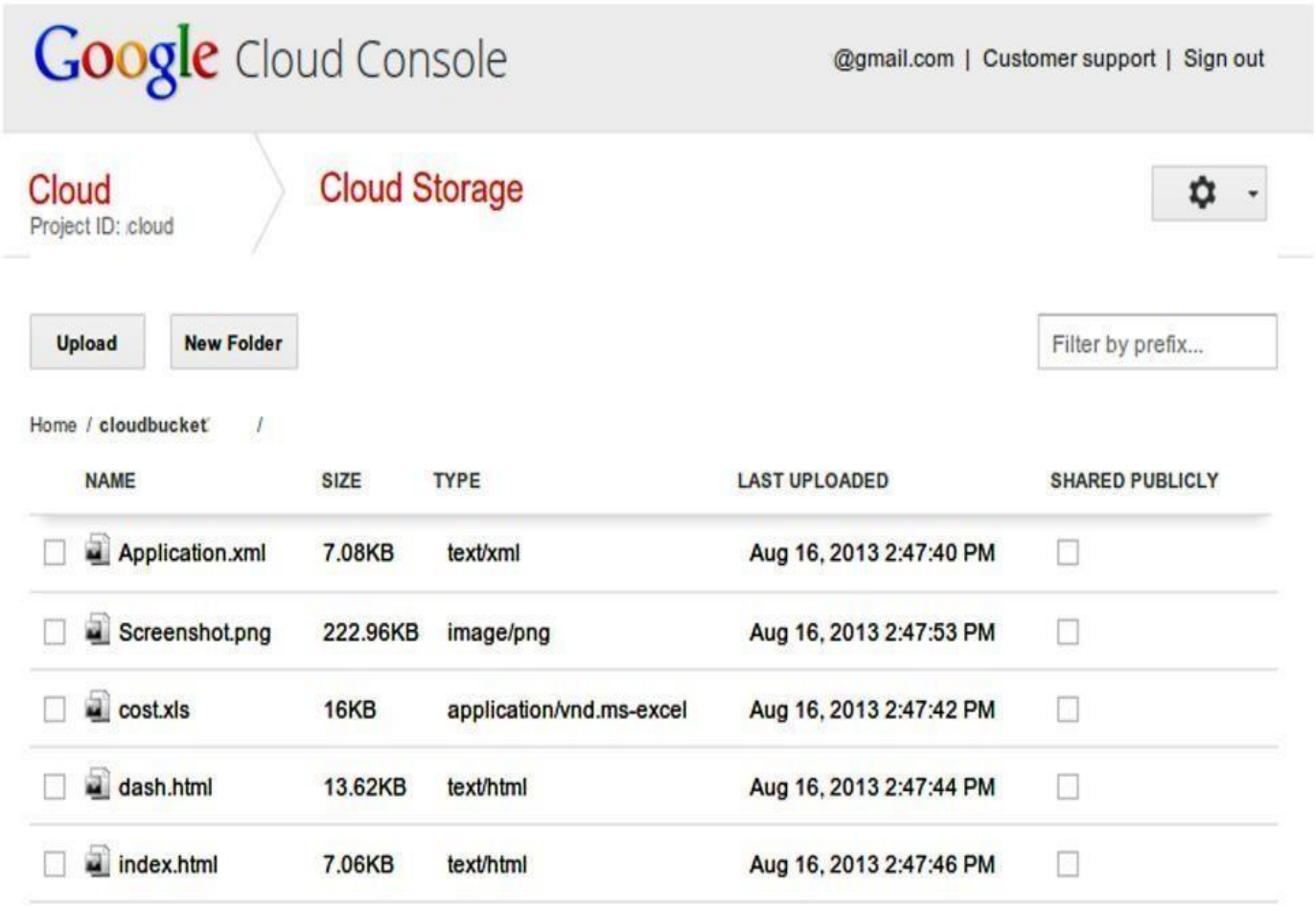
- Cloud storage services allow storage and retrieval of any amount of data, at any time from anywhere on theweb.
- Most cloud storage services organize data into buckets or containers.
- Scalability
  - ✓ Cloud storage services provide high capacity and scalability. Objects upto several tera-bytes in size can be uploaded andmultiple buckets/containers can be created on cloud storages.
- Replication
  - ✓ When an object is uploaded it is replicated at multiple facilities and/or on multiple devices within each facility.
- Access Policies
  - ✓ Cloud storage services provide several security features such as Access Control Lists (ACLs), bucket/container level policies, etc. ACLs can be used to selectively grant access permissions on individual objects. Bucket/container level policies can also be defined to allow or deny permissions across some or all of the objects within a single bucket/container.
- Encryption
  - ✓ Cloud storage services provide Server Side Encryption (SSE) options to encrypt all data stored in the cloud storage.
- Consistency
  - ✓ Strong data consistency is provided for all upload and delete operations. Therefore, any object that is uploaded can beimmediately downloaded after the upload is complete.

**Storage Services – Amazon S3**

- Amazon Simple Storage Service(S3) is an online cloud-based data storage infrastructure for storing and retrieving any amount of data.
- S3 provides highly reliable, scalable, fast, fully redundant andaffordable storage infrastructure.
- Buckets
  - ✓ Data stored on S3 is organized in the form of buckets. You must create a bucket before you can store data on S3.
- Uploading Files to Buckets
  - ✓ S3 console provides simple wizards for creating a new bucket anduploading files.
  - ✓ You can upload any kind of file to S3.
  - ✓ While uploading a file, you can specify the redundancy and encryption options and access permissions.

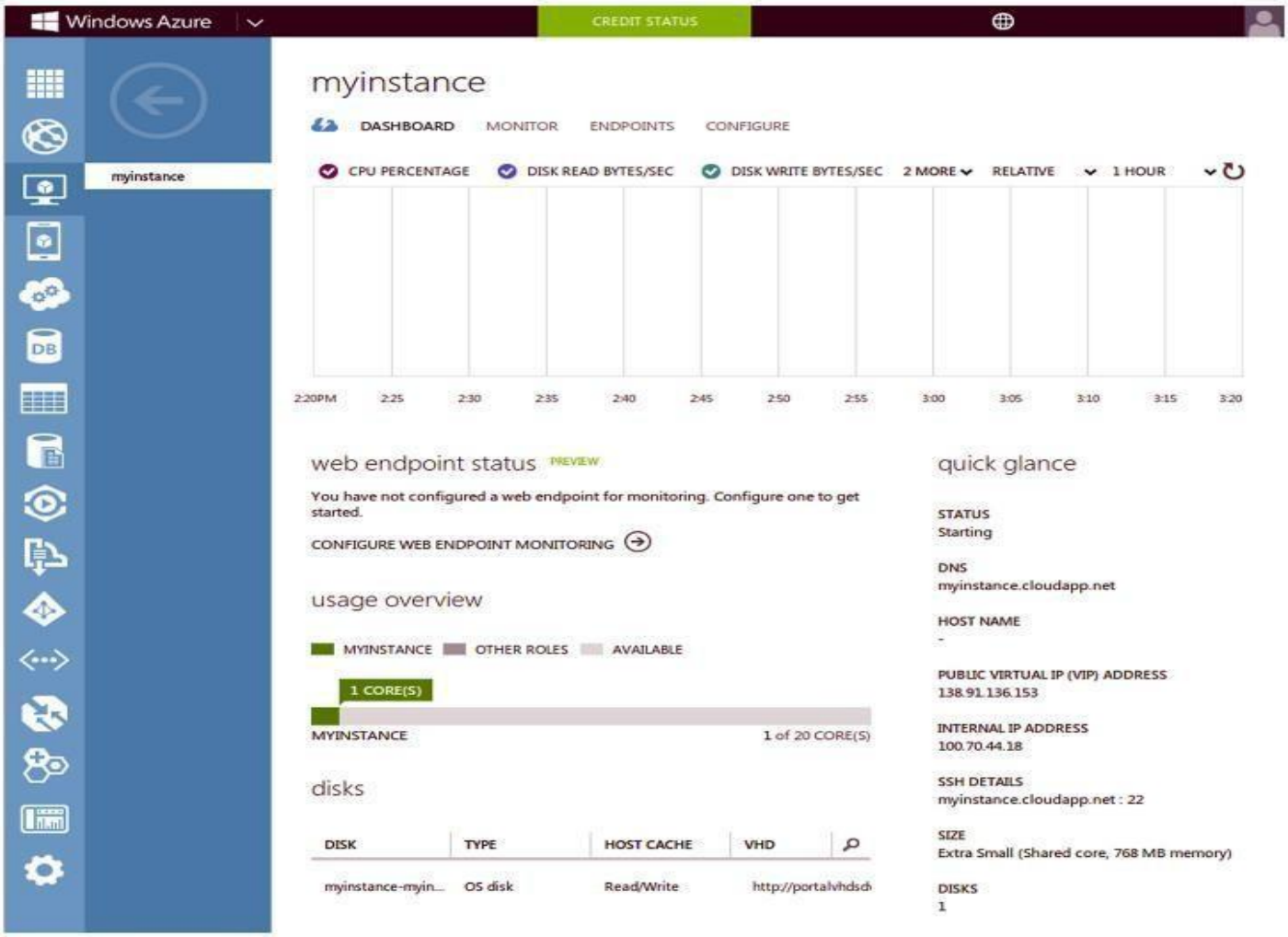

**Storage Services - Google Cloud Storage**

- GCS is the Cloud storage service from Google
- Buckets
  - ✓ Objects in GCS are organized into buckets.
- Access Control Lists
  - ✓ ACLs are used to control access to objects and buckets. ACLs can be configured to share objects and buckets with the entire world, a Google group, a Google-hosted domain, or specific Google account holders.



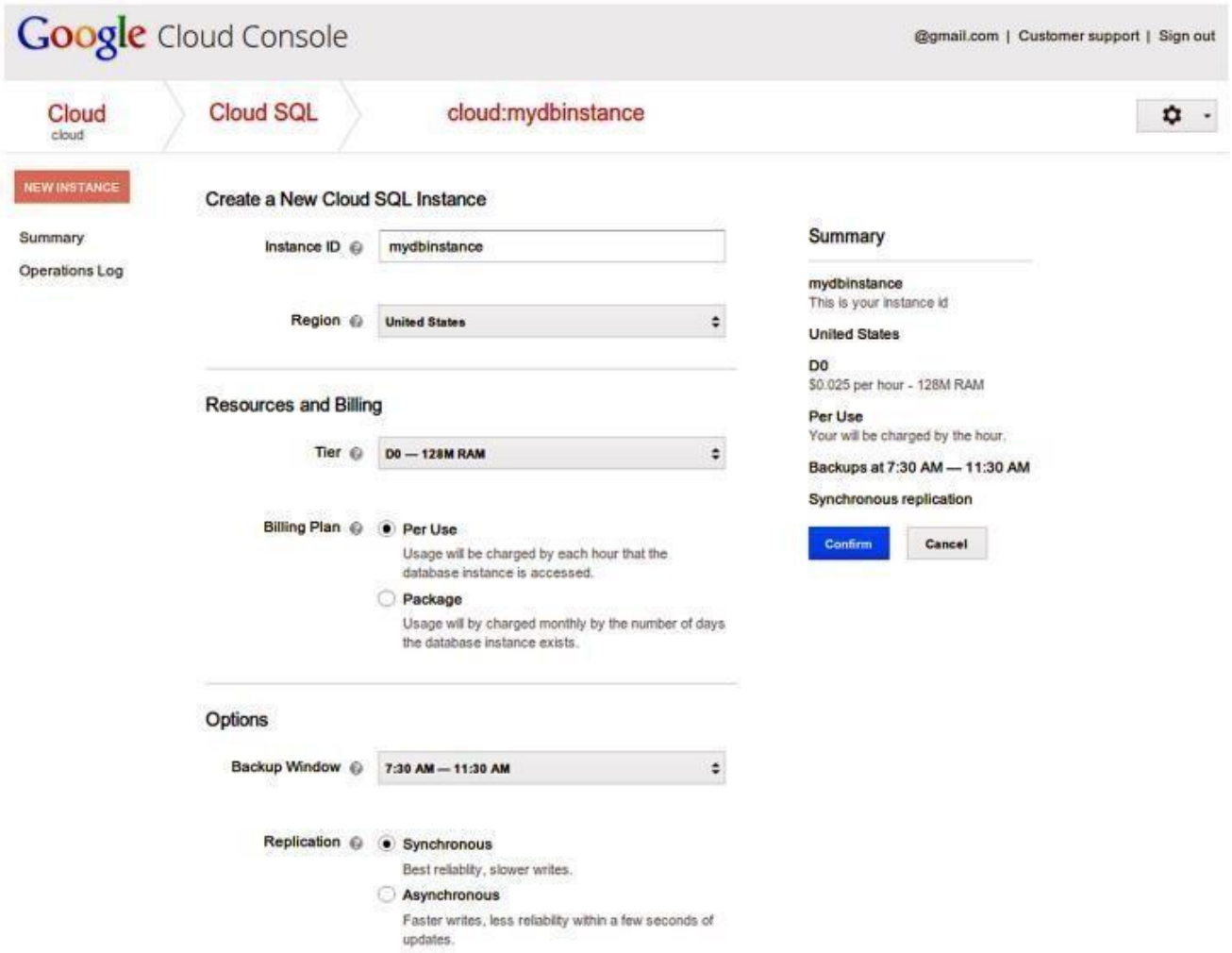
## Storage Services - Windows Azure Storage

- Windows Azure Storage is the cloud storage service fromMicrosoft.
- Windows Azure Storage provides various storage services such as blob storage service, table service and queue service.
- Blob storage service
  - ✓ The blob storage service allows storing unstructured binary data or binarylarge objects (blobs).
  - ✓ Blobs are organized into containers.
  - ✓ Block blobs - can be subdivided into some number of blocks. If a failure occurs while transferring a block blob, retransmission can resume with the most recent block rather than sending the entire blob again.
  - ✓ Page blobs - are divided into number of pages and are designed for random access. Applications can read and write individual pages at random in a page blob.



## Storage Service – Google Cloud SQL

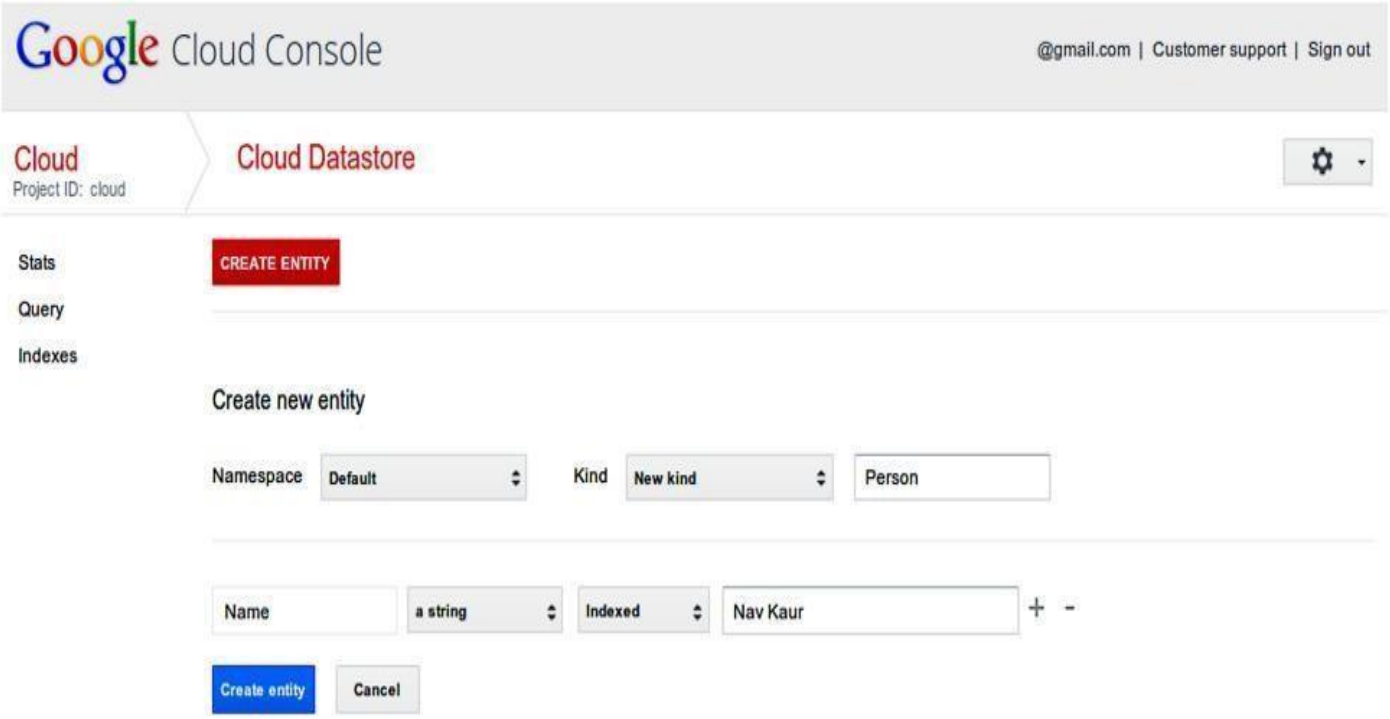
- Google SQL is the relational database service from Google.
- Google Cloud SQL service allows you to host MySQL databasesin the Google's cloud.
- Launching DB Instances
  - ✓ You can create new database instances from the console and manage existing instances. To create a new instance you select a region, databasetier, billing plan and replication mode.
- Backups
  - ✓ You can schedule daily backups for your Google Cloud SQL instances, andalso restore backed-up databases.
- Replication
  - ✓ Cloud SQL provides both synchronous or asynchronous geographic replication and the ability to import/ export databases.




## Storage Service – Google Cloud Datastore

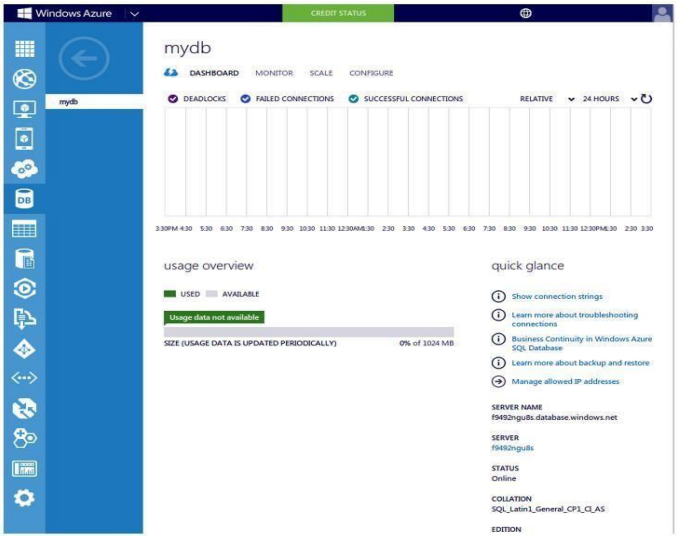
- Google Cloud Datastore is a fully managed non-relationaldatabase from Google.
- Cloud Datastore offers ACID transactions and high availabilityof reads and writes.
- Data Model

- ✓ The Cloud Datastore data model consists of entities. Each entity has one or more properties (key-value pairs) which can be of one of several supported data types, such as strings and integers. Each entity has a kind and a key. The entity kind is used for categorizing the entity for the purpose of queries and the entity key uniquely identifies the entity.




**Storage Service – Windows Azure SQL DB**

- Windows Azure SQL Database is the relational databaseservice from Microsoft.
- Azure SQL Database is based on the SQL server, but it does notgive each customer a separate instance of SQL server.
- Multi-tenant Service
  - ✓ SQL Database is a multi-tenant service, with a logical SQL Database serverfor each customer.



### Storage Service – Windows Azure Table Service

- Windows Azure Table Service is a non-relational (No-SQL) database service from Microsoft.
- Data Model
  - ✓ The Azure Table Service data model consists of tables having multiple entities.
  - ✓ Tables are divided into some number of partitions, each of which can be stored on a separate machine.
  - ✓ Each partition in a table holds a specified number of entities, each containing as many as 255 properties.
  - ✓ Each property can be one of the several supported data types such as integers and strings.
- No Fixed Schema
  - ✓ Tables do not have a fixed schema and different entities in a table can have different properties.

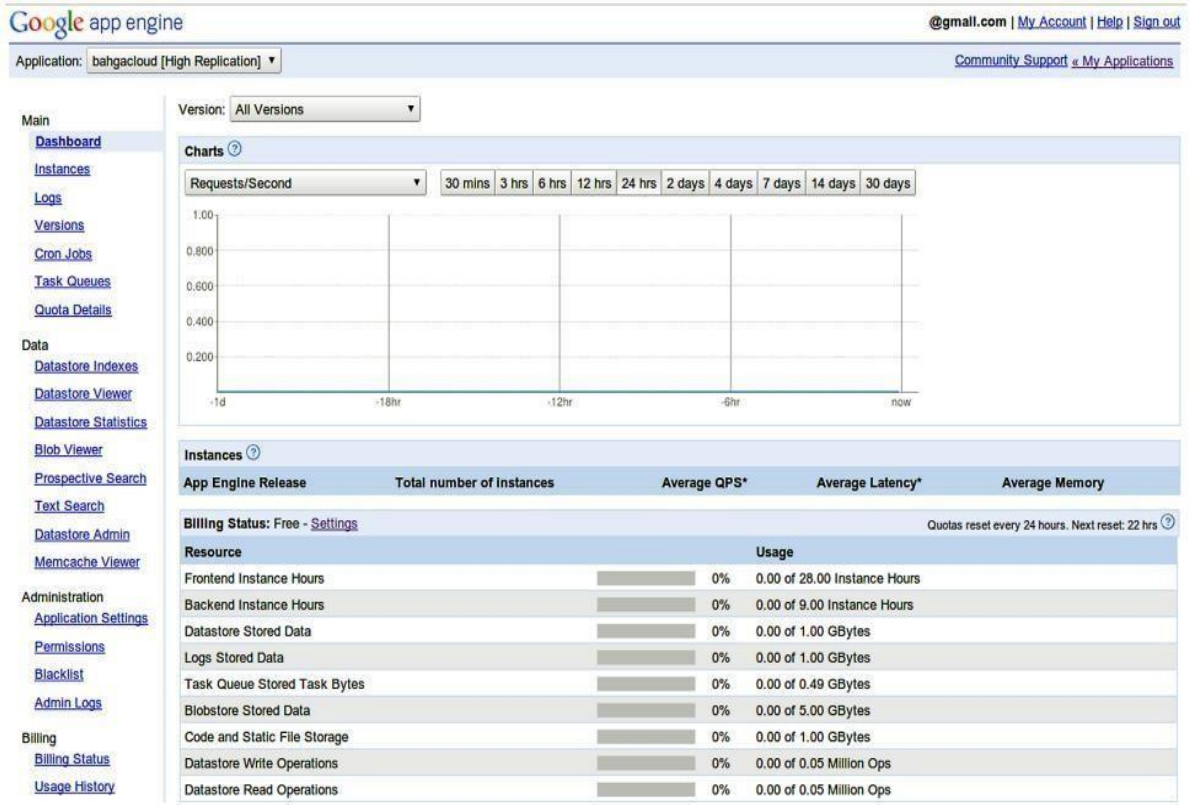
**11. Explain in detail about Application Services with examples.**

### Applications Runtimes and Frameworks

- **Cloud-based application runtimes and frameworks allow developers to develop and host applications in the cloud.**
- Support for various programming languages
  - ✓ Application runtimes provide support for programming languages (e.g., Java, Python, or Ruby).
- **Resource Allocation**
  - ✓ Application runtimes automatically allocate resources for applications and handle the application scaling, without the need to run and maintain servers.

### Google App Engine

- Google App Engine is the platform-as-a-service (PaaS) from Google, which includes both an application runtime and web frameworks.
- Runtimes
  - ✓ App Engine provides runtime environments for Java, Python, PHP and Go programming language.
- Sandbox
  - ✓ Applications run in a secure sandbox environmentisolated from other applications.
  - ✓ The sandbox environment provides a limited access tothe underlying operating system.




## Google App Engine

- **Web Frameworks**
  - ✓ App Engine provides a simple Python web application framework called webapp2. App Engine also supports any framework written in pure Python that speaks WSGI, including Django, CherryPy, Pylons, web.py, and web2py.
- **Datastore**
  - ✓ App Engine provides a no-SQL data storage service.
- **Authentication**
  - ✓ App Engine applications can be integrated with Google Accounts for user authentication.
- **URL Fetch service**
  - ✓ URL Fetch service allows applications to access resources on the Internet, such as web services or other data.
- **Other services**
  - ✓ Email service
  - ✓ Image Manipulation service
  - ✓ Memcache
  - ✓ Task Queues
  - ✓ Scheduled Tasks service

**Windows Azure Website**

- Windows Azure Web Sites is a Platform-as-a-Service (PaaS) from Microsoft.
- Azure Web Sites allows you to host web applications in the Azure cloud.
- Shared & Standard Options.
  - ✓ In the shared option, Azure Web Sites run on a set of virtual machines that may contain multiple websites created by multiple users.
  - ✓ In the standard option, Azure Web Sites run on virtual machines (VMs) that belong to an individual user.
- Azure Web Sites supports applications created in ASP.NET, PHP, Node.js and Python programming languages.
- Multiple copies of an application can be run in different VMs, with Web Sites automatically load balancing requests across them.