CS 3551 DISTRIBUTED COMPUTING

UNIT I

INTRODUCTION

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

Introduction:

1.1 Definition

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

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.
- **No 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. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem jointly.

1.2 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.

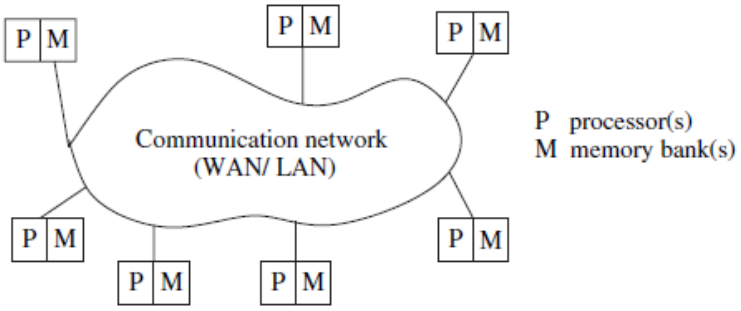


Figure 1.1 A distributed system connects processors by a communication network.

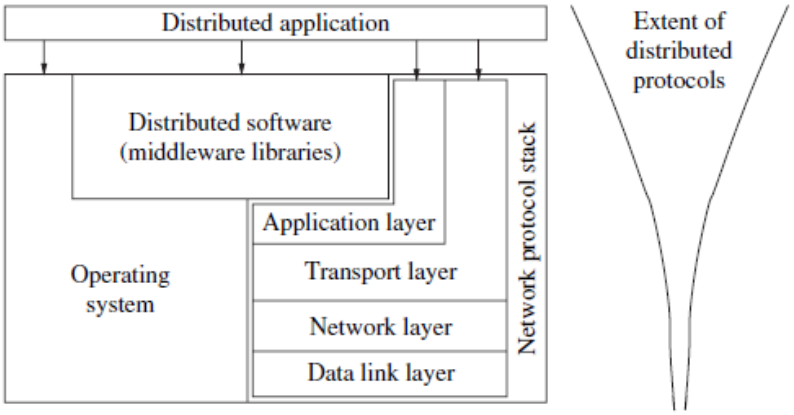


Figure 1.2 Interaction of the software components at each processor.

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

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

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 primitive 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 functions 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.

1.3 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. Therefore, such resources are typically distributed across the system. For example, distributed databases such as DB2 partition the data sets across several servers, in addition to replicating them at a few sites for rapid access as well as reliability
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. For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every branch office/site.

4. **Enhanced reliability:** A distributed system has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions, as well as the reality that geographically distributed resources are not likely to crash/malfunction at the same time under normal circumstances. 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, in the face of concurrent access from multiple processors, as per the semantics expected by the application;
    - fault-tolerance, i.e., the ability to recover from system failures
5. **Increased performance/cost ratio:** By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased. 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.
6. **Scalability**: As the processors are usually connected by a wide-area network, adding more processors does not pose a direct bottleneck for the communication network.
7. **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.

1.4 Message Passing Systems versus Shared Memory Systems

Shared memory systems are those in which there is a (common) shared address space throughout the system. Communication among processors takes place via shared data variables, and control variables for synchronization among the processors. Semaphores and monitors that were originally designed for shared memory uniprocessors and multiprocessors are examples of how synchronization can be achieved in shared memory systems.

All multicomputer (NUMA as well as message-passing) systems that do not have a shared address space provided by the underlying architecture and hardware necessarily communicate by message passing. Conceptually, programmers find it easier to program using shared memory than by message passing.

For this and several other reasons that we examine later, the abstraction called shared memory is sometimes provided to simulate a shared address space. For a distributed system, this abstraction is called distributed shared memory. Implementing this abstraction has a certain cost but it simplifies the task of the application programmer. There also exists a well-known folklore result that communication via message-passing can be simulated by communication via shared memory and vice-versa. Therefore, the two paradigms are equivalent.

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

The shared address space can be partitioned into disjoint parts, one part being assigned to each processor. "Send" and "receive" operations can be implemented by writing to and reading from the destination/sender processor's address space, respectively.

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

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

This involves the use of "send" and "receive" operations for "write" and "read" operations. Each shared location can be modeled as a separate process; "write" to a shared location is emulated by sending an update message to the corresponding owner process; a "read" to a shared location is emulated by sending a query message to the owner process. As accessing another processor's memory requires send and receive operations, this emulation is expensive.

Thus, the latencies involved in read and write operations may be high even when using shared memory emulation because the read and write operations are implemented by using network-wide communication under the covers.

In a MIMD message-passing multicomputer system, each “processor” may be a tightly coupled multiprocessor system with shared memory. Within the multiprocessor system, the processors communicate via shared memory. Between two computers, the communication is by message passing. As message-passing systems are more common and more suited for wide-area distributed systems, we will consider message-passing systems more extensively than we consider shared memory systems.

1.5 Primitives for distributed communication

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.

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

Fig. A non-blocking send primitive.

The code for a non-blocking Send would look as shown in Figure. 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.

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 $handle_k$. 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 $handle_k$ and wakes up (signals) any process with a Wait call blocked on this $handle_k$ 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 using a timing diagram. Here the timelines

are shown for each process: (1) for the process execution, (2) for the user buffer from/to which data is sent/received, and (3) for the kernel/communication subsystem.

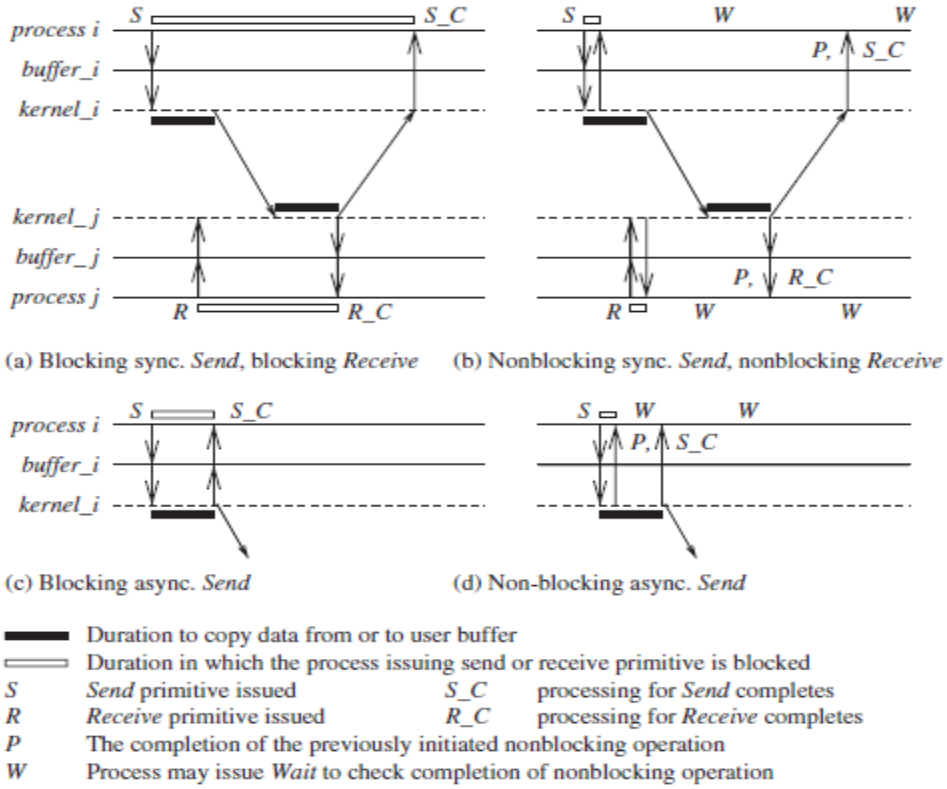


Figure: Blocking/ non-blocking and synchronous/asynchronous primitives.
Process Pi is sending and process Pj is receiving.
(a) Blocking synchronous Send and blockin (synchronous) Receive.
(b) Non-blocking synchronous Send and nonblocking (synchronous) Receive.
(c) Blocking asynchronous Send.
(d) Non-blocking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

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).

The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community, but other alternative libraries exist.

Commercial software is often written using the remote procedure calls (RPC) mechanism in which procedures that potentially reside across the network are invoked transparently to the user, in the same manner that a local procedure is invoked.

1.6 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,

(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. 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.

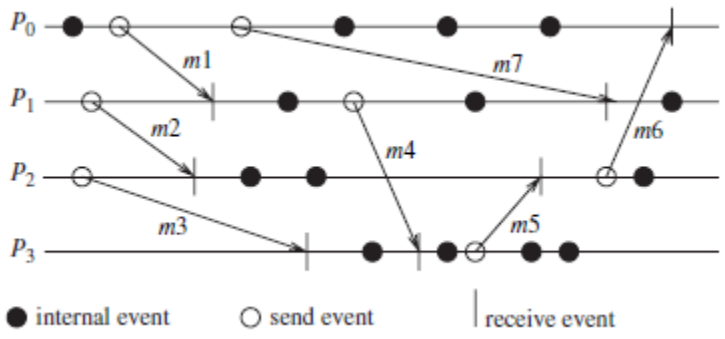


Figure: An example of an asynchronous execution in a message-passing system

A synchronous execution is an execution in which

(i) processors are synchronized and the clock drift rate between any two processors is bounded,

(ii) message delivery (transmission + delivery) times are such that they occur in one logical step or round,

(iii) there is a known upper bound on the time taken by a process to execute a step.

An example of a synchronous execution with four processes P0 to P3 is shown in Figure. The arrows denote the messages.

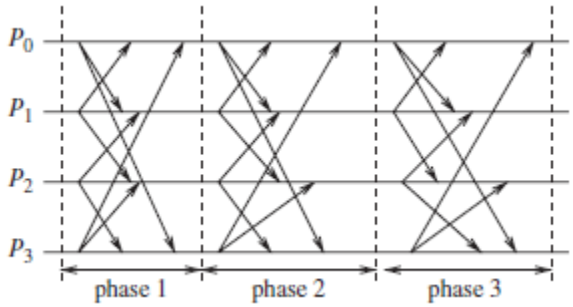


Figure: An example of a synchronous execution in a message-passing system

The 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

subphases 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.

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

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

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

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

Emulations

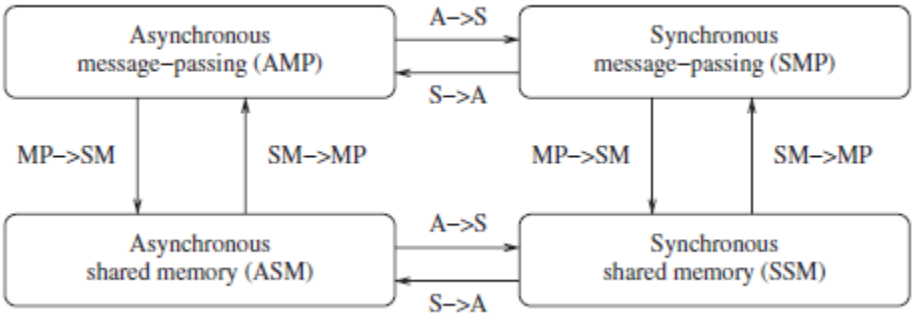


Figure: Emulations among the principal system classes in a failure free system.

There are four broad classes of programs, as shown in Figure. Using the emulations shown, any class can be emulated by any other. If system A can be emulated by system B denoted A/B, and if a problem is not solvable in B, then it is also not solvable in A. Likewise, if a problem is solvable in A, it is also solvable in B. Hence, in a sense, all four classes are equivalent in terms of "computability" – what can and cannot be computed – in failure-free systems.

1.7 Design issues and challenges

Distributed systems challenges from a system perspective

a. **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 cation (ROI), message-oriented communication versus stream-oriented communication.

b. **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.

c. **Naming:** Devising easy to use and robust schemes for names, identifiers, and addresses is essential for locating resources and processes in a transparent and scalable manner. Naming in mobile systems provides additional challenges because naming cannot easily be tied to any static geographical topology.

d. **Synchronization**: Mechanisms for synchronization or coordination among the processes are essential. Mutual exclusion is the classical example of synchronization, but many other forms of synchronization, such as leader election are also needed.

e. **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.

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

g. **Fault tolerance:** Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes.

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

i. **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.

a. Access Transparency hides differences in data representation on different systems and provides uniform operations to access system resources.
b. Location transparency makes the locations of resources transparent to the users.
c. Migration transparency allows relocating resources without changing names.
d. The ability to relocate the resources as they are being accessed is relocation transparency.
e. Replication transparency does not let the user become aware of any replication.
f. Concurrency transparency deals with masking the concurrent use of shared resources for the user.
g. Failure transparency refers to the system being reliable and fault-tolerant.

j. **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

a. 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.

b. 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 communication, data dissemination, object location, and object search functions.

c. 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.

d. 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. Here are some

examples of problems requiring synchronization. They are Physical clock synchronization, Leader election, Mutual exclusion, Deadlock detection and resolution, Termination detection and Garbage collection.

e. 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.

f. 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.

g. 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

h. 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.

i. 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.

j. World Wide Web design – caching, searching, scheduling

The Web is an example of a widespread distributed system with a direct interface to the end user, wherein the operations are predominantly read-intensive on most objects.

k. 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. Hence, in terms of overheads, the shared memory abstraction is not less expensive.

l. **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. They are Consensus algorithms, Replication and replica management, Voting and quorum systems, Distributed databases and distributed commit, Self-stabilizing systems, Checkpointing and recovery algorithms, Failure detectors.

m. **Load balancing**

The goal of load balancing is to gain higher throughput, and reduce the userperceived latency. The following are some forms of load balancing: Data migration, Computation migration and Distributed scheduling.

n. **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.

o. **Performance**

Although high throughput is not the primary goal of using a distributed system, achieving good performance is important. The following are some example issues arise in determining the performance: Metrics and Measurement methods/tools

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. Hence, 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

- **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. Recent developments in cost-effective hardware technology have made it possible to deploy very large (of the order of 106 or higher) low-cost sensors.

- **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, much like in sci-fi movies. The intelligent home, and the smart workplace are some example of ubiquitous environments currently under intense research and development.

- **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.

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

  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),

  (ii) an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information (subscribe)

  (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.

- **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.

- **Grid computing**

  Many challenges in making grid computing a reality include: scheduling jobs in such a distributed environment, a framework for implementing quality of service and real-time guarantees, and, of course, security of individual machines as well as of jobs being executed in this setting.

- **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.

A Model of Distributed Computations

1.8 A Distributed Program

A distributed program is composed of a set of n asynchronous processes p_1, p_2…, p_i… ,p_n that communicate by message passing over the communication network. 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 C_{ij} denote the channel from process pi to process p_j and let m_{ij} denote a message sent by pi to p_j . 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.

1.9 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^i denote the xth 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 $e_i^1, e_i^2, \ldots e_i^x, \ldots e_i^{x+1}, \ldots$ and it is denoted by H_i,

$$\mathcal{H}_i = (h_i, \rightarrow_i),$$

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

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

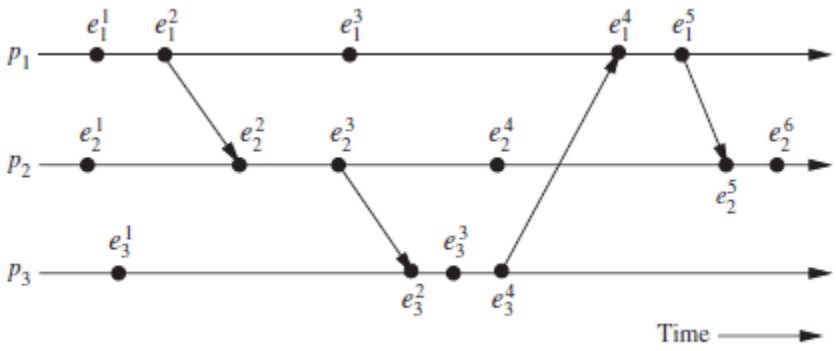


Figure: The space–time diagram of a distributed execution.

Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events. The evolution of a distributed execution is depicted by a space–time diagram. Figure shows the space–time diagram of a distributed execution involving three processes. A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.

Generally, the execution of an event takes a finite amount of time; however, since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to

denote it as a dot on a process line. In this figure, for process p_1, 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.

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

The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as H=(H, $\rightarrow$).

For any two events e_i and e_j , $e_i \not\rightarrow e_j$ 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.

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. Note that two or more events may be logically concurrent even though they do not occur at the same instant 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.

1.10 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 ordering. 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 the causal ordering model satisfies the following property:

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

That is, this property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation. Causally ordered delivery of messages implies FIFO message delivery. Furthermore, note that CO $\subset$ FIFO $\subset$ 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

1.11 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.

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,

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

For a global snapshot to be meaningful, the states of all the components of the distributed system must be recorded at the same instant. 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 = $\{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a consistent global state iff it satisfies the following condition:

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

That is, channel state $SC_{ik}^{y_i,z_k}$ and process state $LS_k^{z_k}$ must not include any message that process p_i sent after executing event e_i^{xi}.

In the distributed execution of Figure, 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 p_2 has recorded the receipt of message m_{12}, however, the state of p_1 has not recorded its send. On the contrary, a global state GS_2

consisting of local states $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is consistent; all the channels are empty except C_{21} that contains message m_{21}

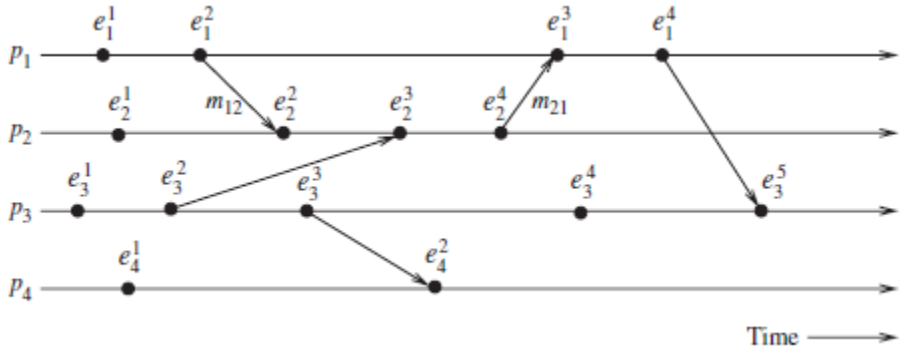


Figure: The space–time diagram of a distributed execution.

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

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

Thus, all channels are recorded as empty in a transitless global state. A global state is strongly consistent iff it is transitless as well as consistent. Note that in Figure, the global state consisting of local states $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is strongly consistent.

Recording the global state of a distributed system is an important paradigm when one is interested in analyzing, monitoring, testing, or verifying properties of distributed applications, systems, and algorithms. Design of efficient methods for recording the global state of a distributed system is an important problem.

CS 3551 DISTRIBUTED COMPUTING

UNIT II

LOGICAL TIME AND GLOBAL STATE

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

Logical Time

Definition

A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation <. This relation is usually called 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:

$$C : H \mapsto T,$$

such that the following property is satisfied: for two events e_i and e_j ,

$e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$. This monotonicity property is called the clock consistency condition.

When T and C satisfy the following condition, for two events

$$e_i \text{ and } e_j \text{ , } e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j),$$

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:

1. A *local logical clock*, denoted by lc_i, that helps process p_i measure its own progress.
2. 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:

1. R1 This rule governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal).
2. R2 This rule governs how a process updates its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

Scalar time

Definition:

The scalar time representation was proposed by Lamport in 1978 as an attempt to totally order events in a distributed system. Time domain in this representation is the set of non-negative integers. The logical local clock of a process pi and its local view of the global time are squashed into one integer variable C_i.

Rules **R1** and **R2** to update the clocks are as follows:

1. **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 time of each event uniquely at a process, while keeping the rate of increase of *d* to its lowest level.

2. **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 shows the evolution of scalar time with d=1.

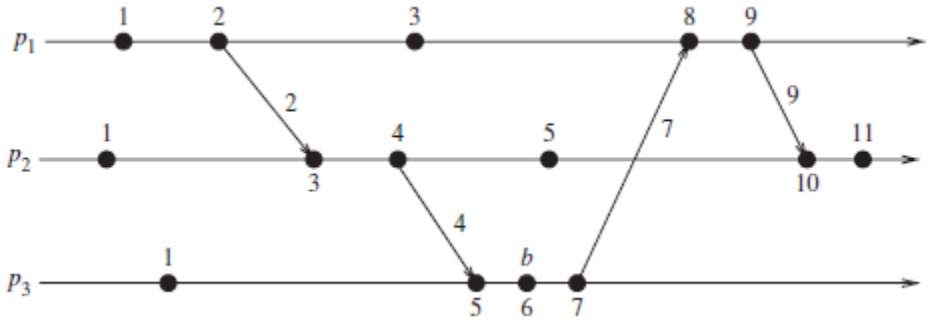


Figure: Evolution of scalar time

Basic properties

Consistency property

Clearly, scalar clocks satisfy the monotonicity and hence the consistency property:

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

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 at different processes may have an identical timestamp.

Event counting

If the increment value d is always 1, the scalar time has the following interesting 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.

No strong consistency

The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j,

$$C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$$

Vector time

Definition

The system of vector clocks was developed independently by Fidge, Mattern, and Schmuck. 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 $vt_i[1\ldots n]$, where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i. $vt_i[j]$ represents process p_i's latest knowledge of process p_i local time. If $vt_i[j] = x$, then process p_i knows that local time at process p_i has progressed till x. The entire vector vt_i constitutes p_i's view of the global logical time and is used to timestamp events.

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

1. R1 Before executing an event, process p_i updates its local logical time as follows:

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

2. 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:

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

   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 shows an example of vector clocks progress with the increment value d = 1. Initially, a vector clock is [0,0,0,….].

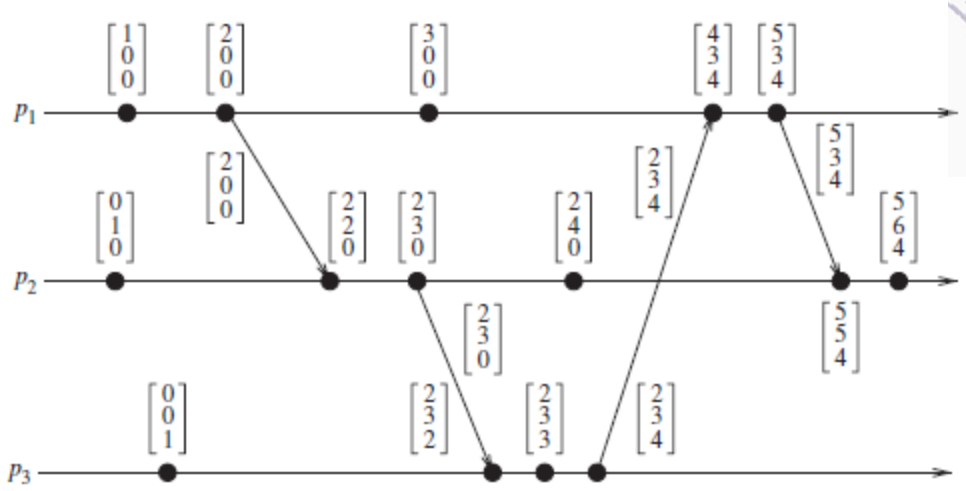


Figure: Evolution of vector time

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

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$
$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$
$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$
$$vh \parallel vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh).$$

Basic Properties

Isomorphism

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

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

Thus, there is an isomorphism between the set of partially ordered events produced by a distributed 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.

Event counting

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

Applications

Since vector time tracks causal dependencies exactly, it finds a wide variety of applications. For example, they are used in distributed debugging, implementations of causal ordering communication and causal distributed shared memory, establishment of global breakpoints, and in determining the consistency of checkpoints in optimistic recovery.

Size of vector clocks

- A vector clock provides the latest known local time at each other process. If this information in the clock is to be used to explicitly track the progress at every other process, then a vector clock of size *n* is necessary.
- A popular use of vector clocks is to determine the causality between a pair of events. Given any events *e* and *f*, the test for $e \prec f$ if and only if $T(e) < T(f)$, which requires a comparison of the vector clocks of *e* and *f*. Although it appears that the clock of size *n* is necessary, that is not quite accurate. It can be shown that a size equal to the dimension of the partial order $(E, \prec)$ is necessary, where the upper bound on this dimension is *n*.

Physical clock synchronization: NTP

In distributed systems, there is no global clock or common memory. Each processor has its own internal clock and its own notion of time. In practice, these clocks can easily drift apart by several seconds per day, accumulating significant errors over time. Also, because different clocks tick at different rates, they may not remain always synchronized although they might be synchronized when they start.

Some practical examples that stress the need for synchronization are listed below:

- In database systems, the order in which processes perform updates on a database is important to ensure a consistent, correct view of the database. To ensure the right ordering of events, a common notion of time between co-operating processes becomes imperative.
- It is quite common that distributed applications and network protocols use timeouts, and their performance depends on how well physically dispersed processors are time-synchronized. Design of such applications is simplified when clocks are synchronized.

Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time. It has a significant effect on many problems like secure systems, fault diagnosis and recovery, scheduled operations, database systems, and real-world clock values.

Due to different clocks rates, the clocks at various sites may diverge with time, and periodically a clock synchronization must be performed to correct this clock skew in distributed systems. Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time). Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed physical clocks.

Definitions and terminology

We provide the following definitions. C_a and C_b are any two clocks.

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

- **Drift (rate)**: The drift of clock C_a is the second derivative of the clock value with respect to time, namely, $C_a''(t)$. The drift of clock C_a relative to clock C_b at time t is $C_a''(t) - C_b''(t)$

Clock inaccuracies

Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).

However, due to the clock inaccuracy discussed above, a timer (clock) is said to be working within its specification if

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

where constant ρ is the maximum skew rate specified by the manufacturer.

Offset delay estimation method

The *Network Time Protocol (NTP)*, which is widely used for clock synchronization on the Internet, uses the the *offset delay estimation* method. The design of NTP involves a hierarchical tree of time servers. The primary server at the root synchronizes with the UTC. The next level contains secondary servers, which act as a backup to the primary server. At the lowest level is the synchronization subnet which has the clients.

Clock offset and delay estimation

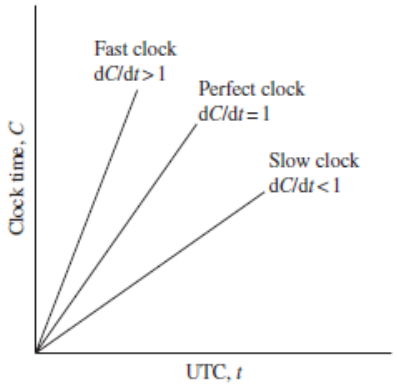


Figure: The behavior of fast, slow, and perfect clocks with respect to UTC

In practice, a source node cannot accurately estimate the local time on the target node due to varying message or network delays between the nodes. This protocol employs a very common practice of performing several trials and chooses the trial with the minimum delay. Recall that Cristian's remote clock reading method also relied on the same strategy to estimate message delay.

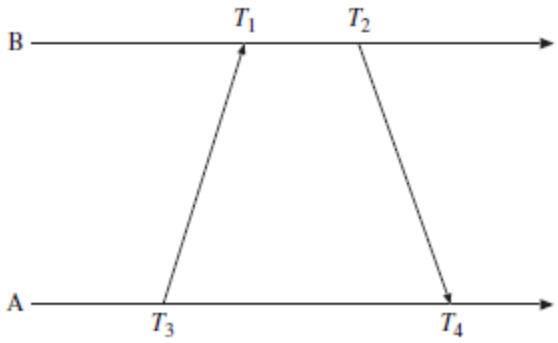


Figure: Offset and delay estimation

Figure shows how NTP timestamps are numbered and exchanged between peers A and B. Let T_1, T_2, T_3, T_4 be the values of the four most recent timestamps as shown. Assume that clocks A and B are stable and running at the same speed. Let $a = T_1 - T_3$ and $b = T_2 - T_4$. If the network delay difference from A to B and from B to A, called differential delay, is small, the clock offset θ and roundtrip delay δ of B relative to A at time T_4 are approximately given by the following:

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

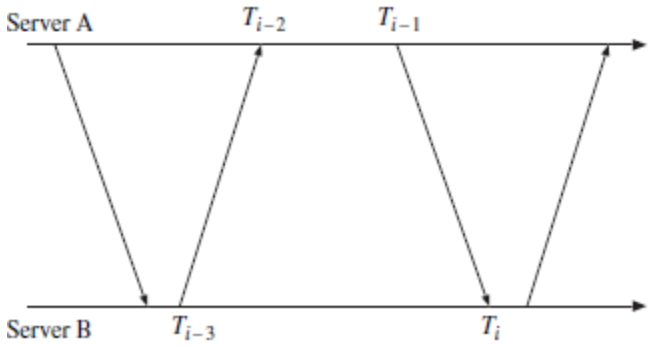


Figure: Timing diagram for the two servers

Each NTP message includes the latest three timestamps T_1, T_2, and T_3, while T_4 is determined upon arrival. Thus, both peers A and B can independently calculate delay and offset using a single bidirectional message stream as shown in Figure.

Message ordering and group communication

Message ordering paradigms

The order of delivery of messages in a distributed system is an important aspect of system executions because it determines the messaging behavior that can be expected by the distributed program. Distributed program logic greatly depends on this order of delivery.

Several orderings on messages have been defined:

(i) non-FIFO

(ii) FIFO

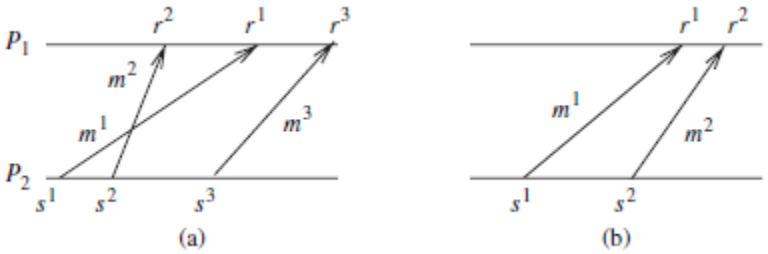
(iii) causal order, and

(iv) synchronous order

Asynchronous executions

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

On any logical link between two nodes in the system, messages may be delivered in any order, not necessarily first-in first-out. Such executions are also known as non-FIFO executions. 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. The following Figure (a) illustrates an A-execution under non-FIFO ordering.



a) An A-execution that is not a FIFO execution.

(b) An A-execution that is also a FIFO

FIFO executions

A FIFO execution is an A-execution in which,

for all (s, r) and $(s', r') \in$ T, $(s \sim s' and\ r \sim r' and\ s \prec s') \Rightarrow r \prec r'$

On any logical link in the system, messages are necessarily delivered in the order in which they are sent. Although the logical link is inherently non- FIFO, most network protocols provide a connection-oriented service at the transport layer.

A simple algorithm to implement a FIFO logical channel 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. The above Figure (b) illustrates an A-execution under FIFO ordering.

Causally ordered (CO) executions

A CO execution is an A-execution in which,

for all (s, r) and $(s', r') \in$ T, $(r \sim r' and\ s \prec s') \Rightarrow r \prec r'$

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

Causal order is useful for applications requiring updates to shared data, implementing distributed shared memory, and fair resource allocation such as granting of requests for distributed mutual exclusion.

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 (the "overtaken" messages) 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 (instead of as a receive event) for emphasis.

Definition of causal order (CO) for implementations

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.

Observe that 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.

Message order (MO)

A MO execution is an A-execution in which,

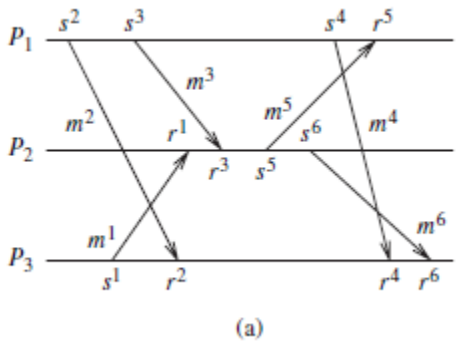
$$\text{for all } (s,r) \text{ and } (s',r') \in \mathrm{T}\ s \prec \mathrm{s}' \Rightarrow \neg(r \prec \mathrm{r}')$$

Empty-interval execution

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

Synchronous execution (SYNC)

When all the communication between pairs of processes uses synchronous send and receive primitives, the resulting order is the synchronous order. As each synchronous communication involves a handshake between the receiver and the sender, the corresponding send and receive events can be viewed as occuring instantaneously and atomically.



a) Execution in an Asynchronous system

b) Equivalent instantaneous communication

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

The "instantaneous communication" property of synchronous executions requires a modified definition of the causality relation because for each $(s, r) \in T$, the send event is not causally ordered before the receive event. The two events are viewed as being atomic and simultaneous, and neither event precedes the other.

Causality in a synchronous execution

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

S1: If x occurs before y at the same process, then $x \ll y$.
S2: If $(s, r) \in \mathcal{T}$, then for all $x \in E$, $[(x \ll s \iff x \ll r)$ and $(s \ll x \iff r \ll x)]$.
S3: If $x \ll y$ and $y \ll z$, then $x \ll z$.

Synchronous execution

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

Timestamping a synchronous execution

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

for any message M, $T(s(M)) = T(r(M))$;
for each process P_i, if $e_i \prec e_i'$ then $T(e_i) < T(e_i')$.

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, $e_i \prec e_j \Rightarrow T(e_i) < T(e_j)$.

2.1 Asynchronous execution with synchronous communication

When all the communication between pairs of processes is by using synchronous send and receive primitives, the resulting order is synchronous order. The algorithms run on asynchronous systems will not work in synchronous system and vice versa is also true.

Realizable Synchronous Communication (RSC)

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

- An execution can be modeled to give a total order that extends the partial order (E, $\prec$).
- In an A-execution, the messages can be made to appear instantaneous if there exist a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event in this linear extension.

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

A 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.

Crown

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: $s^0 \prec r^1$, $s^1 \prec r^2$, $s^{k-2} \prec r^{k-1}$, $s^{k-1} \prec r^0$.

The crown is $<(s^1, r^1) (s^2, r^2)>$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$. Cyclic dependencies may exist in a crown. The crown criterion states that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, if and only if it contains no crown.

Timestamp criterion for RSC execution

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

- for any message M, $T(s(M)) = T(r(M))$;
- for each (a, b) in $(E \times E) \setminus \mathcal{T}$, $a \prec b \Longrightarrow T(a) < T(b)$

2.2.1 Hierarchy of ordering paradigms

The orders of executions are:

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

The Execution order have the following results

- For an A-execution, A is RSC if and only if A is an S-execution.
- RSC $\subset$ CO $\subset$ FIFO $\subset$ A
- This hierarchy is illustrated in Figure 2.3(a), and example executions of each class are shown side-by-side in Figure 2.3(b)

- The above hierarchy implies that some executions belonging to a class X will not belong to any of the classes included in X. The degree of concurrency is most in A andleast in SYNC.
- A program using synchronous communication is easiest to develop and verify.
- A program using non-FIFO communication, resulting in an A execution, is hardest to design and verify.

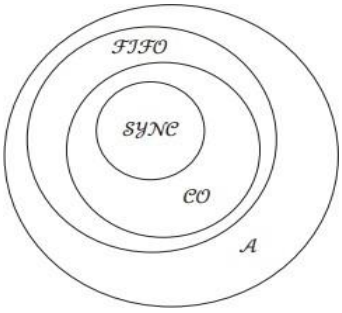


Fig (a)

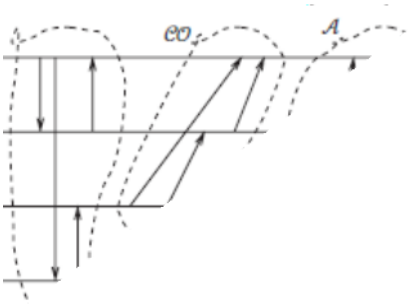


Fig (b)

Fig 2.3: Hierarchy of execution classes

2.2.3 Simulations

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

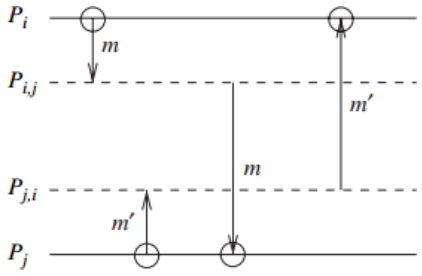


Fig 2.4: Modeling channels as processes to simulate an execution using asynchronous primitives on synchronous system

Synchronous programs on asynchronous systems

- A (valid) S-execution can be trivially realized on an asynchronous system by scheduling the messages in the order in which they appear in the S-execution.
- The partial order of the S-execution remains unchanged but the communication occurs

on an asynchronous system that uses asynchronous communication primitives.

– Once a message send event is scheduled, the middleware layer waits for acknowledgment; after the ack is received, the synchronous send primitive completes.

2.2 SYNCHRONOUS PROGRAM ORDER ON AN ASYNCHRONOUS SYSTEM

Non deterministic programs

The partial ordering of messages in the distributed systems makes the repeated runs of the same program will produce the same partial order, thus preserving deterministic nature. But sometimes the distributed systems exhibit non determinism:

- A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.
- Multiple send and receive calls which are enabled at a process can be executed in an interchangeable order.
- If i sends to j, and j sends to i concurrently using blocking synchronous calls, there results a deadlock.
- There is no semantic dependency between the send and the immediately following receive at each of the processes. If the receive call at one of the processes can be scheduled before the send call, then there is no deadlock.

2.3.1 Rendezvous

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

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

Features of binary rendezvous:

- For the receive command, the sender must be specified. However, multiple 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 receives commands that are both enabled. The communication events for the control messages under the covers do not alter the partial order of the execution.

2.3.2 Binary rendezvous algorithm

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

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

Steps in Bagrodia algorithm

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

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

(message types)

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

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

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

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

(2) P_i **wants to execute SEND(**M**) to a higher priority process** P_j**:**

(2a) Pi seeks permission from Pj by executing *send*(*request*(M)).

// to avoid deadlock in which cyclically blocked processes queue // messages.

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

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

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

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

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

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

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

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

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

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

Fig 2.5: Bagrodia Algorithm

2.3 GROUP COMMUNICATION

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

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

The network layer protocol cannot provide the following functionalities:

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

Differences between closed and open group algorithms:

Closed group algorithms	Open group algorithms
If sender is also one of the receiver in the multicast algorithm, then it is closed group algorithm.	If sender is not a part of the communication group, then it is open group algorithm.
They are specific and easy to implement.	They are more general, difficult to design and expensive.
It does not support large systems where client processes have short life.	It can support large systems.

2.4 CAUSAL ORDER (CO)

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

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

2.5.1 The Raynal–Schiper–Toueg algorithm

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

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

2.5 Causal Order (CO)

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 messageM sent in the causal past, as long as and only as long as:

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

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

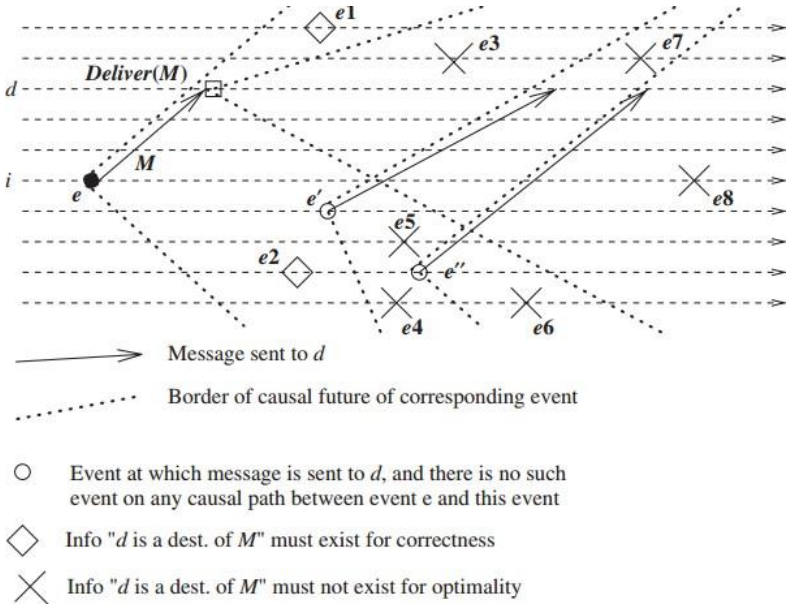


Fig 2.6: Conditions for causal ordering

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

- not in the causal future of $\text{Deliver}_d(M_{1,}\ a)$
- not in the causal future of $e_{k,\ c}$ where d $\in M_{k,c}$Dests and there is no other message sent causally between $M_{i,}a$ and $M_{k,\ c}$ to the same destination d.

Information about messages:

(i) not known to be delivered

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

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

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

(1a) $clock_j \longleftarrow clock_j + 1$;
(1b) **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;
(1c) **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 $l.Dests = \emptyset$
(1d) $LOG_j \longleftarrow LOG_j \bigcup \{(j, clock_j, Dests)\}$.

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

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

(2a) // Delivery Condition: ensure that messages sent causally before M are delivered.
for all $o_{m,t_m} \in O_M$ **do**
if $j \in o_{m,t_m}.Dests$ **wait until** $t_m \leq SR_j[m]$;
(2b) Deliver M; $SR_j[k] \longleftarrow t_k$;
(2c) $O_M \longleftarrow \{(k, t_k, Dests)\} \bigcup O_M$;
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
(2d) // Merge O_M and LOG_j by eliminating all redundant entries.
// Implicitly track "already delivered" & "guaranteed to be delivered in CO"
// messages.
for all $o_{m,t} \in O_M$ **and** $l_{s,t'} \in LOG_j$ **such that** $s = m$ **do**
if $t < t' \bigwedge l_{s,t} \notin LOG_j$ **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 non-redundant information of O_M into LOG_j
(2e) $PURGE\_NULL\_ENTRIES(LOG_j)$. // 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

Fig 2.7 b) Receive algorithm by Kshemkalyani–Singhal to optimally implement causal ordering

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

1. Explicit tracking:

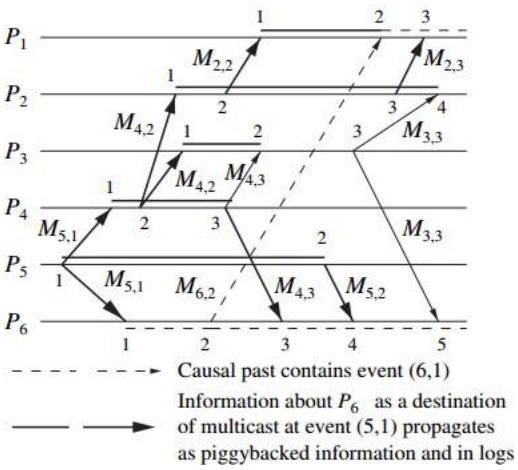
- Tracking of (source, timestamp, destination) information for messages (i) not known to be delivered and (ii) not guaranteed tobe delivered in CO, is done explicitly using the I.Dests

field of entries inlocal logs at nodes and o.Dests field of entries in messages.

- Sets $l_{i,a}$Destsand $o_{i,a}$. 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 $\in M_{i,a}$ .Destsis propagated up to the earliestevents on all causal paths from (i, a) at which it is known that $M_{i,a}$ isdelivered to d or is guaranteed to be delivered to d in CO.

2. Implicit tracking:

- Tracking of messages that are either (i) already delivered, or (ii) guaranteed to be delivered in CO, is performed implicitly.
- The information about messages (i) already delivered or (ii) guaranteed tobe delivered in CO is deleted and not propagated because it is redundantas far as enforcing CO is concerned.
- It is useful in determiningwhat information that is being carried in other messages and is being storedin logs at other nodes has become redundant and thus can be purged.
- Thesemantics are implicitly stored and propagated. This information about messages that are (i) already delivered or (ii) guaranteed to be delivered in CO is tracked without explicitly storing it.
- The algorithm derives it from the existing explicit information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, by examining only $o_{i,a}$Dests or $l_{i,a}$Dests, which is a part of the explicit information.



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

Fig 2.8: Illustration of propagation constraints

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

Message $M_{5,1}$ sent to processes P4 and P6 contains the piggybacked information $M_{5,1.}$ Dest= {P4, P6}. Additionally, at the send event (5, 1), the information $M_{5,1.}$Dests = {P4,P6} is also inserted in the local log Log_5. When $M_{5,1}$ is delivered to P6, the (new) piggybacked information P4 $\in M_{5,1}$ .Dests is stored in Log_6 as $M_{5,1.}$Dests ={P4} information about P6 $\in$ $M_{5,1.}$Dests which was needed for routing, must not be stored in Log_6 because of constraint I. In the same way when $M_{5,1}$ is delivered to process P4
at event (4, 1), only the new piggybacked information P6 $\in M_{5,1}$ .Dests is inserted in Log_4 as $M_{5,1.}$Dests =P6which is later propagated duringmulticast $M_{4,2.}$

Multicast $M_{4,3}$

At event (4, 3), the information P6 $\in M_{5,1.}$Dests in Log4 is propagated onmulticast $M_{4,3}$only to process P6 to ensure causal delivery using the DeliveryCondition. The piggybacked

information on message $M_{4,3}$sent to process P3must not contain this information because of constraint II. As long as any future message sent to P6 is delivered in causal order w.r.t. $M_{4,3}$sent to P6, it will also be delivered in causal order w.r.t. $M_{5,1.}$ And as $M_{5,1}$ is already delivered to P4, the information $M_{5,1}$Dests = ∅ is piggybacked on $M_{4,3}$ sent to P 3. Similarly, the information P6 ∈ $M_{5,1}$Dests must be deleted from Log4 as it will no longer be needed, because of constraint II. $M_{5,1}$Dests = ∅ is stored in Log4 to remember that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to all its destinations.

Learning implicit information at P2 and P3

When message $M_{4,2}$is received by processes P2 and P3, they insert the (new) piggybacked information in their local logs, as information $M_{5,1.}$Dests = P6. They both continue to store this in Log2 and Log3 and propagate this information on multicasts until they learn at events (2, 4) and (3, 2) on receipt of messages $M_{3,3}$and $M_{4,3}$, respectively, that any future message is expected to be delivered in causal order to process P6, w.r.t. $M_{5,1}$sent toP6. Hence by constraint II, this information must be deleted from Log2 andLog3. The flow of events is given by;

- When $M_{4,3}$ with piggybacked information $M_{5,1}$Dests = ∅ is received byP3at (3, 2), this is inferred to be valid current implicit information aboutmulticast $M_{5,1}$because the log Log3 already contains explicit informationP6 ∈$M_{5,1.}$Dests about that multicast. Therefore, the explicit informationin Log3 is inferred to be old and must be deleted to achieve optimality. $M_{5,1}$Dests is set to ∅ in Log3.
- The logic by which P2 learns this implicit knowledge on the arrival of $M_{3,3}$is identical.

Processing at P6

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

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

Processing at P1

- When $M_{2,2}$arrives carrying piggybacked information $M_{5,1.}$Dests = P6 this (new) information is inserted in Log1.
- When $M_{6,2}$arrives with piggybacked information $M_{5,1.}$Dests ={P4}, P1learns implicit information $M_{5,1}$has been delivered to P6 by the very absence of explicit information P6 ∈ $M_{5,1.}$Dests in the piggybacked information, and hence marks information P6 ∈ $M_{5,1}$Dests for deletion from Log1. Simultaneously, $M_{5,1}$Dests = P6 in Log1 implies the implicit information that $M_{5,1}$has been delivered or is guaranteed to be delivered in causal order to P4.Thus, P1 also learns that the explicit piggybacked information $M_{5,1.}$Dests = P4 is outdated. $M_{5,1.}$Dests in Log1 is set to ∅.

- The information "P6 $\in M_{5,1}$.Dests piggybacked on $M_{2,3}$,which arrives at P 1, is inferred to be outdated usingthe implicit knowledge derived from $M_{5,1}$.Dest= $\emptyset$" in Log1.

2.6 TOTAL ORDER

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

Centralized Algorithm for total ordering

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

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

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

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

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

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

Three phase distributed algorithm

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

Sender side

Phase 1

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

Phase 2

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

Phase 3

- The process multicasts the final timestamp to the group.

```
record Q_entry
        M: int;                                  // the application message
        tag: int;                                // unique message identifier
        sender_id: int;                          // sender of the message
        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).
```

Fig 2.9: Sender side of three phase distributed algorithm

Receiver Side

Phase 1

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

Phase 2

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

Phase 3

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

Complexity

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

2.7 GLOBAL STATE AND SNAPSHOT RECORDING ALGORITHMS

- A distributed computing system consists of processes that do not share a common memory and communicate asynchronously with eachother by message passing.
- Each component ofhas a local state. The state of the process is the local memory and a history of its activity.
- The state of achannel is characterized by the set of messages sent along the channel lessthe messages received along the channel. The global state of a distributedsystem is a collection of the local states of its components.
- If shared memory were available, an up-to-date state of the entire systemwould be available to the processes sharing the memory.
- The absence ofshared memory necessitates ways of getting a coherent and complete view ofthe system based on the local states of individual processes.
- A meaningfulglobal snapshot can be obtained if the components of the distributed systemrecord their local states at the same time.
- This would be possible if thelocal clocks at processes were perfectly synchronized or if there were aglobal system clock that could be instantaneously read by the processes.
- If processes read time froma single common clock, various indeterminatetransmission delays during the read operation will cause the processes toidentify various physical instants as the same time.

2.8.1 System Model

- The system consists of a collection of n processes, p_1, p_2,…,p_n that are connected by channels.
- Let C_{ij} denote the channel from process p_i to process p_j.
- Processes and channels have states associated with them.
- The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc., and may be highly dependent on the local context of the distributed application.
- The state of channel C_{ij}, denoted by SC_{ij}, is given by the set of messages in transit in the channel.
- The events that may happen are: internal event, send (send (m_{ij})) and receive (rec(m_{ij})) events.
- The occurrences of events cause changes in the processstate.
- A **channel** is a distributed entity and its state depends on the local states of the processes on which it is incident.

  Transit: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j\}$
- The transit function records the state of the channel $C_{ij.}$
- In the FIFO model, each channel acts as a first-in first-out message queue and, thus, message ordering is preserved by a channel.
- In the non-FIFO model, achannel acts like a set in which the sender process adds messages and thereceiver process removes messages from it in a random order.

2.8.2 A consistent global state

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

$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}$.

The two conditions for global state are:

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

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

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

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

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

2.8.3 Interpretation of cuts

- Cuts in a space–time diagram provide a powerful graphical aid in representingand reasoning about the global states of a computation. A cut is a line joiningan arbitrary point on each process line that slices the space–time diagraminto a PAST and a FUTURE.
- A consistent global state corresponds to a cut in which every messagereceived in the PAST of the cut has been sent in the PAST of that cut. Sucha cut is known as a consistent cut.
- In a consistent snapshot, all the recorded local states of processes are concurrent; that is, the recorded local state of no process casuallyaffects the recorded local state of any other process.

2.8.4 Issues in recording global state

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

Issue 1:

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

Answer:

- Any message that is sent by a process before recording its snapshot,must be recorded in the global snapshot (from C1).
- Any message that is sent by a process after recording its snapshot, mustnot be recorded in the global snapshot (from C2).

Issue 2:

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

The answer

Answer:

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

2.8 SNAPSHOT ALGORITHMS FOR FIFO CHANNELS

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

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

Snapshots are required to:

- Checkpointing
- Collecting garbage
- Detecting deadlocks
- Debugging

2.9.1 Chandy–Lamport algorithm

- The algorithm will record a global snapshot for each process channel.
- The Chandy-Lamport algorithm uses a control message, called a marker.
- Aftera site has recorded its snapshot, it sends a marker along all of its outgoingchannels before sending out any more messages.
- Since channels are FIFO, amarker separates the messages in the channel into those to be included in the snapshot from those not to be recorded inthe snapshot.
- This addresses issue I1. The role of markers in a FIFO systemis to act as delimiters for the messages in the channels so that the channelstate recorded by the process at the receiving end of the channel satisfies thecondition C2.

Marker sending rule for process p_i

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

Marker receiving rule for process p_j
On receiving a marker along channel C:
if p_j has not recorded its state **then**
Record the state of C as the empty set
Execute the "marker sending rule"
else
Record the state of C as the set of messages received along C after p_j's state was recorded and before p_j received the marker along C

Fig 2.10: Chandy–Lamport algorithm

Initiating a snapshot

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

Propagating a snapshot

- For all processes $P_j$consider a message on channel C_{kj}.

- If marker message is seen for the first time:
  - $P_j$records own sate and marks C_{kj} as empty
  - Send the marker message to all other processes.
  - Record all incoming messages from channels C_{lj} for 1 not equal to j or k.
  - Else add all messages from inbound channels.

Terminating a snapshot

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

Correctness of the algorithm

- Since a process records its snapshot when itreceives the first marker on any incoming channel, no messages that followmarkers on the channels incoming to it are recorded in the process's snapshot.
- A process stops recording the state of an incoming channel whena marker is received on that channel.
- Due to FIFO property of channels, itfollows that no message sent after the marker on that channel is recorded inthe channel state. Thus, condition C2 is satisfied.
- When a process p_j receives message m_{ij} that precedes the marker on channel C_{ij}, it acts as follows: ifprocess p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij}. Thus,condition C1 is satisfied.

Complexity

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

2.9.2 Properties of the recorded global state

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

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

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

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

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

UNIT III

DISTRIBUTED MUTEX & DEADLOCK

DISTRIBUTED MUTEX & DEADLOCK

Distributed mutual exclusion algorithms: Introduction – Preliminaries – Lamport‘s algorithm –Ricart-Agrawala algorithm – Maekawa‘s algorithm – Suzuki–Kasami‘s broadcast algorithm. Deadlock detection in distributed systems: Introduction – System model – Preliminaries –Models of deadlocks – Knapp‘s classification – Algorithms for the single resource model, the AND model and the OR model.

3.1 DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS

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

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

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

1. Token-based approach:

- A unique token is shared among all the sites.
- If a site possesses the unique token, it is allowed to enter its critical section
- This approach uses sequence number to order requests for the critical section.
- Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
- This approach insures Mutual exclusion as the token is unique.
- Eg: Suzuki-Kasami’s Broadcast Algorithm

2. Non-token-based approach:

- A site communicates with other sites in order to determine which sites should execute critical section next. This requires exchange of two or more successive round of messages among sites.
- This approach use timestamps instead of sequence number to order requests for the critical section.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport’s scheme.
- Eg: Lamport's algorithm, Ricart–Agrawala algorithm

3. Quorum-based approach:

- Instead of requesting permission to execute the critical section from all other sites, Each site requests only a subset of sites which is called a quorum.
- Any two subsets of sites or Quorum contains a common site.
- This common site is responsible to ensure mutual exclusion.
- Eg: Maekawa's Algorithm

3.1.1 Preliminaries

- The system consists of N sites, S_1, S_2, S_3, …, S_N.
- Assume that a single process is running on each site.
- The process at site S_i is denoted by p_i. All these processes communicate asynchronously over an underlying communication network.
- A 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.
- In the requesting the CS state, the site is blocked and cannot make further requests for the CS.
- In the idle state, the site is executing outside the CS.
- In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such state is referred to as the idle token state.
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.
- N denotes the number of processes or sites involved in invoking the critical section, T denotes the average message delay, and E denotes the average critical section execution time.

3.1.2 Requirements of mutual exclusion algorithms

- **Safety property:**

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.

- **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.

- **Fairness:**

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

3.1.3 Performance metrics

- **Message complexity:** This is the number of messages that are required per CS execution by a site.
- **Synchronization delay:** After a site leaves the CS, it is the time required and before the next site enters the CS. (Figure 3.1)
- **Response time:** This is the time interval a request waits for its CS execution to be over after its request messages have been sent out. Thus, response time does not include the time a request waits at a site before its request messages have been sent out. (Figure 3.2)
- **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.

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

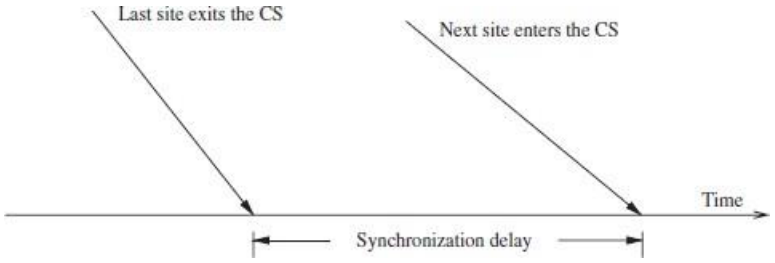


Figure 3.1 Synchronization delay

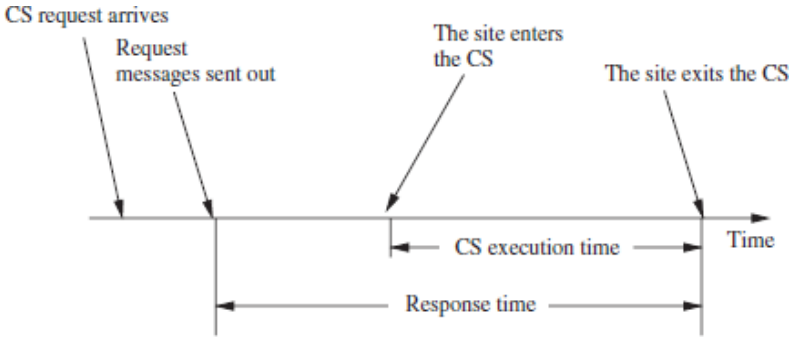


Figure 3.2 Response Time

Low and High Load Performance:

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

Best and worst case performance

- In the best case, prevailing conditions are such that a performance metric attains the best possible value. For example, the best value of the response time is a roundtrip message delay plus the CS execution time, 2T +E.

- For examples, the best and worst values of the response time are achieved when load is, respectively, low and high;
- The best and the worse message traffic is generated at low and heavy load conditions, respectively.

3.2 LAMPORT'S ALGORITHM

- Lamport's Distributed Mutual Exclusion Algorithm is a permission based algorithm proposed by Lamport as an illustration of his synchronization scheme for distributed systems.
- In permission based timestamp is used to order critical section requests and to resolve any conflict between requests.
- In Lamport's Algorithm critical section requests are executed in the increasing order of timestamps i.e a request with smaller timestamp will be given permission toexecute critical section first than a request with larger timestamp.
- Three type of messages (REQUEST, REPLY and RELEASE) are used and communication channels are assumed to follow FIFO order.
- A site send a REQUEST message to all other site to get their permission to enter critical section.
- A site send a REPLY message to requesting site to give its permission to enter the critical section.
- A site send a RELEASE message to all other site upon exiting the critical section.
- Every site Si, keeps a queue to store critical section requests ordered by their timestamps.
- request_queuei denotes the queue of site Si.
- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.

Requesting the critical section

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

Executing the critical section

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

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

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

Releasing the critical section

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

Fig 3.1: Lamport's distributed mutual exclusion algorithm

To enter Critical section:

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

To execute the critical section:

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

To release the critical section:

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

Correctness

Theorem: Lamport's algorithm achieves mutual exclusion.

Proof: Proof is by contradiction.

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

Theorem: Lamport's algorithm is fair.

***Proof: The proof is by contradiction*.**

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

Message Complexity:

Lamport's Algorithm requires invocation of 3(N – 1) messages per critical section execution. These 3(N – 1) messages involves

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

Drawbacks of Lamport's Algorithm:

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

Performance:

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

3.3 RICART–AGRAWALA ALGORITHM

- Ricart–Agrawala algorithm is an algorithm to for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala.
- This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm.
- It follows permission based approach to ensure mutual exclusion.
- Two type of messages (REQUEST and REPLY) are used and communication channels are assumed to follow FIFO order.
- A site send a REQUEST message to all other site to get their permission to enter critical section.
- A site send a REPLY message to other site to give its permission to enter the critical section.
- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests.
- Smaller timestamp gets high priority over larger timestamp.
- The execution of critical section request is always in the order of their timestamp.

Requesting the critical section

(a) When a site S_i wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.

(b) When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS, or if the site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. Otherwise, the reply is deferred and S_j sets $RD_j[i] := 1$.

Executing the critical section

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

Releasing the critical section

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

Fig 3.2: Ricart–Agrawala algorithm

To enter Critical section:

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

To execute the critical section:

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

To release the critical section:

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

Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.
***Proof: Proof is by contradiction*.**

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

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

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

Drawbacks of Ricart–Agrawala algorithm:

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

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

3.4 MAEKAWA's ALGORITHM

- Maekawa's Algorithm is quorum based approach to ensure mutual exclusion in distributed systems.

Requesting the critical section:
(a) A site S_i requests access to the CS by sending REQUEST(i) messages to all sites in its request set R_i.
(b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i 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 S_i executes the CS only after it has received a REPLY message from every site in R_i.
Releasing the critical section:
(d) After the execution of the CS is over, site S_i sends a RELEASE(i) message to every site in R_i.
(e) When a site S_j receives a RELEASE(i) message from site S_i, 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.

Fig 3.3: Maekawa's Algorithm

- In permission based algorithms like Lamport's Algorithm, Ricart-Agrawala Algorithm etc. a site request permission from every other site but in quorum based approach, a site does not request permission from every other site but from a subset ofsites which is called quorum.
- Three type of messages (REQUEST, REPLY and RELEASE) are used.
- A site send a REQUEST message to all other site in its request set or quorum to get their permission to enter critical section.
- A site send a REPLY message to requesting site to give its permission to enter the critical section.
- A site send a RELEASE message to all other site in its request set or quorum upon exiting the critical section

The following are the conditions for Maekawa's algorithm:

M1 $(\forall i\ \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$.
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 R_is, $1 \leq i, j \leq N$.

Maekawa used the theory of projective planes and showed that N = K(K – 1)+ 1. This relation gives |Ri|= √N.

To enter Critical section:

- When a site S_i wants to enter the critical section, it sends a request message REQUEST(i) to all other sites in the request set R_i.
- When a site S_j receives the request message REQUEST(i) from site S_i, it returns a REPLY message to site S_i if it has not sent a REPLY message to the site from the time it received the last RELEASE message. Otherwise, it queues up the request.

To execute the critical section:

- A site S_i can enter the critical section if it has received the REPLY message from all the site in request set R_i

To release the critical section:

- When a site S_i exits the critical section, it sends RELEASE(i) message to all other sites in request set R_i
- When a site S_j receives the RELEASE(i) message from site S_i, it send REPLY message to the next site waiting in the queue and deletes that entry from the queue
- In case queue is empty, site S_j update its status to show that it has not sent any REPLY message since the receipt of the last RELEASE message.

Correctness

Theorem: 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 ∩ Rj = {Sk }, then site Sk must have sent REPLY messages to both Si and Sj concurrently, which is a contradiction

Message Complexity:
Maekawa's Algorithm requires invocation of 3√N messages per critical section execution as the size of a request set is √N. These 3√N messages involves.

- √N request messages
- √N reply messages
- √N release messages

Drawbacks of Maekawa's Algorithm:

This algorithm is deadlock prone because a site is exclusively locked by other sites and requests are not prioritized by their timestamp.

Performance:
Synchronization delay is equal to twice the message propagation delay time. It requires 3√n messages per critical section execution.

3.5 SUZUKI–KASAMI's BROADCAST ALGORITHM

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

Requesting the critical section:

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

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

Executing the critical section:

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

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

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

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

(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.

Fig 3.4: Suzuki–Kasami's broadcast algorithm

To enter Critical section:

- When a site S_i wants to enter the critical section and it does not have the token then it increments its sequence number $RN_i[i]$ and sends a request message REQUEST(i, s_n) to all other sites in order to request the token.
- Here s_n is update value of $RN_i[i]$
- When a site S_j receives the request message REQUEST(i, s_n) from site S_i, it sets $RN_j[i]$ to maximum of $RN_j[i]$ and s_ni.e$RN_j[i] = \max(RN_j[i], s_n)$.
  After updating $RN_j[i]$, Site S_j sends the token to site S_i if it has token and $RN_j[i] = LN[i] + 1$

To execute the critical section:

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

To release the critical section:

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

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

Correctness

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

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

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

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

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

Message Complexity:

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

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

Drawbacks of Suzuki–Kasami Algorithm:

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

Performance:

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

3.6 DEADLOCK DETECTION IN DISTRIBUTED SYSTEMS

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

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

There are three approaches to detect deadlocks in distributed systems.

Centralized approach:

- Here there is only one responsible resource to detect deadlock.
- The advantage of this approach is that it is simple and easy to implement, while the drawbacks include excessive workload at one node, single point failure which in turns makes the system less reliable.

Distributed approach:

- In the distributed approach different nodes work together to detect deadlocks. No single point failure as workload is equally divided among all nodes.
- The speed of deadlock detection also increases.

Hierarchical approach:

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

System Model

- A distributed program is composed of a set of n asynchronous processes p1, p2, . . . , pi , . . . , pn that communicates by message passing over the communication network.
- Without loss of generality we assume that each process is running on a different processor.
- The processors do not share a common global memory and communicate solely by passing messages over the communication network.
- There is no physical global clock in the system to which processes have instantaneous access.
- The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.

We make the following assumptions:

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

Wait for graph

This is used for deadlock deduction. A graph is drawn based on the request and acquirement of the resource. If the graph created has a closed loop or a cycle, then there is a deadlock.

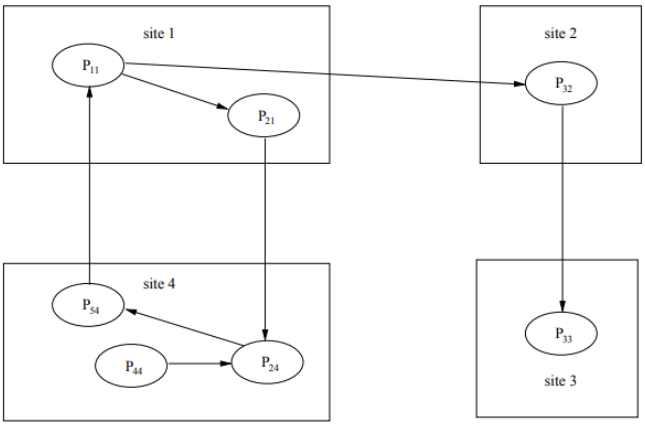


Fig 3.5: Wait for graph

Preliminaries

3.6.1 Deadlock Handling Strategies

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

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

3.6.2 Issues in deadlock Detection

Deadlock handling faces two major issues

1. Detection of existing deadlocks
2. Resolutionof detected deadlocks

Deadlock Detection

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

Correctness criteria

A deadlock detection algorithm must satisfy the following two conditions:

1. Progress-No undetected deadlocks:

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

2. Safety -No false deadlocks:

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

Resolution of a Detected Deadlock

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

3.7 MODELS OF DEADLOCKS

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

3.7.1 Single Resource Model

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

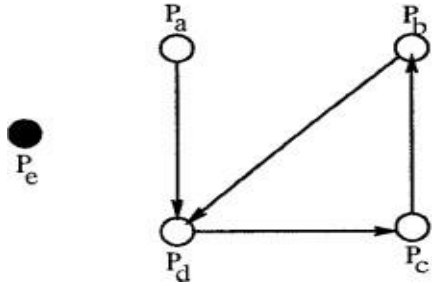


Fig 3.6: Deadlock in single resource model

3.7.2 AND Model

- In the AND model, a passive process becomes active (i.e., its activation condition is fulfilled) only after a message from each process in its dependent set has arrived.
- In the AND model, a process can request more than one resource simultaneously and 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 an AND node.

- In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked.

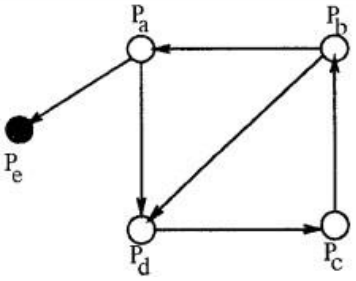


Fig 3.7: Deadlock in AND model

3.7.3 OR Model

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

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

- With every blocked process, there is an associated set of processes called **dependent set.**
- A process shall move from an idle to an active state on receiving a grant message from any of the processes in its dependent set.
- A process is permanently blocked if it never receives a grant message from any of the processes in its dependent set.
- A set of processes S is deadlocked if all the processes in S are permanently blocked.
- In short, a processis deadlocked or permanently blocked, if the following conditions are met:
  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.
- 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.

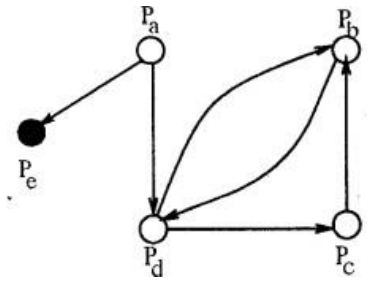


Fig 3.8: OR Model

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

- This is a variation of AND-OR model.

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

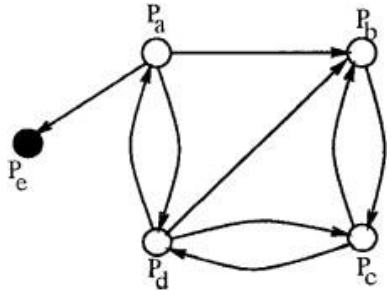


Fig 3.9: p out of q Model

3.7.5 Unrestricted model

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

3.8 KNAPP'S CLASSIFICATION OF DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

The four classes of distributed deadlock detection algorithm are:

1. Path-pushing
2. Edge-chasing
3. Diffusion computation
4. Global state detection

3.8.1 Path Pushing algorithms

- In path pushing algorithm, the distributed deadlock detection are detected by maintaining an explicit global wait for graph.
- The basic idea is to build a global WFG (Wait For Graph) for each site of the distributed system.
- At each site whenever deadlock computation is performed, it sends its local WFG to all the neighbouring sites.
- After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.
- This feature of sending around the paths of global WFGhas led to the term path-pushing algorithms.
  Examples: Menasce-Muntz , Gligor and Shattuck, Ho and Ramamoorthy, Obermarck

3.8.2 Edge Chasing Algorithms

- The presence of a cycle in a distributed graph structure is be verified by propagating special messages called probes, along the edges of the graph.
- These probe messages are different than the request and reply messages.
- The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.
- Whenever a process that is executing receives a probe message, it discards this message and continues.
- Only blocked processes propagate probe messages along their outgoing edges.
- Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short.
  Examples: Chandy et al., Choudhary et al., Kshemkalyani–Singhal, Sinha–Natarajan algorithms.

3.8.3 Diffusing Computation Based Algorithms

- In diffusion computation based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system.
- These algorithms make use of echo algorithms to detect deadlocks.
- This computation is superimposed on the underlying distributed computation.
- If this computation terminates, the initiator declares a deadlock.
- To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG.
- These queries are successively propagated (i.e., diffused) through the edges of the WFG.
- When a blocked process receives first query message for a particular deadlock detection initiation, it does not send a reply message until it has received a reply message for every query it sent.

- For all subsequent queries for this deadlock detection initiation, it immediately sends back a reply message.
- The initiator of a deadlock detection detects a deadlock when it receives reply for every query it had sent out.

Examples: Chandy–Misra–Haas algorithm for one OR model, Chandy–Herman algorithm

3.8.4 Global state detection-based algorithms

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

1. A consistent snapshot of a distributed system can be obtained without freezing the underlying computation.
2. If a stable property holds in the system before the snapshot collection is initiated, this property will still hold in the snapshot.

Therefore, distributed deadlocks can be detected by taking a snapshot of the system and examining it for the condition of a deadlock

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

- This deadlock detection algorithm assumes a single resource model.
- This detects the local and global deadlocks each process has assumed two different labels namely private and public each label is accountant the process id guarantees only one process will detect a deadlock.
- Probes are sent in the opposite direction to the edges of the WFG.
- When a probe initiated by a process comes back to it, the process declares deadlock.

Features:

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

Each node of the WFG has two local variables, called labels:

1. a private label, which is unique to the node at all times, though it is not constant.
2. a public label, which can be read by other processes and which may not be unique.

Each process is represented as u/v where u and u are the public and private labels, respectively. Initially, private and public labels are equal for each process. A global WFG is maintained and it defines the entire state sof the system.

- The algorithm is defined by the four state transitions as shown in Fig.3.10, where z = inc(u, v), and inc(u, v) yields aunique label greater than both u and v labels that are notshown do not change.
- The transitions in the defined by the algorithm are block, activate , transmit and detect.
- **Block** creates an edge in the WFG.
- Two messages are needed, one resource request and onemessage back to the blocked process to inform it of thepublic label of the process it is waiting for.
- **Activate** denotes that a process has acquired the resourcefrom the process it was

waiting for.

- **Transmit** propagates larger labels in the opposite directionof the edges by sending a probe message.

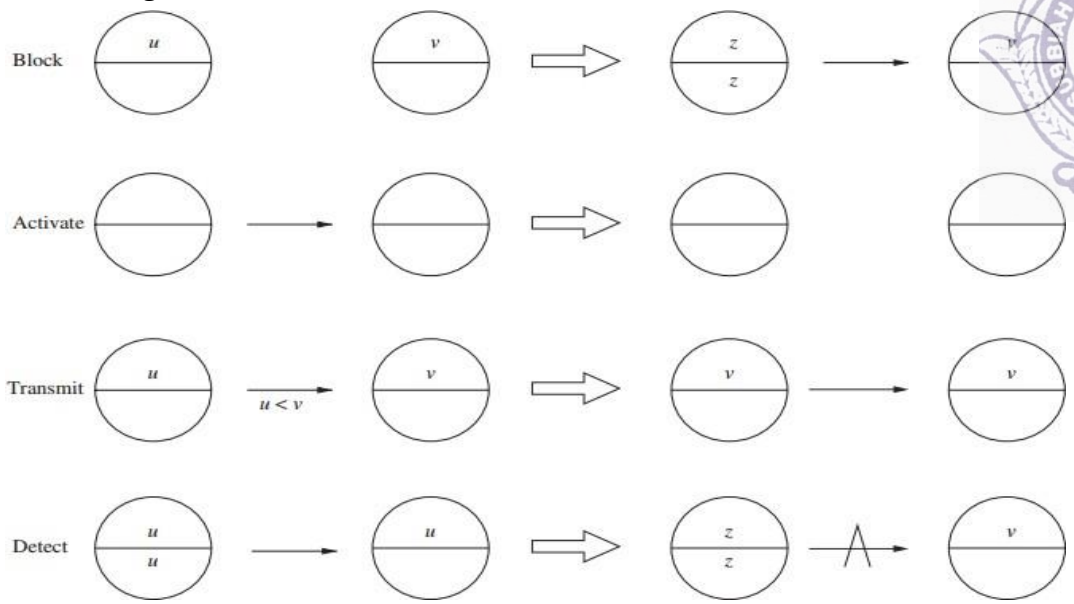


Fig 3.10: Four possible state transitions

- **Detect** means that the probe with the private label of some process has returned to it, indicating a deadlock.
- This algorithm can easily be extended to include priorities, so that whenever a deadlock occurs, the lowest priority process gets aborted.
- This priority based algorithm has two phases.
    1. The first phase is almost identical to the algorithm.
    2. The second phase the smallest priority is propagated around the circle. The propagation stops when one process recognizes the propagated priority as its own.

Message Complexity:

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

3.10 CHANDY–MISRA–HAAS ALGORITHM FOR THE AND MODEL

- This is considered an edge-chasing, probe-based algorithm.
- It is also considered one of the best deadlock detection algorithms for distributed systems.
- If a process makes a request for a resource which fails or times out, the process generates a probe message and sends it to each of the processes holding one or more of its requested resources.
- This algorithm uses a special message called probe, which is a triplet (i, j,k), denoting that it belongs to a deadlock detection initiated for process Pi andit is being sent by the home site of process Pj to the home site of process Pk.
- Each probe message contains the following information:
  - the id of the process that is blocked (the one that initiates the probe message);
  - the id of the process is sending this particular version of the probe message;
  - the id of the process that should receive this probe message.
- A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.
- A process P_j is said to be dependent on another process P_k if there exists a sequence of processes P_j, P_{i1} , P_{i2} , . . . , P_{im}, $P_k$such that each process except $P_k$in the sequence is blocked and each process, except the P_j, holds a resource for which the previous process in the sequence is waiting.
- Process P_j is said to be locally dependent upon process P_k if P_j is dependent upon $P_k$and both the processes are on the same site.
- When a process receives a probe message,it checks to see if it is also waiting for resources
- If not, it is currently using the needed resource and will eventually finish and release the resource.
- If it is waiting for resources, it passes on the probe message to all processes it knows to be holding resources it has itself requested.
- The process first modifies the probe message, changing the sender and receiver ids.
- If a process receives a probe message that it recognizes as having initiated,it knows there is a cycle in the system and thus, deadlock.

Data structures

Each process Pi maintains a boolean array, dependenti, where dependent(j) is true only if Pi knows that Pj is dependent on it. Initially, dependenti (j) is false for all i and j.

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

On the receipt of a probe (i, j, k), the site takes
   the following actions:

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

Fig 3.11: Chandy–Misra–Haas algorithm for the AND model

Performance analysis

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

Advantages:

- It is easy to implement.
- Each probe message is of fixed length.
- There is very little computation.
- There is very little overhead.
- There is no need to construct a graph, nor to pass graph information to other sites.
- This algorithm does not find false (phantom) deadlock.
- There is no need for special data structures.

3.11 CHANDY–MISRA–HAAS ALGORITHM FOR THE OR MODEL

- A blocked process determines if it is deadlocked by initiating a diffusion computation.
- Two types of messages are used in a diffusion computation:
  - query(i, j, k)
  - reply(i, j, k)

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

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

Initiate a diffusion computation for a blocked process P_i:
send $query(i, i, j)$ to all processes P_j in the dependent set DS_i of P_i;
$num_i(i) := |DS_i|$; $wait_i(i) := true$;

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

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

Fig 3.12: Chandy–Misra–Haas algorithm for the OR model

Performance analysis

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

UNIT IV
CONSENSUS AND RECOVERY

Consensusand agreement algorithms: Problem definition – Overview of results – Agreement in a failure – free system (Synchronous and Asynchronous) – Agreement in synchronous systems with failures. Check pointing and rollback recovery: Introduction – Background and definitions – Issues in failure recovery – Checkpoint-based recovery – Coordinated check pointing algorithm – Algorithm for asynchronous check pointing and recovery.

CONSENSUS PROBLEM IN ASYNCHRONOUS SYSTEMS.

Table: 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; common knowledge attainable	agreement attainable; concurrent common knowledge
Crash Failure	agreement attainable $f < n$ processes	agreement not attainable
Byzantie Failure	agreement attainable $f \leq [(n - 1)/3]$ Byzantine processes	agreement not attainable

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

Consensus Problem (all processes have an initial value)

Agreement: All non-faulty processes must agree on the same (single) value.

Validity: If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.

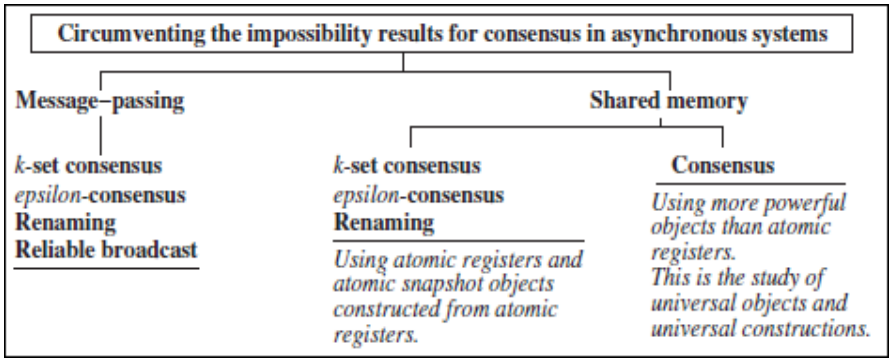
Termination: Each non-faulty process must eventually decide on a value.

Consensus Problem in Asynchronous Systems.

The overhead bounds are for the given algorithms, and not necessarily tight bounds for the problem.

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

Circumventing the impossibility results for consensus in asynchronous systems:



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

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor \frac{n-1}{3} \rfloor$;
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 $v^{concat(\langle j \rangle, L)}_{k}$, $\forall k$ such that $k \neq j$, i and k is not a member of list L.

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

(1) Initiator (i.e., Commander) initiates Oral Byzantine agreement:
(1a) **send** $OM(v, N-\{i\}, \langle P_i \rangle, f)$ to $N-\{i\}$;
(1b) **return**(v).

(2) (Non-initiator, i.e., Lieutenant) receives Oral Message *OM*:
(2a) **for** $rnd = 0$ **to** f **do**
(2b) **for** each message OM that arrives in this round, **do**
(2c) **receive** $OM(v, Dests, L = \langle P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty)$ from P_{k_1};
// $faulty + round = f$; $|Dests| + sizeof(L) = n$
(2d) $v^{tail(L)}_{head(L)} \longleftarrow v$; // $sizeof(L) + faulty = f + 1$. fill in estimate.
(2e) **send** $OM(v, Dests - \{i\}, \langle P_i, P_{k_1} \ldots P_{k_{f+1-faulty}} \rangle, faulty - 1)$ to $Dests - \{i\}$ **if** $rnd < f$;
(2f) **for** $level = f - 1$ **down to** 0 **do**
(2g) **for** each of the $1 \cdot (n-2) \cdot \ldots \cdot (n-(level+1))$ nodes v^{L}_{x} in level *level*, **do**
(2h) $v^{L}_{x}(x \neq i, x \notin L) = majority_{y \notin concat(\langle x \rangle, L); y \neq i}(v^{L}_{x}, v^{concat(\langle x \rangle, L)}_{y})$;

Byzantine Agreement (single source has an initial value) Agreement:

All non faulty processes must agree on the same value.

Validity: If the source process is non-faulty, then the agreed upon value by all the non- faulty processes must be the same as the initial value of the source.

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

(variables)
boolean: $v \longleftarrow$ initial value;
integer: $f \longleftarrow$ maximum number of malicious processes, $\leq \lfloor (n-1)/3 \rfloor$;
(message type)
OralMsg(v, *Dests*, *List*, *faulty*), where
v is a boolean,
Dests is a set of destination process ids to which the message is sent,
List is a list of process ids traversed by this message, ordered from most recent to earliest,
faulty is an integer indicating the number of malicious processes to be tolerated.

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

1. The algorithm is initiated by the Commander, who sends his source value v to all other processes using a $OM(v, N, \langle i \rangle, f)$ message. The commander returns his own value v and terminates.
2. **[Recursion unfolding:]** For each message of the form $OM(v_j, Dests, List, f')$ received in this round from some process j, the process i uses the value v_j it receives from the source, and using that value, acts as a *new* source. (If no value is received, a default value is assumed.)
   To act as a new source, the process i initiates $OralMsg(f' - 1)$, 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.

OralMsg(0):

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

CODE FOR THE PHASE KING ALGORITHM:

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

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

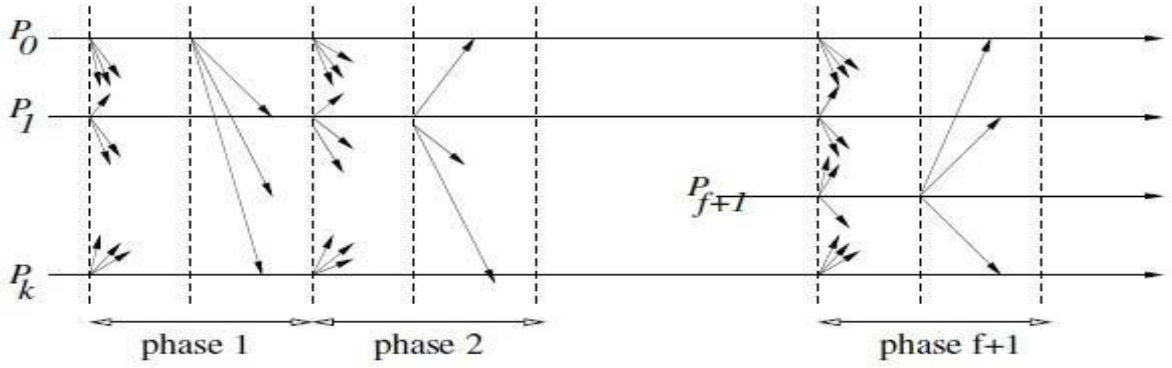


Fig. Message pattern for the phase-king algorithm.

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

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

PHASE KING ALGORITHM CODE:

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

Correctness Argument

- 1 Among f + 1 phases, at least one phase k where phase-king is non-malicious.
- 2 In phase k, all non-malicious processes Pi and Pj will have same estimate of consensus value as Pk does.
- Pi and Pj use their own majority values. Pi 's mult > n=2 + f)
- Pi uses its majority value; Pj uses phase-king's tie-breaker value. (Pi's mult > n=2 + f , Pj 's mult > n=2 for same value)
- Pi and Pj use the phase-king's tie-breaker value. (In the phase in which Pk is non-malicious, it sends same value to Pi and Pj)

In all 3 cases, argue that Pi and Pj end up with same value as estimate

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

CODE FOR THE EPSILON CONSENSUS (MESSAGE-PASSING, ASYNCHRONOUS):

Agreement: All non-faulty processes must make a decision and the values decided upon by any two non-faulty processes must be within range of each other.

Validity: If a non-faulty process Pi decides on some value vi , then that value must be within the range of values initially proposed by the processes.

Termination: Each non-faulty process must eventually decide on a value. The algorithm for the message-passing model assumes n ≥ 5f + 1, although the problem is solvable for n > 3f + 1.

- Main loop simulates sync rounds.
- Main lines (1d)-(1f): processes perform all-all msg exchange
- Process broadcasts its estimate of consensus value, and awaits n - f similar
- msgs from other processes

- the processes' estimate of the consensus value converges at a particular rate,
- until it is _ from any other processes estimate.
- # rounds determined by lines (1a)-(1c).

```
(variables)
real: v ⟵ input value;                                          //initial value
multiset of real V;
integer r ⟵ 0;                                    // number of rounds to execute

(1) Execution at process P_i, 1 ≤ i ≤ n:
(1a) V ⟵ Asynchronous_Exchange(v, 0);
(1b) v ⟵ any element in(reduce^{2f}(V));
(1c) r ⟵ ⌈log_c(diff(V))/ε⌉, where c = c(n − 3f, 2f).
(1d) for round from 1 to r do
(1e)      V ⟵ Asynchronous_Exchange(v, round);
(1f)      v ⟵ new_{2f,f}(V);
(1g) broadcast (⟨v, halt⟩, r + 1);
(1h) output v as decision value.

(2) Asynchronous_Exchange(v,h) returns V:
(2a) broadcast (v, h) to all processes;
(2b) await n − f responses belonging to round h;
(2c)      for each process P_k that sent ⟨x, halt⟩ as value, use x as its input henceforth;
(2d) return the multiset V.
```

TWO-PROCESS WAIT-FREE CONSENSUS USING FIFO QUEUE, COMPARE & SWAP:

Wait-free Shared Memory Consensus using Shared Objects:

Not possible to go from bivalent to univalent state if even a single failure is allowed. Difficulty is not being able to read & write a variable atomically.

- It is not possible to reach consensus in an asynchronous shared memory system using Read/Write atomic registers, even if a single process can fail by crashing.
- There is no wait-free consensus algorithm for reaching consensus in an asynchronous

shared memory system using Read/Write atomic registers.

To overcome these negative results:

- Weakening the consensus problem, e.g., k-set consensus, approximate consensus, and renaming using atomic registers.
- Using memory that is stronger than atomic Read/Write memory to design wait- free consensus algorithms. Such a memory would need corresponding access primitives.

Are there objects (with supporting operations), using which there is a wait-free (i.e., (n -1)- crash resilient) algorithm for reaching consensus in a n-process system? Yes, e.g., Test&Set, Swap, Compare&Swap. The crash failure model requires the solutions to be wait-free.

TWO-PROCESS WAIT-FREE CONSENSUS USING FIFO QUEUE:

```
(shared variables)
queue: Q ⟵ ⟨0⟩;                                   // queue Q initialized
integer: Choice[0, 1] ⟵ [⊥, ⊥]                    // preferred value of each process
(local variables)
integer: temp ⟵ 0;
integer: x ⟵ initial choice;

(1) Process P_i, 0 ≤ i ≤ 1, executes this for 2-process consensus using a FIFO queue:
(1a) Choice[i] ⟵ x;
(1b) temp ⟵ dequeue(Q);
(1c) if temp = 0 then
(1d)      output(x)
(1e) else output(Choice[1 − i]).
```

WAIT-FREE CONSENSUS USING COMPARE & SWAP:

```
(shared variables)
integer: Reg ⟵ ⊥;                                 // shared register Reg initialized
(local variables)
integer: temp ⟵ 0;                                // temp variable to read value of Reg
integer: x ⟵ initial choice;                      // initial preference of process

(1) Process P_i, (∀i ≥ 1), executes this for consensus using Compare&Swap:
(1a) temp ⟵ Compare&Swap(Reg, ⊥, x);
(1b) if temp = ⊥ then
(1c)      output(x)
(1d) else output(temp).
```

NONBLOCKING UNIVERSAL ALGORITHM:

Universality of Consensus Objects

An object is defined to be universal if that object along with read/write registers can simulate any other object in a wait-free manner. In any system containing up to k processes, an object X such that CN(X) = k is universal.

For any system with up to k processes, the universality of objects X with consensus number k is shown by giving a universal algorithm to wait-free simulate any object using objects of type X and read/write registers.

This is shown in two steps.

- 1 A universal algorithm to wait-free simulate any object whatsoever using read/write registers and arbitrary k-processor consensus objects is given. This is the main step.
- 2 Then, the arbitrary k-process consensus objects are simulated with objects of type X, having consensus number k. This trivially follows after the first step.

Any object X with consensus number k is universal in a system with $n \leq k$ processes.

A nonblocking operation, in the context of shared memory operations, is an operation that may not complete itself but is guaranteed to complete at least one of the pending operations in a finite number of steps.

Nonblocking Universal Algorithm:

The linked list stores the linearized sequence of operations and states following each operation.

Operations to the arbitrary object Z are simulated in a nonblocking way using an arbitrary consensus object (the field op.next in each record) which is accessed via the Decide call.

Each process attempts to thread its own operation next into the linked list.

- There are as many universal objects as there are operations to thread.
- A single pointer/counter cannot be used instead of the array Head. Because reading and updating the pointer cannot be done atomically in a wait-free manner.
- Linearization of the operations given by the sequence number. As algorithm is nonblock

4.1 Check pointing and rollback recovery: Introduction

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

Checkpoints

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

Why is rollback recovery of distributed systems complicated?

Messages induce inter-process dependencies during failure-free operation

Rollback propagation

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

Uncoordinated check pointing

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

Techniques that avoid domino effect

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

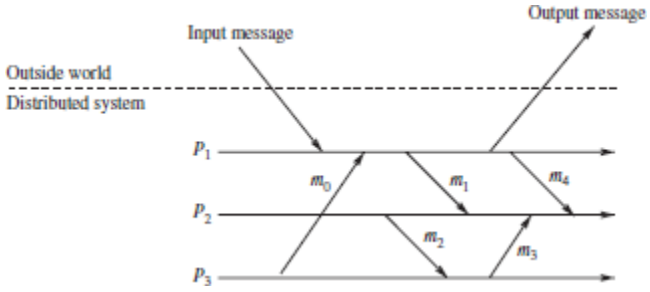
4.2 Background and definitions

4.2.1 System model

- A distributed system consists of a fixed number of processes, P1, P2,…_ PN , which communicate only through messages.
- Processes cooperate to execute a distributed application and interact with the outside world by receiving and sending input and output messages, respectively.
- Rollback-recovery protocols generally make assumptions about the reliability of theinter-

process communication.

- Some protocols assume that the communication uses first-in-first-out (FIFO) order, while other protocols assume that the communication subsystem can lose, duplicate, or reorder messages.
- Rollback-recovery protocols therefore must maintain information about the internal interactions among processes and also the external interactions with the outside world.



An example of a distributed system with three processes.

4.2.2 A local checkpoint

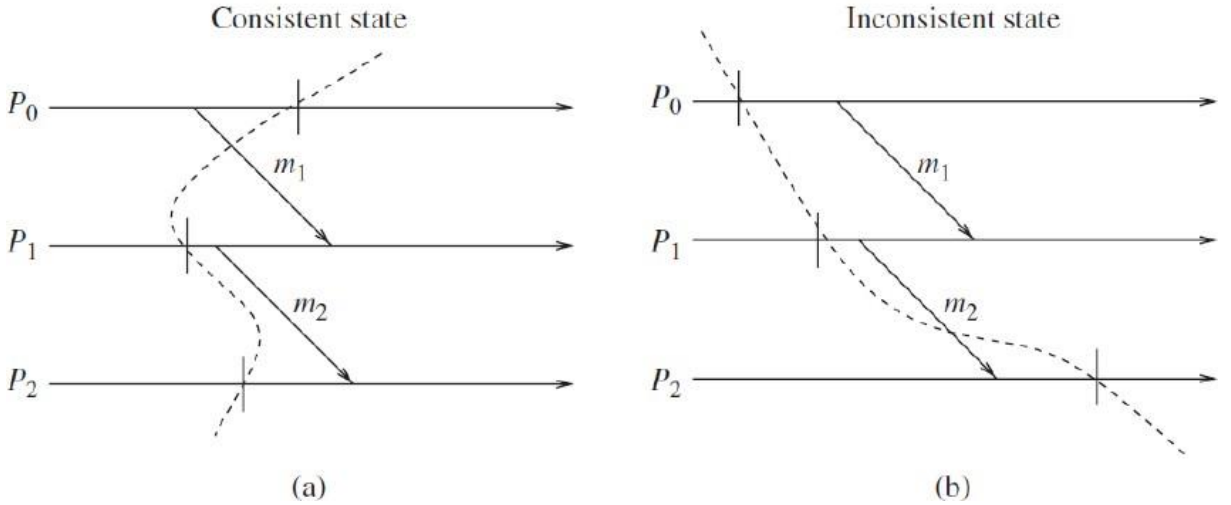
- All processes save their local states at certain instants of time
- A local check point is a snapshot of the state of the process at a given instance
- Assumption
  - A process stores all local checkpoints on the stable storage
  - A process is able to roll back to any of its existing local checkpoints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

4.2.3 Consistent states

- A global state of a distributed system is a collection of the individual states of all participating processes and the states of the communication channels
- Consistent global state
  - a global state that may occur during a failure-free execution of distribution of distributed computation
  - if a process‟s state reflects a message receipt, then the state of the corresponding sender must reflect the sending of the message
- A global checkpoint is a set of local checkpoints, one from each process

- A consistent global checkpoint is a global checkpoint such that no message is sent by a process after taking its local point that is received by another process before taking its checkpoint.

Consistent states - examples



- For instance, Figure shows two examples of global states.
- The state in fig (a) is consistent and the state in Figure (b) is inconsistent.
- Note that the consistent state in Figure (a) shows message *m*1 to have been sent but not yet received, but that is alright.
- The state in Figure (a) is consistent because it represents a situation in which every message that has been received, there is a corresponding message send event.
- The state in Figure (b) is inconsistent because process *P*2 is shown to have received *m*2 but the state of process *P*1 does not reflect having sent it.
- Such a state is impossible in any failure-free, correct computation. Inconsistent states occur because of failures.

4.2.4 Interactions with outside world

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

Outside World Process (OWP)

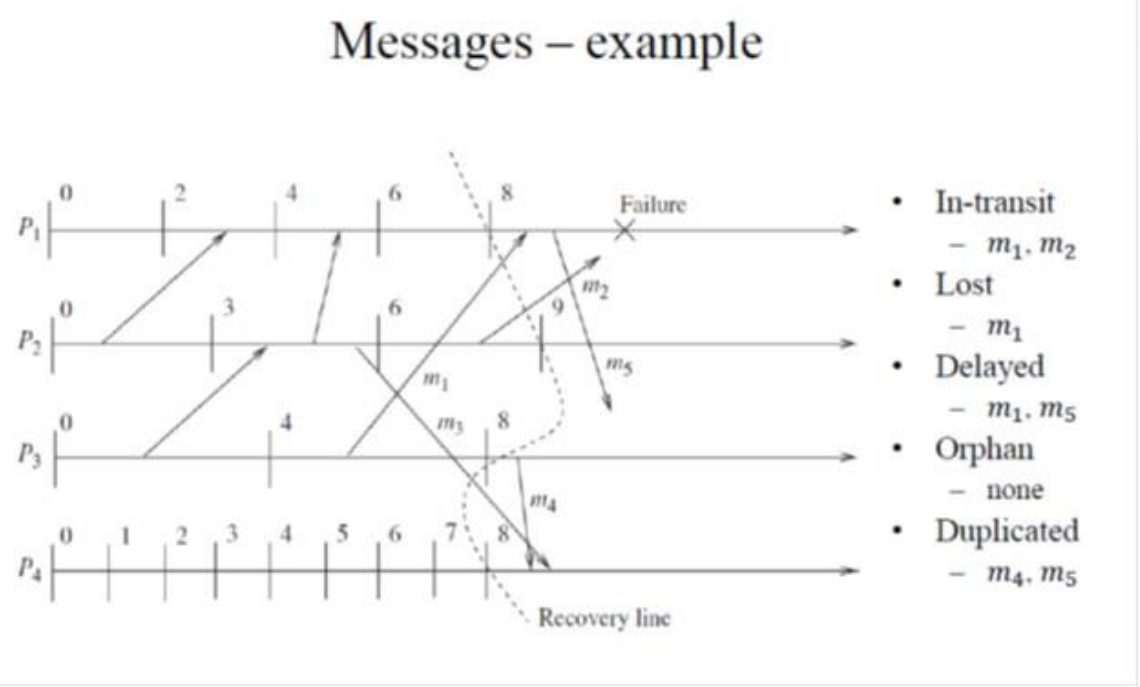
- It is a special process that interacts with the rest of the system through message passing.
- It is therefore necessary that the outside world see a consistent behavior of the system despite failures.
- Thus, before sending output to the OWP, the system must ensure that the state from which the output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

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

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

4.2.5 Different types of Messages

1. In-transit message
    - messages that have been sent but not yet received
2. Lost messages
    - messages whose "send" is done but "receive" is undone due to rollback
3. Delayed messages
    - messages whose "receive" is not recorded because the receiving process was either down or the message arrived after rollback
4. Orphan messages
    - messages with "receive" recorded but message "send" not recorded
    - do not arise if processes roll back to a consistent global state
5. Duplicate messages
    - arise due to message logging and replaying during process recovery



In-transit messages

In Figure , the global state {C1,8 , C2, 9 , C3,8, C4,8} shows that message *m*1 has been sent but not yet received. We call such a message an *in-transit* message. Message *m*2 is also an in-transit message.

Delayed messages

Messages whose receive is not recorded because the receiving process was either down or the message arrived after the rollback of the receiving process, are called *delayed* messages. For example, messages m2 and m5 in Figure are delayed messages.

Lost messages

Messages whose send is not undone but receive is undone due to rollback are called lostmessages. This type of messages occurs when the process rolls back to a checkpoint prior to reception of the message while the sender does not rollback beyond the send operation of the message. In Figure , message m1 is a lost message.

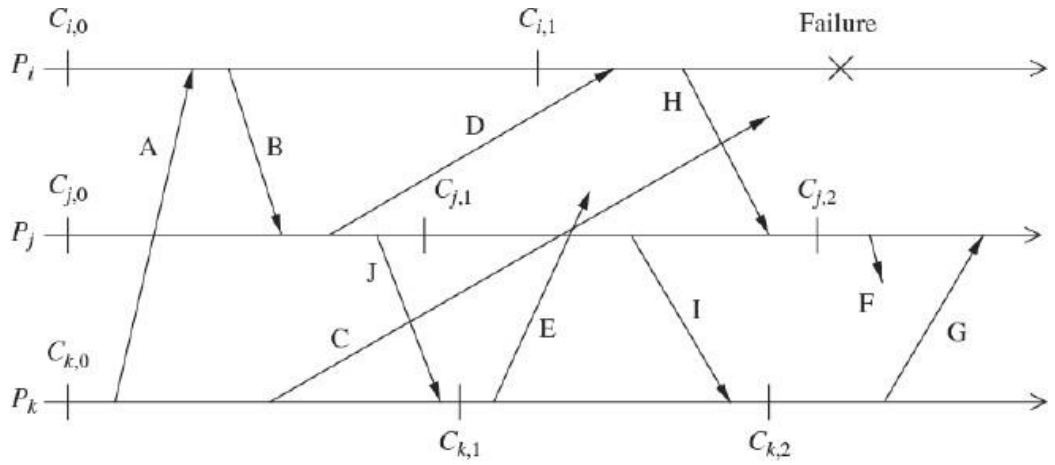
Duplicate messages

- Duplicate messages arise due to message logging and replaying during process recovery. For example, in Figure, message m4 was sent and received before the rollback. However, due to the rollback of process P4 to C4,8 and process P3 to C3,8, both send and receipt of message m4 are undone.

- When process P3 restarts from C3,8, it will resend message m4.
- Therefore, P4 should not replay message m4 from its log.
- If P4 replays message m4, then message m4 is called a duplicate message.

4.3 Issues in failure recovery

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



The computation comprises of three processes Pi, Pj , and Pk, connected through a communication network. The processes communicate solely by exchanging messages over fault- free, FIFO communication channels.

Processes Pi, Pj , and Pk have taken checkpoints

- Checkpoints : $\{C_{i,0}, C_{i,1}\}$, $\{C_{j,0}, C_{j,1}, C_{j,2}\}$, and $\{C_{k,0}, C_{k,1}, C_{k,2}\}$
- Messages : A - J
- The restored global consistent state : $\{C_{i,1}, C_{j,1}, C_{k,1}\}$

- The rollback of process Pi to checkpoint Ci,1 created an orphan message H
- Orphan message I is created due to the roll back of process Pj to checkpoint Cj,1
- Messages C, D, E, and F are potentially problematic
  - Message C: a delayed message
  - Message D: a lost message since the send event for D is recorded in the restored state for Pj, but the receive event has been undone at process Pi.
  - Lost messages can be handled by having processes keep a message log of all the sent messages
  - Messages E, F: delayed orphan messages. After resuming execution from their checkpoints, processes will generate both of these messages

4.4 Checkpoint-based recovery

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

1. U*ncoordinated checkpointing*
2. *Coordinated checkpointing*
3. *Communication-induced checkpointing*

1. Uncoordinated Checkpointing

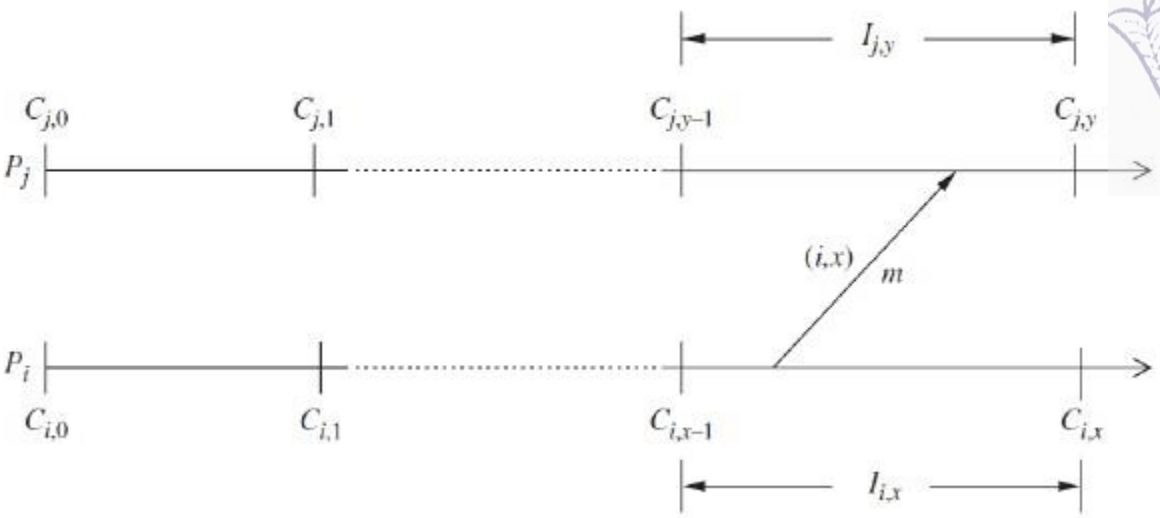
- Each process has autonomy in deciding when to take checkpoints
- Advantages

  The lower runtime overhead during normal execution

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

Direct dependency tracking technique

- Assume each process Pi starts its execution with an initial checkpoint $Ci,0$
- Ii,x : checkpoint interval, interval between $Ci,x-1$ and Ci,x
- When Pj receives a message m during Ij,y , it records the dependency from Ii,x to Ij,y, which is later saved onto stable storage when Pj takes Cj,y



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

2. Coordinated Checkpointing

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

Types

1. Blocking Checkpointing: After a process takes a local checkpoint, to prevent orphan messages, it remains blocked until the entire checkpointing activity is complete

   Disadvantages: The computation is blocked during the checkpointing
2. Non-blocking Checkpointing: The processes need not stop their execution while taking checkpoints. A fundamental problem in coordinated checkpointing is to prevent a process from receiving application messages that could make the checkpoint inconsistent.

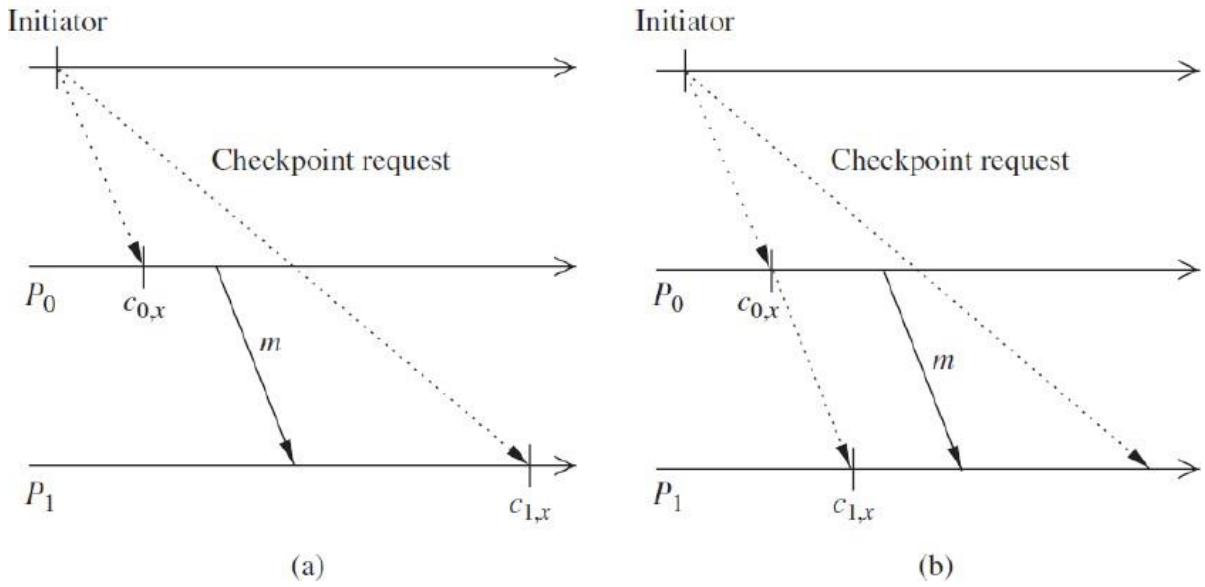
Example (a) : Checkpoint inconsistency

- Message m is sent by $P0$ after receiving a checkpoint request from the checkpoint coordinator
- Assume m reaches $P1$ before the checkpoint request
- This situation results in an inconsistent checkpoint since checkpoint $C1,x$ shows the receipt of message m from $P0$, while checkpoint $C0,x$ does not show m being sent from $P0$

Example (b) : A solution with FIFO channels

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

Coordinated Checkpointing



Impossibility of min-process non-blocking checkpointing

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

Algorithm

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

3. Communication-induced Checkpointing

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

Two types of checkpoints

1. Autonomous checkpoints
2. Forced checkpoints

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

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

Two types of communication-induced checkpointing

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

Model-based checkpointing

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

Index-based checkpointing.

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

4.5 Log-based rollback recovery

A log-based rollback recovery makes use of deterministic and nondeterministic events in a computation.

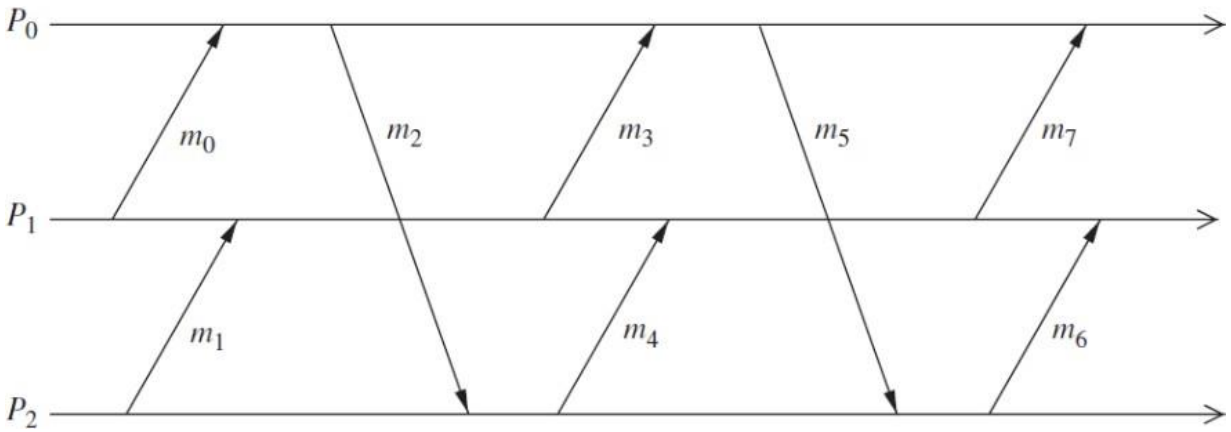
Deterministic and non-deterministic events

- Log-based rollback recovery exploits the fact that a process execution can be modeled as a sequence of deterministic state intervals, each starting with the execution of a non-deterministic event.
- A non-deterministic event can be the receipt of a message from another process or an event internal to the process.
- Note that a message send event is *not* a non-deterministic event.
- For example, in Figure, the execution of process *P*0 is a sequence of four deterministic intervals. The first one starts with the creation of the process, while the remaining three start with the receipt of messages *m*0, *m*3, and *m*7, respectively.
- Send event of message *m*2 is uniquely determined by the initial state of *P*0 and by the receipt of message *m*0, and is therefore not a non-deterministic event.
- Log-based rollback recovery assumes that all non-deterministic events can be identified and their corresponding determinants can be logged into the stable storage.
- Determinant: the information need to "replay" the occurrence of a non-deterministic

event (e.g., message reception).

- During failure-free operation, each process logs the determinants of all non-deterministic events that it observes onto the stable storage. Additionally, each process also takes checkpoints to reduce the extent of rollback during recovery.

Log-based Rollback Recovery



The no-orphans consistency condition

Let *e* be a non-deterministic event that occurs at process *p*. We define the following:

• *Depend*(*e*): the set of processes that are affected by a non-deterministic event *e.*

• *Log*(*e*): the set of processes that have logged a copy of *e*'s determinant in their volatile memory.

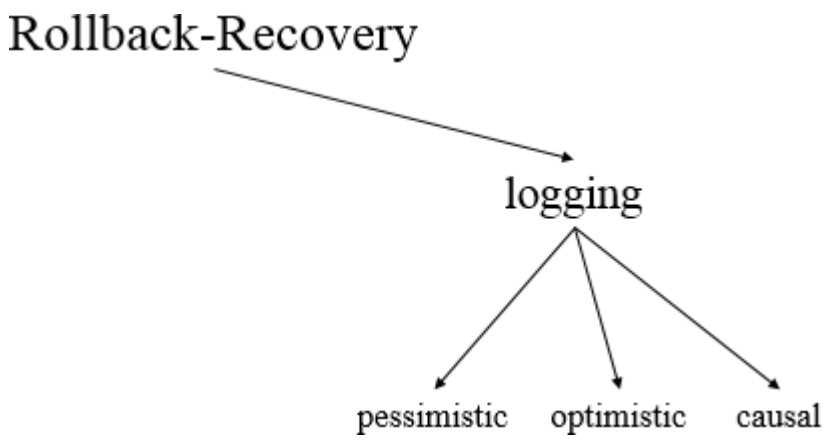
• *Stable*(*e*): a predicate that is true if *e*'s determinant is logged on the stable storage.

Suppose a set of processes Ψ crashes. A process *p* in Ψ becomes an orphan when *p* itself does not fail and *p*'s state depends on the execution of a nondeterministic event *e* whose determinant cannot be recovered from the stable storage or from the volatile memory of a surviving process. storage or from the volatile memory of a surviving process. Formally, it can be stated as follows

always-no-orphans condition

– $\forall(e) : \neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$

Types

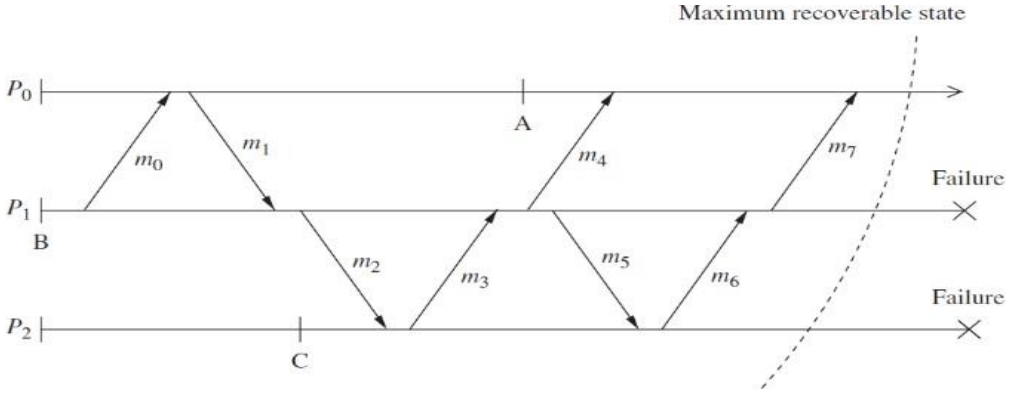


1. Pessimistic Logging

- Pessimistic logging protocols assume that a failure can occur after any non-deterministic event in the computation. However, in reality failures are rare
- Pessimistic protocols implement the following property, often referred to as *synchronous logging*, which is a stronger than the always-no-orphans condition
- *Synchronous logging*
  - $\forall e: \neg \text{Stable}(e) \Rightarrow |\text{Depend}(e)| = 0$
- Thai is,if an event has not been logged on the stable storage, then no process can depend on it.

Example:

- Suppose processes $P1$ and $P2$ fail as shown, restart from checkpoints B and C, and roll forward using their determinant logs to deliver again the same sequence of messages as in the pre-failure execution
- Once the recovery is complete, both processes will be consistent with the state of $P0$ that includes the receipt of message $m7$ from $P1$



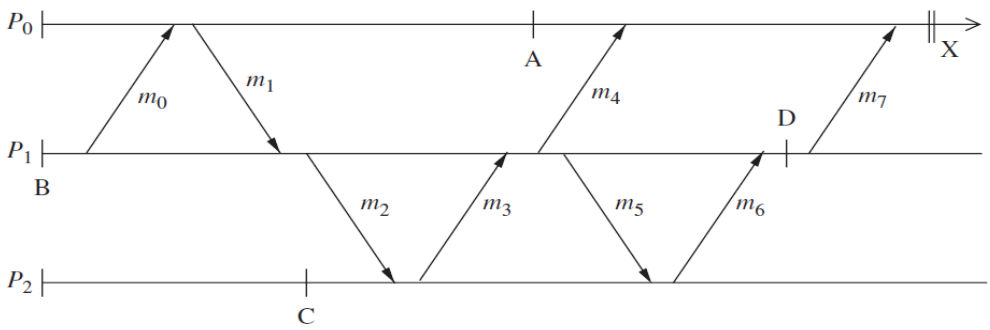
- Disadvantage: performance penalty for synchronous logging
- Advantages:
  - immediate output commit
  - restart from most recent checkpoint
  - recovery limited to failed process(es)
  - simple garbage collection
- Some pessimistic logging systems reduce the overhead of synchronous logging without relying on hardware. For example, the *sender-based message logging* (SBML) protocol keeps the determinants corresponding to the delivery of each message *m* in the volatile memory of its sender.
- The ***sender-based message logging* (SBML) protocol**

  Two steps.

  1. First, before sending m, the sender logs its content in volatile memory.
  2. Then, when the receiver of m responds with an acknowledgment that includes the order in which the message was delivered, the sender adds to the determinant the ordering information.

2. Optimistic Logging

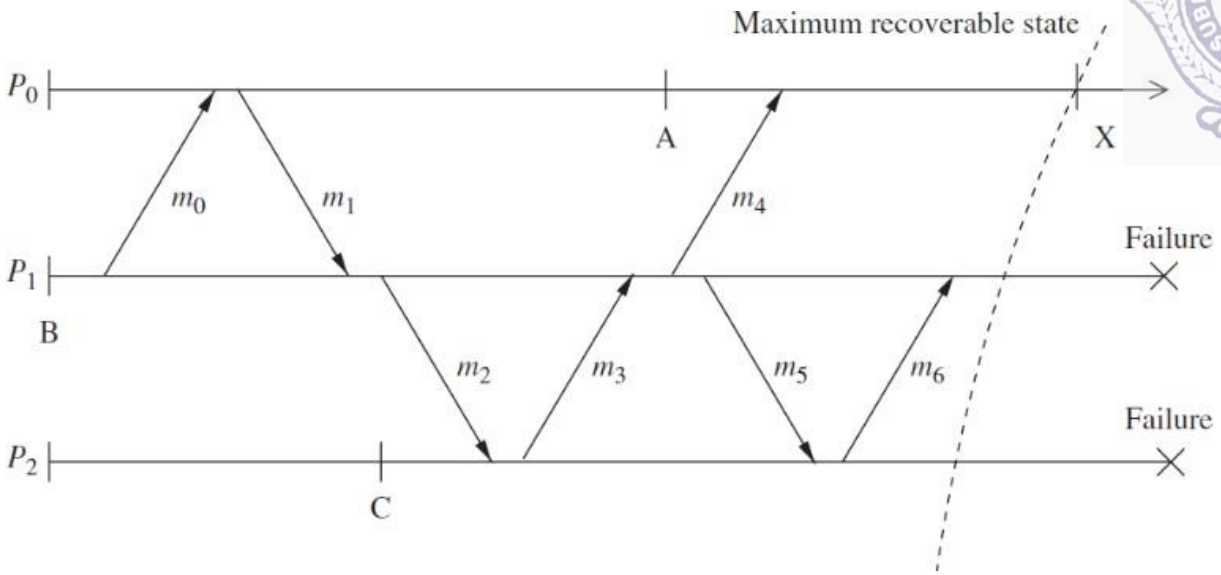
- Processes log determinants asynchronously to the stable storage
- Optimistically assume that logging will be complete before a failure occurs
- Do not implement the *always-no-orphans* condition
- To perform rollbacks correctly, optimistic logging protocols track causal dependencies during failure free execution
- Optimistic logging protocols require a non-trivial garbage collection scheme
- Pessimistic protocols need only keep the most recent checkpoint of each process, whereas optimistic protocols may need to keep multiple checkpoints for each process



- Consider the example shown in Figure Suppose process *P*2 fails before the determinant for *m*5 is logged to the stable storage. Process *P*1 then becomes an orphan process and must roll back to undo the effects of receiving the orphan message *m*6. The rollback of *P*1 further forces *P*0 to roll back to undo the effects of receiving message *m*7.
- **Advantage: better performance in failure-free execution**
- **Disadvantages:**
  - **coordination required on output commit**
  - **more complex garbage collection**
- Since determinants are logged asynchronously, output commit in optimistic logging protocols requires a guarantee that no failure scenario can revoke the output. For example, if process *P*0 needs to commit output at state X, it must log messages *m*4 and*m*7 to the stable storage and ask *P*2 to log *m*2 and *m*5. In this case, if any process fails, the computation can be reconstructed up to state X.

3. Causal Logging

- Combines the advantages of both pessimistic and optimistic logging at the expense of a more complex recovery protocol
- Like optimistic logging, it does not require synchronous access to the stable storage except during output commit
- Like pessimistic logging, it allows each process to commit output independently and never creates orphans, thus isolating processes from the effects of failures at other processes
- Make sure that the always-no-orphans property holds
- Each process maintains information about all the events that have causally affected its state



- Consider the example in Figure Messages *m*5 and *m*6 are likely to be lost on the failures of P1 and P2 at the indicated instants. Process
- *P*0 at state X will have logged the determinants of the nondeterministic events that causally precede its state according to Lamport's *happened-before* relation.
- These events consist of the delivery of messages *m*0, *m*1, *m*2, *m*3, and *m*4.
- The determinant of each of these non-deterministic events is either logged on the stable storage or is available in the volatile log of process *P*0.
- The determinant of each of these events contains the order in which its original receiver delivered the corresponding message.
- The message sender, as in sender-based message logging, logs the message content. Thus, process *P*0 will be able to "guide" the recovery of *P*1 and *P*2 since it knows the order in which *P*1 should replay messages *m*1 and *m*3 to reach the state from which *P*1 sent message *m*4.
- Similarly, *P*0 has the order in which *P*2 should replay message *m*2 to be consistent with both *P*0 and *P*1.
- The content of these messages is obtained from the sender log of *P*0 or regenerated deterministically during the recovery of *P*1 and *P*2.
- Note that information about messages *m*5 and *m*6 is lost due to failures. These messages may be resent after recovery possibly in a different order.
- However, since they did not causally affect the surviving process or the outside world, the

resulting state is consistent.

- Each process maintains information about all the events that have causally affected its state.

4.6 KOO AND TOUEG COORDINATED CHECKPOINTING AND RECOVERY TECHNIQUE:

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

A. The Checkpointing Algorithm

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

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

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

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

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

The algorithm consists of two phases.

First Phase

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

Second Phase

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

3. Either all or none of the processes advance the checkpoint by taking permanent checkpoints.
4. The algorithm requires that after a process has taken a tentative checkpoint, it cannot send messages related to the basic computation until it is informed of Pi's decision.

Correctness: for two reasons

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

An Optimization

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

B. The Rollback Recovery Algorithm

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

First Phase

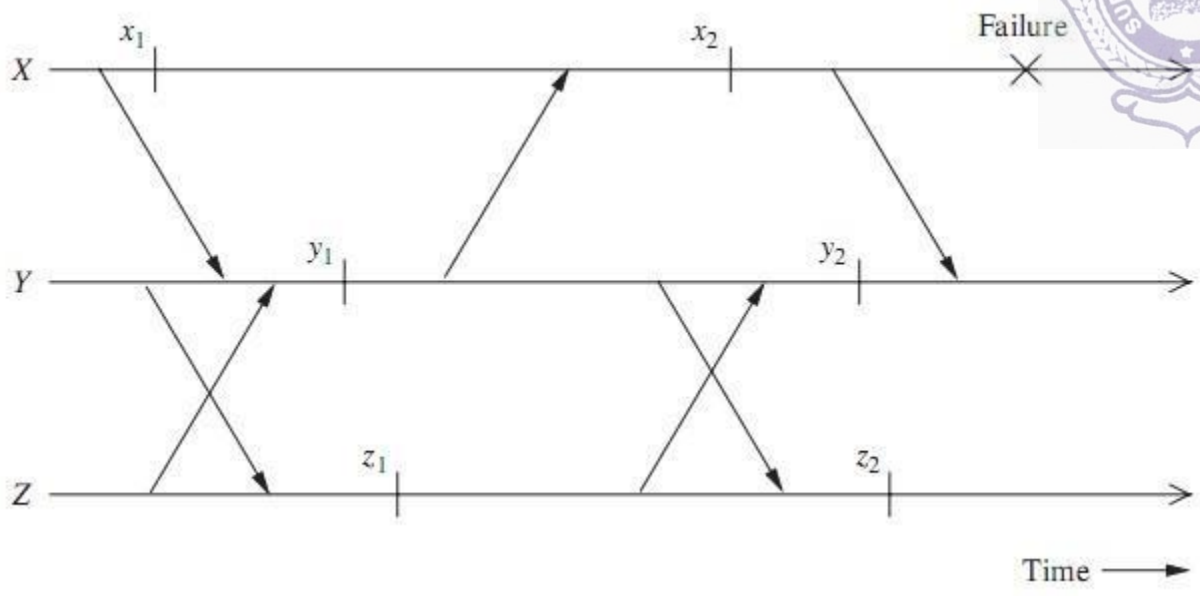
1. An initiating process Pi sends a message to all other processes to check if they all are willing to restart from their previous checkpoints.
2. A process may reply "no" to a restart request due to any reason (e.g., it is already participating in a check pointing or a recovery process initiated by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

Second Phase

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

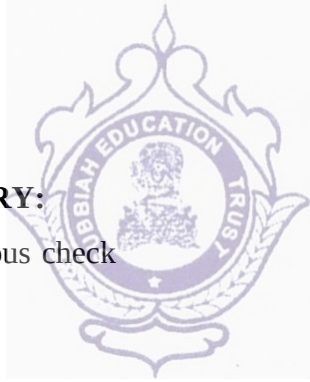
Correctness: Resume from a consistent state

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



The above protocol, in the event of failure of process X, the above protocol will require processes X, Y, and Z to restart from checkpoints x2, y2, and z2, respectively.

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



4.7 ALGORITHM FOR ASYNCHRONOUS CHECKPOINTING AND RECOVERY:

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

A. System Model and Assumptions

The algorithm makes the following assumptions about the underlying system:

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

Two type of log storage are maintained:

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

A. Asynchronous Check pointing

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

B. The Recovery Algorithm

Notations and data structure

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

• RCVDi←j(CkPti) represents the number of messages received by processor pi from processor pj , from the beginning of the computation till the checkpoint CkPti.

• SENTi→j(CkPti) represents the number of messages sent by processor pi to processor pj , from the beginning of the computation till the checkpoint CkPti.

Basic idea

- Since the algorithm is based on asynchronous check pointing, the main issue in the recovery is to find a consistent set of checkpoints to which the system can be restored.
- The recovery algorithm achieves this by making each processor keep track of both the

number of messages it has sent to other processors as well as the number of messages it has received from other processors.

- Whenever a processor rolls back, it is necessary for all other processors to find out if any message has become an orphan message. Orphan messages are discovered by comparing the number of messages sent to and received from neighboring processors.

For example, if RCVDi←j(CkPti) > SENTj→i(CkPtj) (that is, the number of messages received by processor pi from processor pj is greater than the number of messages sent by processor pj to processor pi, according to the current states the processors), then one or more messages at processor pj are orphan messages.

The Algorithm

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

Procedure RollBack_Recovery

processor pi executes the following:

STEP (a)

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

CkPti := latest event logged in the stable storage

else

CkPti := latest event that took place in pi {The latest event at pi can be either in stable or in volatile storage.}

end if

STEP (b)

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

for each neighboring processor pj **do**

compute SENTi→j(CkPti)

send a ROLLBACK(i, SENTi→j(CkPti)) message to pj

end for

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

if RCVDi←j(CkPti) > c {Implies the presence of orphan messages} **then**

find the latest event e such that RCVDi←j(e) = c {Such an event e may be in the volatile storage or stable storage.}

CkPti := e

end if

end for

end for{for k}

D. An Example

Consider an example shown in Figure 2 consisting of three processors. Suppose processor Y fails and restarts. If event ey2 is the latest checkpointed event at Y, then Y will restart from the state corresponding to ey2.

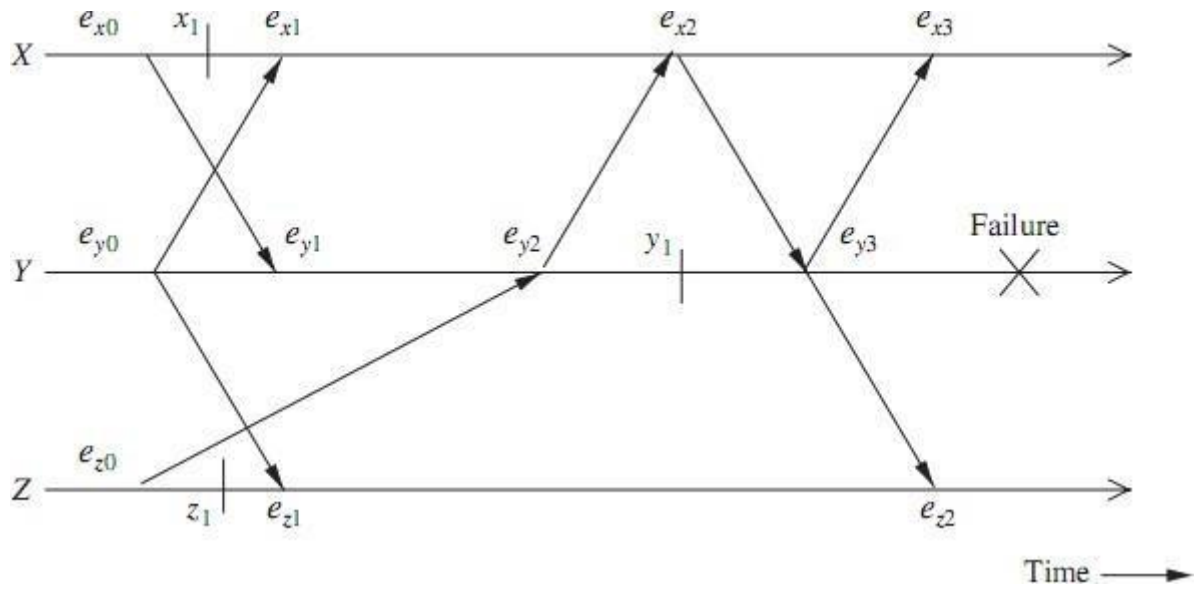


Figure 2: An example of Juan-Venkatesan algorithm.

- Because of the broadcast nature of ROLLBACK messages, the recovery algorithm is initiated at processors X and Z.
- Initially, X, Y, and Z set CkPtX ← ex3, CkPtY ← ey2 and CkPtZ ← ez2, respectively, and X, Y, and Z send the following messages during the first iteration:
- Y sends ROLLBACK(Y,2) to X and ROLLBACK(Y,1) to Z;
- X sends ROLLBACK(X,2) to Y and ROLLBACK(X,0) to Z;
- Z sends ROLLBACK(Z,0) to X and ROLLBACK(Z,1) to Y.

Since RCVDX←Y (CkPtX) = 3 > 2 (2 is the value received in the ROLLBACK(Y,2) message from Y), X will set CkPtX to ex2 satisfying RCVDX←Y (ex2) = 1≤ 2.

Since RCVDZ←Y (CkPtZ) = 2 > 1, Z will set CkPtZ to ez1 satisfying RCVDZ←Y (ez1) = 1 ≤ 1.

At Y, RCVDY←X(CkPtY) = 1 < 2 and RCVDY←Z(CkPtY) = 1 = SENTZ←Y (CkPtZ).

Y need not roll back further.

In the second iteration, Y sends ROLLBACK(Y,2) to X and ROLLBACK(Y,1) to Z;

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

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

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

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

Definition of Cloud Computing

Cloud computing is on-demand access, via the internet, to computing resources applications, servers (physical servers and virtual servers), data storage, development tools, networking capabilities, and more—hosted at a remote data center managed by a cloud services provider (or CSP). The CSP makes these resources available for a monthly subscription fee or bills them according to usage.

Cloud computing is a virtualization-based technology that allows us to create, configure, and customize applications via an internet connection. The cloud technology includes a development platform, hard disk, software application, and database.

The term cloud refers to a network or the internet. It is a technology that uses remote servers on the internet to store, manage, and access data online rather than local drives. The data can be anything such as files, images, documents, audio, video, and more.

Cloud Computing is defined as storing and accessing of data and computing services over the internet. It doesn't store any data on your personal computer. It is the on-demand availability of computer services like servers, data storage, networking, databases, etc. The main purpose of cloud computing is to give access to data centers to many users. Users can also access data from a remote server.

Cloud computing decreases the hardware and software demand from the user's side. The only thing that user must be able to run is the cloud computing systems interface software, which can be as simple as Web browser, and the Cloud network takes care of the rest. We all have experienced cloud computing at some instant of time, some of the popular cloud services we have used or we are still using are mail services like gmail, hotmail or yahoo etc.

Examples of Cloud Computing Services: AWS, Azure,

Characteristics of Cloud

The characteristics of cloud computing are given below:

1) Agility

The cloud works in a distributed computing environment. It shares resources among users and works very fast.

2) High availability and reliability

The availability of servers is high and more reliable because the chances of infrastructure failure are minimum.

3) High Scalability

Cloud offers "on-demand" provisioning of resources on a large scale, without having engineers for peak loads.

4) Multi-Sharing

With the help of cloud computing, multiple users and applications can work more efficiently with cost reductions by sharing common infrastructure.

5) Device and Location Independence

Cloud computing enables the users to access systems using a web browser regardless of their location or what device they use e.g. PC, mobile phone, etc. As infrastructure is off-site (typically provided by a third-party) and accessed via the Internet, users can connect from anywhere.

6) Maintenance

Maintenance of cloud computing applications is easier, since they do not need to be installed on each user's computer and can be accessed from different places. So, it reduces the cost also.

7) Low Cost

By using cloud computing, the cost will be reduced because to take the services of cloud computing, IT company need not to set its own infrastructure and pay-as-per usage of resources.

8) Services in the pay-per-use mode

Application Programming Interfaces (APIs) are provided to the users so that they can access services on the cloud by using these APIs and pay the charges as per the usage of services.

Cloud Deployment Models

The cloud deployment model identifies the specific type of cloud environment based on ownership, scale, access, and the cloud's nature and purpose. There are various deployment models are based on the location and who manages the infrastructure.

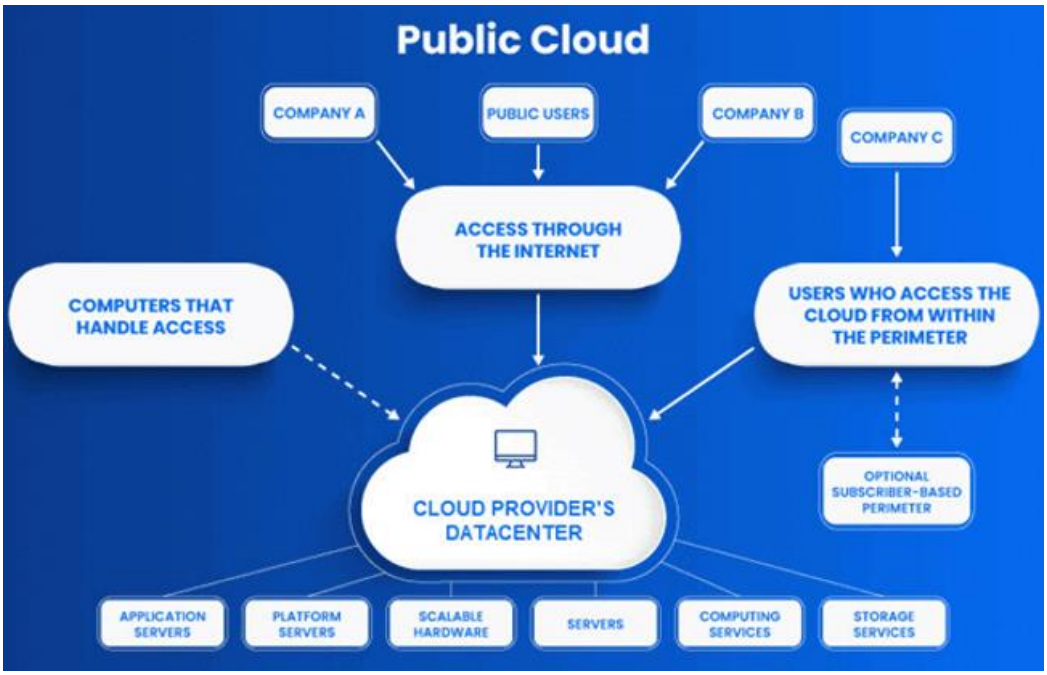
Type of Cloud Deployment Model

Here are some important types of Cloud Deployment models:

- **Private Cloud:** Resource managed and used by the organization.
- **Public Cloud:** Resource available for the general public under the Pay as you go model.
- **Community Cloud:** Resource shared by several organizations, usually in the same industry.
- **Hybrid Cloud:** This cloud deployment model is partly managed by the service provided and partly by the organization.

Public Cloud

The public cloud is available to the general public, and resources are shared between all users. They are available to anyone, from anywhere, using the Internet. The public cloud deployment model is one of the most popular types of cloud.



This computing model is hosted at the vendor's data center. The public cloud model makes the resources, such as storage and applications, available to the public over the WWW. It serves all the requests; therefore, resources are almost infinite.

Characteristics of Public Cloud

Here are the essential characteristics of the Public Cloud:

- Uniformly designed Infrastructure
- Works on the Pay-as-you-go basis
- Economies of scale
- SLA guarantees that all users have a fair share with no priority
- It is a multitenancy architecture, so data is highly likely to be leaked

Advantages of Public Cloud Deployments

Here are the pros/benefits of the Public Cloud Deployment Model:

- Highly available anytime and anywhere, with robust permission and authentication mechanism.
- There is no need to maintain the cloud.
- Does not have any limit on the number of users.
- The cloud service providers fully subsidize the entire Infrastructure. Therefore, you don't need to set up any hardware.
- Does not cost you any maintenance charges as the service provider does it.
- It works on the Pay as You Go model, so you don't have to pay for items you don't use.
- There is no significant upfront fee, making it excellent for enterprises that require immediate access to resources.

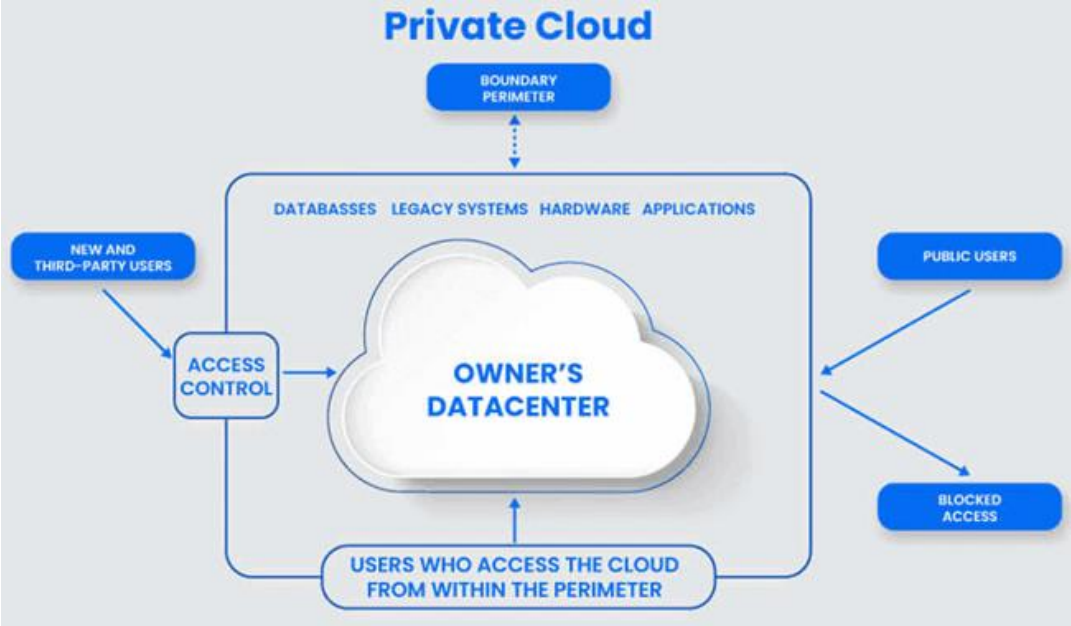
Disadvantages of Public Cloud Deployments

Here are the cons/drawbacks of the Public Cloud Deployment Model:

- It has lots of issues related to security.
- Privacy and organizational autonomy are not possible.
- You don't control the systems hosting your business applications.

Private Cloud Model

The private cloud deployment model is a dedicated environment for one user or customer. You don't share the hardware with any other users, as all the hardware is yours. It is a one-to-one environment for single use, so there is no need to share your hardware with anyone else. The main difference between private and public cloud deployment models is how you handle the hardware. It is also referred to as "internal cloud," which refers to the ability to access systems and services within an organization or border.



Characteristics of Private Cloud

Here are the essential characteristics of the Private Cloud:

- It has a non-uniformly designed infrastructure.
- Very low risk of data leaks.
- Provides End-to-End Control.
- Weak SLA, but you can apply custom policies.
- Internal Infrastructure to manage resources easily.

Advantages of Private Cloud Deployments

Here are the pros/benefits of the Private Cloud Deployment Model:

- You have complete command over service integration, IT operations, policies, and user behavior.
- Companies can customize their solution according to market demands.
- It offers exceptional reliability in performance.
- A private cloud enables the company to tailor its solution to meet specific needs.
- It provides higher control over system configuration according to the company's requirements.
- Private cloud works with legacy systems that cannot access the public cloud.
- This Cloud Computing Model is small, and therefore it is easy to manage.
- It is suitable for storing corporate information that only permitted staff can access.
- You can incorporate as many security services as possible to secure your cloud.

Disadvantages of Private Cloud Deployments

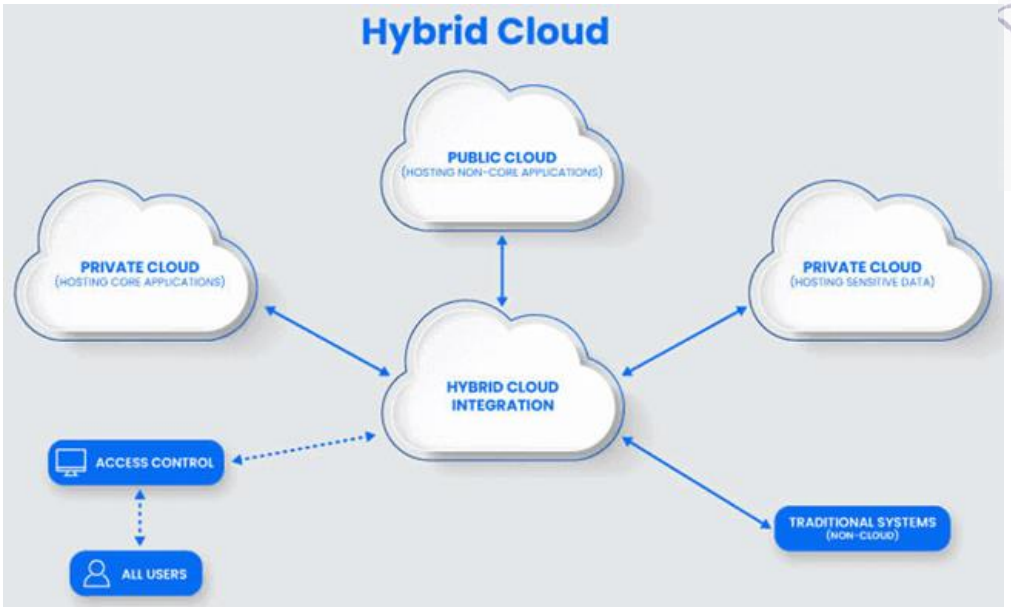
Here are the cons/drawbacks of the Private Cloud Deployment Model:

- It is a fully on-premises-hosted cloud that requires significant capital to purchase and maintain the necessary hardware.
- Companies that want extra computing power must take extra time and money to scale up their Infrastructure.
- Scalability depends on the choice of hardware.

Hybrid Cloud Model

A hybrid cloud deployment model combines public and private clouds. Creating a hybrid cloud computing model means that a company uses the public cloud but owns on-premises systems and provides a connection between the two. They work as one system, which is a beneficial model for a smooth transition into the public cloud over an extended period.

Some companies cannot operate solely in the public cloud because of security concerns or data protection requirements. So, they may select the hybrid cloud to combine the requirements with the benefits of a public cloud. It enables on-premises applications with sensitive data to run alongside public cloud applications.



Characteristics of Hybrid Cloud

Here are the Characteristics of the Hybrid Cloud:

- Provides betters security and privacy
- Offers improved scalability
- Cost-effective Cloud Deployment Model
- Simplifies data and application portability

Advantages of Hybrid Cloud Deployments

Here are the pros/benefits of the Hybrid Cloud Deployment Model:

- It gives the power of both public and private clouds.
- It offers better security than the Public Cloud.
- Public clouds provide scalability. Therefore, you can only pay for the extra capacity if required.
- It enables businesses to be more flexible and to design personalized solutions that meet their particular needs.
- Data is separated correctly, so the chances of data theft by attackers are considerably reduced.
- It provides robust setup flexibility so that customers can customize their solutions to fit their requirements.

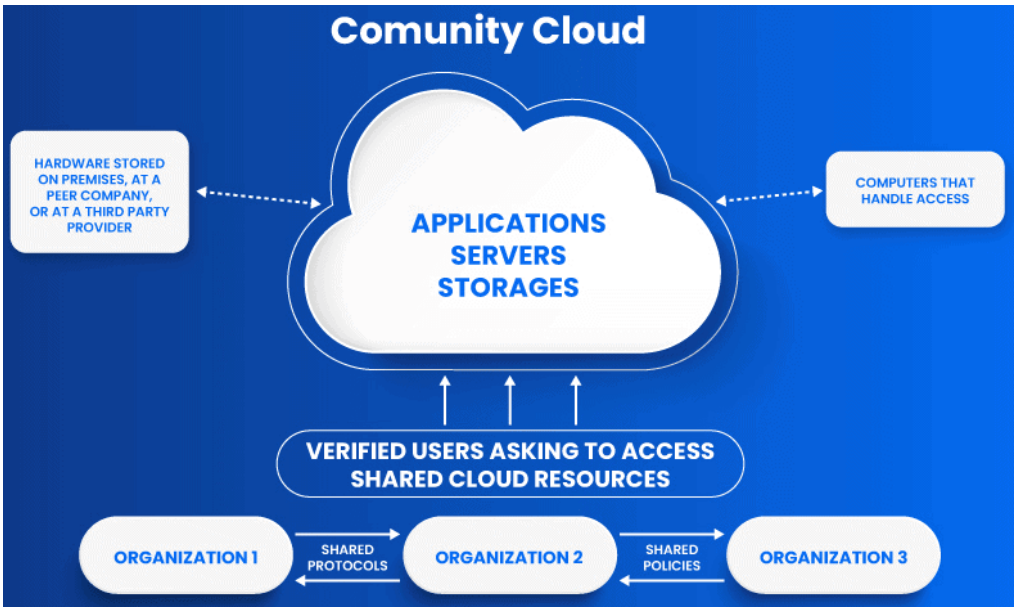
Disadvantages of Hybrid Cloud Deployments

Here are the cons/drawbacks of the Hybrid Cloud Deployment Model:

- It is applicable only when a company has varied use or demand for managing the workloads.
- Managing a hybrid cloud is complex, so if you use a hybrid cloud, you may spend too much.
- Its security features are not good as the Private Cloud.

Community Cloud Model

Community clouds are cloud-based infrastructure models that enable multiple organizations to share resources and services based on standard regulatory requirements. It provides a shared platform and resources for organizations to work on their business requirements. This Cloud Computing model is operated and managed by community members, third-party vendors, or both. The organizations that share standard business requirements make up the members of the community cloud.



Advantages of Community Cloud Deployments

Here are the pros/benefits of the Community Cloud Deployment Model:

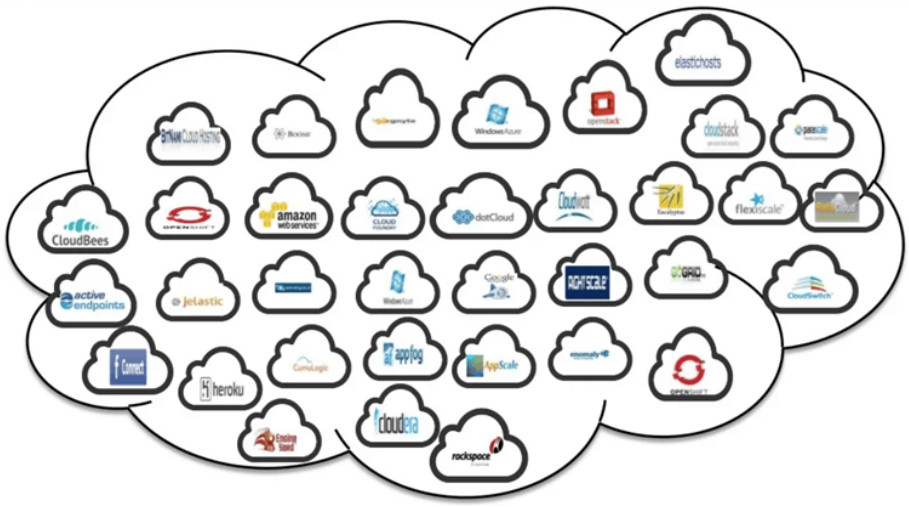
- You can establish a low-cost private cloud.
- It helps you to do collaborative work on the cloud.
- It is cost-effective, as multiple organizations or communities share the cloud.
- You can share resources, Infrastructure, etc., with multiple organizations.
- It is a suitable model for both collaboration and data sharing.
- Gives better security than the public cloud.
- It offers a collaborative space that allows clients to enhance their efficiency.

Disadvantages of Community Cloud Deployments

Here are the cons/drawbacks of the Community Cloud Deployment Model:

- Because of its restricted bandwidth and storage capacity, community resources often pose challenges.
- It is not a very popular and widely adopted cloud computing model.
- Security and segmentation are challenging to maintain.

Multi-cloud Model



Multi-cloud computing refers to using public cloud services from many cloud service providers. A company must run workloads on IaaS or PaaS in a multi-cloud configuration from multiple vendors, such as Azure, AWS, or Google Cloud Platform.

There are many reasons an organization selects a multi-cloud strategy. Some use it to avoid vendor lock-in problems, while others combat shadow IT through multi-cloud

deployments. So, employees can still benefit from a specific public cloud service if it does not meet strict IT policies.

Benefits of Multi-Cloud Deployment Model

Here are the pros/benefits of the Multi-Cloud Deployment Model:

- A multi-cloud deployment model helps organizations choose the specific services that work best for them.
- It provides a reliable architecture.
- With multi-cloud models, companies can choose the best Cloud service provider based on contract options, flexibility with payments, and customizability of capacity.
- It allows you to select cloud regions and zones close to your clients.

Disadvantages of Multi-Cloud Deployments

Here are the cons/drawbacks of the Multi-Cloud Deployment Model:

- Multi-cloud adoption increases the complexity of your business.
- Finding developers, engineers, and cloud security experts who know multiple clouds is difficult.
- **Comparison of Top Cloud Deployment Models**

Parameters	**Public**	**Private**	**Community**	**Hybrid**
Setup and use	Easy	Need help from a professional IT team.	Require a professional IT team.	Require a profession al IT team.
Scalability and Elasticity	Very High	Low	Moderate	High
Data Control	Little to none	Very High	Relatively High	High
Security and privacy	Very low	Very high	High	Very high
Reliability	Low	High	Higher	High
Demand for in-house software	No	Very high in-house software requirement	No	In-house software is not a must.

How to select the suitable Cloud Deployment Models

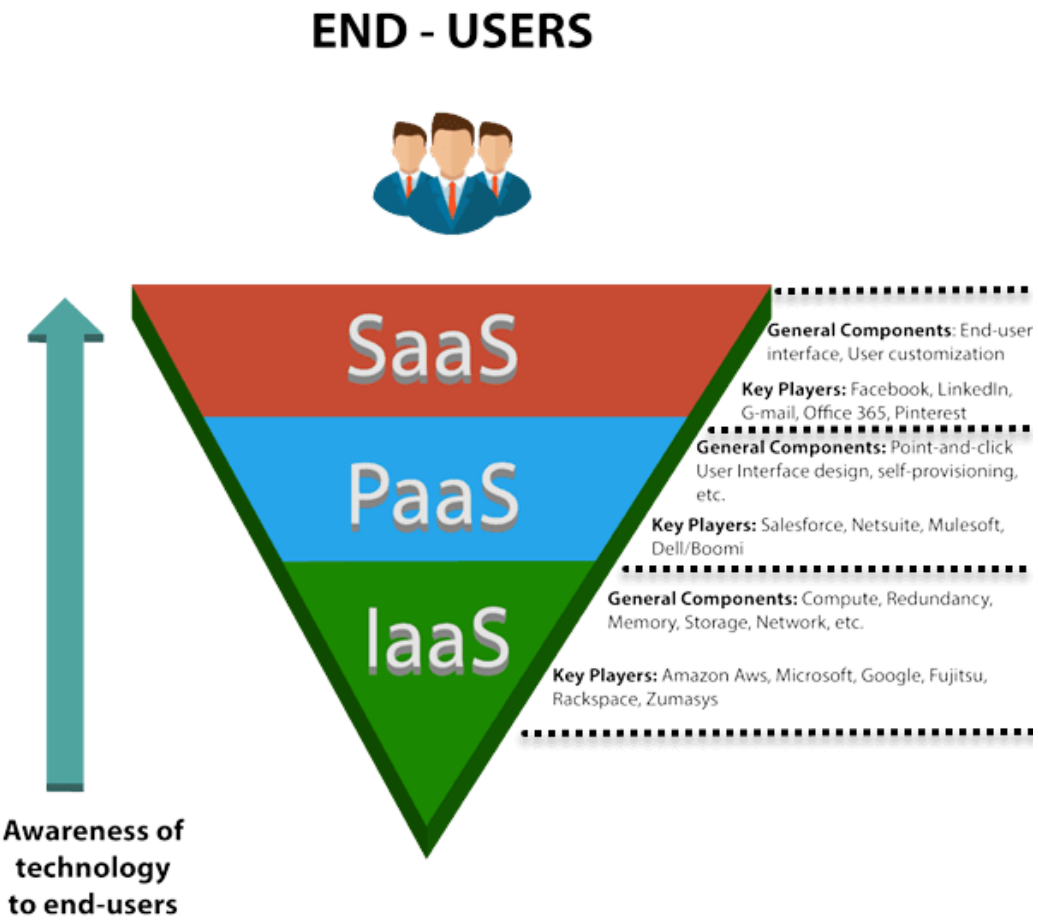
Companies are extensively using these cloud computing models all around the world. Each of them solves a specific set of problems. So, finding the right Cloud Deployment Model for you or your company is important.

Here are points you should remember for selecting the right Cloud Deployment Model:

- **Scalability:** You need to check if your user activity is growing quickly or unpredictably with spikes in demand.
- **Privacy and security:** Select a service provider that protects your privacy and the security of your sensitive data.
- **Cost:** You must decide how many resources you need for your cloud solution. Then calculate the approximate monthly cost for those resources with different cloud providers**.**
- **Ease of use:** You must select a model with no steep learning curve.
- **Legal Compliance:** You need to check whether any relevant low stop you from selecting any specific cloud deployment model.

Cloud Service Models

SaaS, PaaS, and IaaS are the three main cloud computing service model categories. You can access all three via an Internet browser or online apps available on different devices. The cloud service model enables the team to collaborate online instead of offline creation and then share online.



Software as a Service (SaaS)

Software as a Service (SaaS) is a web-based deployment model that makes the software accessible through a web browser. SaaS software users don't need to care where the software is hosted, which operating system it uses, or even which programming language it is written in. The SaaS software is accessible from any device with an internet connection.

This cloud service model ensures that consumers always use the most current version of the software. The SaaS provider handles maintenance and support. In the SaaS model, users don't control the infrastructure, such as storage, processing power, etc.

Characteristics of SaaS

There are the following characteristics of SaaS:

- It is managed from a central location.
- Hosted directly on a remote server.
- It is accessible over the Internet.
- SaaS users are not responsible for hardware and software updates.
- The services are purchased on a pay-as-per-use basis.

Advantages SaaS

Here are the important advantages/pros of SaaS:

- The biggest benefit of using SaaS is that it is easy to set up, so you can start using it instantly.
- Compared with on-premises software, it is more cost-effective.
- You don’t need to manage or upgrade the software, as it is typically included in a SaaS subscription or purchase.
- It won’t use your local resources, such as the hard disk typically required to install desktop software.
- It is a cloud computing service category that provides a wide range of hosted capabilities and services.
- Developers can easily build and deploy web-based software applications.
- You can easily access it through a browser.

Disadvantages SaaS

Here are the important cons/drawbacks of SaaS:

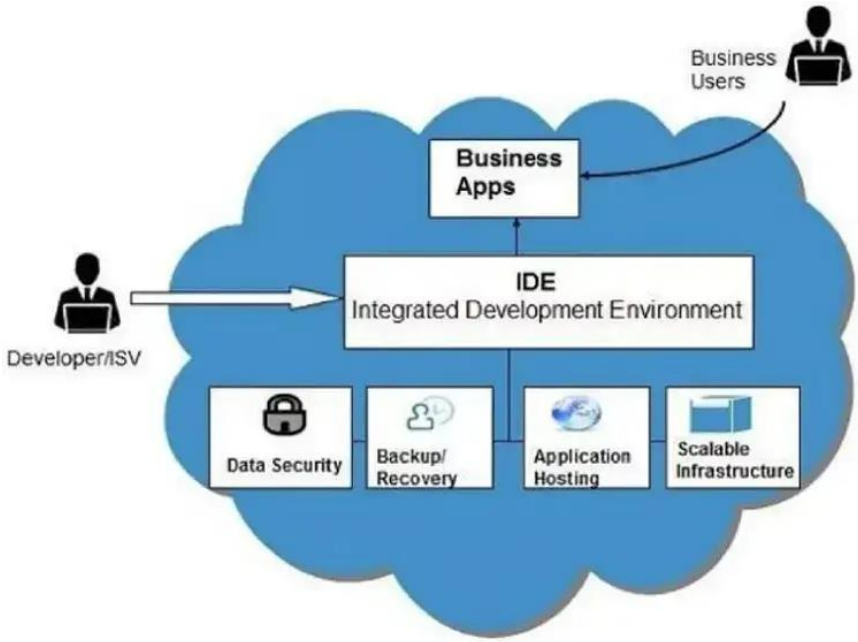
- Integrations are up to the provider, so it's impossible to "patch" an integration on your end.
- SaaS tools may become incompatible with other tools and hardware already used in your business.
- You depend on the SaaS company's security measures, so your data may be compromised if any leaks occur.

Consider Before SaaS Implementation

Need to consider before SaaS implementation:

- It would help if you opted for configuration over customization within a SaaS-based delivery model.
- You must carefully understand the usage rates and set clear objectives to achieve the SaaS adoption.
- You can complement your SaaS solution with integrations and security options to make it more user-oriented.

Platform as a Service (PaaS)



Platform-as-a-Service (PaaS) provides a cloud computing framework for software application creation and deployment. It is a platform for the deployment and management of software apps. This flexible cloud computing model scales up automatically on demand. It also manages the servers, storage, and networking, while the developers manage only the application part. It offers a runtime environment for application development and deployment tools.

This Model provides all the facilities required to support the complex life cycle of building and delivering web applications and services entirely for the Internet. This cloud computing model enables developers to rapidly develop, run, and manage their apps without building and maintaining the infrastructure or platform.

Characteristics of PaaS

There are the following characteristics of PaaS:

- Builds on virtualization technology, so computing resources can easily be scaled up (Auto-scale) or down according to the organization's needs.
- Support multiple programming languages and frameworks.
- Integrates with web services and databases.

Advantages PaaS

Here are the important benefits/pros of PaaS:

- Simple, cost-effective development and deployment of apps
- Developers can customize SaaS apps without the headache of maintaining the software
- Provide automation of Business Policy
- Easy migration to the Hybrid Model
- It allows developers to build applications without the overhead of the underlying operating system or cloud infrastructure
- Offers freedom to developers to focus on the application's design while the platform takes care of the language and the database
- It helps developers to collaborate with other developers on a single app

Disadvantages of SaaS

Here are the important cons/drawbacks of PaaS:

- You have control over the app's code and not its infrastructure.
- The PaaS organization stores your data, so it sometimes poses a security risk to your app's users.
- Vendors provide varying service levels, so selecting the right services is essential.
- The risk of lock-in with a vendor may affect the ecosystem you need for your development environment.

Consider Before PaaS Implementation

Here are essential things you need to consider before PaaS implementation:

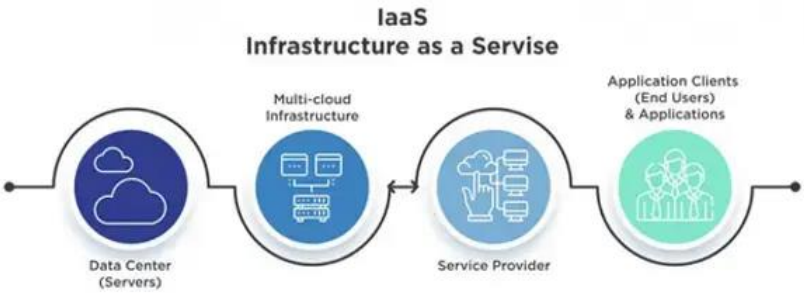
- Analyze your business needs, decide the automation levels, and also decides whether you want a self-service or fully automated PaaS model.
- You need to determine whether to deploy on a private or public cloud.
- Plan through the customization and efficiency levels.

Infrastructure as a Service (IaaS)

Infrastructure-as-a-Service (IaaS) is a cloud computing service offering on-demand computing, storage, and networking resources. It usually works on a pay-as-you-go basis.

Organizations can purchase resources on-demand and as needed instead of buying the hardware outright.

The IaaS cloud vendor hosts the infrastructure components, including the on-premises data center, servers, storage, networking hardware, and the hypervisor (virtualization layer).



This Model contains the basic building blocks for your web application. It provides complete control over the hardware that runs your application (storage, servers, VMs, networks & operating systems). IaaS model gives you the best flexibility and management control over your IT resources.

Characteristics of IaaS

There are the following characteristics of IaaS:

- Resources are available as a service
- Services are highly scalable
- Dynamic and flexible Cloud Service Model
- GUI and API-based access
- Automate the administrative tasks

Advantages of IaaS

Here are the important benefits/pros of PaaS:

- Easy to automate the deployment of storage, networking, and servers.
- Hardware purchases can be based on consumption.
- Clients keep complete control of their underlying infrastructure.
- The provider can deploy the resources to a customer's environment anytime.
- It can be scaled up or downsized according to your needs.

Disadvantages of IaaS

Here are the important Cons/drawbacks of IaaS:

- You should ensure that your apps and operating systems are working correctly and providing the utmost security.
- You're in charge of the data, so if any of it is lost, it's up to you to recover it.
- IaaS firms only provide the servers and API, so you must configure everything else.

Consider Before IaaS Implementation

Here are some specific considerations you should remember before IaaS Implementation:

- You should clearly define your access needs and your network's bandwidth to facilitate smooth implementation and functioning.
- Plan out detailed data storage and security strategy to streamline the business process.
- Ensure that your organization has a proper disaster recovery plan to keep your data safe and accessible.

How can select the Best SaaS Service Provider

Here are some essential criteria for selecting the best cloud service provider:

- **Financial stability**: Look for a well-financed cloud provider that has steady profits from the infrastructure. If the company shuts down because of monetary issues, your solutions will also be in jeopardy.
- **Industries that prefer the solution**: Before finalizing cloud services, examine its existing clients and markets. Your cloud service provider should be popular among companies in your niche or neighboring ones.
- **Datacenter locations**: To avoid safety risks, ensure that cloud providers enable your data's geographical distribution.
- **Encryption standards**: You should make sure the cloud provider supports major encryption algorithms.
- **Check accreditation and auditing**: The widely used online auditing standard is SSAE. This procedure helps you to verify the safety of online data storage. ISO 27001 certificate verifies that a cloud provider complies with international safety standards for data storage.
- **Backup**: The provider should support incremental backups so that you can store offsite and quickly restore.

Driving Factors and Challenges of Cloud

Data Security and Privacy

Data security is a major concern when switching to cloud computing. User or organizational data stored in the cloud is critical and private. Even if the cloud service provider

assures data integrity, it is your responsibility to carry out user authentication and authorization, identity management, data encryption, and access control. Security issues on the cloud include identity theft, data breaches, malware infections, and a lot more which eventually decrease the trust amongst the users of your applications. This can in turn lead to potential loss in revenue alongside reputation and stature. Also, dealing with cloud computing requires sending and receiving huge amounts of data at high speed, and therefore is susceptible to data leaks.

Cost Management

Even as almost all cloud service providers have a "Pay As You Go" model, which reduces the overall cost of the resources being used, there are times when there are huge costs incurred to the enterprise using cloud computing. When there is under optimization of the resources, let's say that the servers are not being used to their full potential, add up to the hidden costs. If there is a degraded application performance or sudden spikes or overages in the usage, it adds up to the overall cost. Unused resources are one of the other main reasons why the costs go up. If you turn on the services or an instance of cloud and forget to turn it off during the weekend or when there is no current use of it, it will increase the cost without even using the resources.

Multi-Cloud Environments

Due to an increase in the options available to the companies, enterprises not only use a single cloud but depend on multiple cloud service providers. Most of these companies use hybrid cloud tactics and close to 84% are dependent on multiple clouds. This often ends up being hindered and difficult to manage for the infrastructure team. The process most of the time ends up being highly complex for the IT team due to the differences between multiple cloud providers.

Performance Challenges

Performance is an important factor while considering cloud-based solutions. If the performance of the cloud is not satisfactory, it can drive away users and decrease profits. Even a little latency while loading an app or a web page can result in a huge drop in the percentage of users. This latency can be a product of inefficient load balancing, which means that the server cannot efficiently split the incoming traffic so as to provide the best user experience.

Challenges also arise in the case of fault tolerance, which means the operations continue as required even when one or more of the components fail.

Interoperability and Flexibility

When an organization uses a specific cloud service provider and wants to switch to another cloud-based solution, it often turns up to be a tedious procedure since applications written for one cloud with the application stack are required to be re-written for the other cloud. There is a lack of flexibility from switching from one cloud to another due to the complexities involved. Handling data movement, setting up the security from scratch and network also add up to the issues encountered when changing cloud solutions, thereby reducing flexibility.

High Dependence on Network

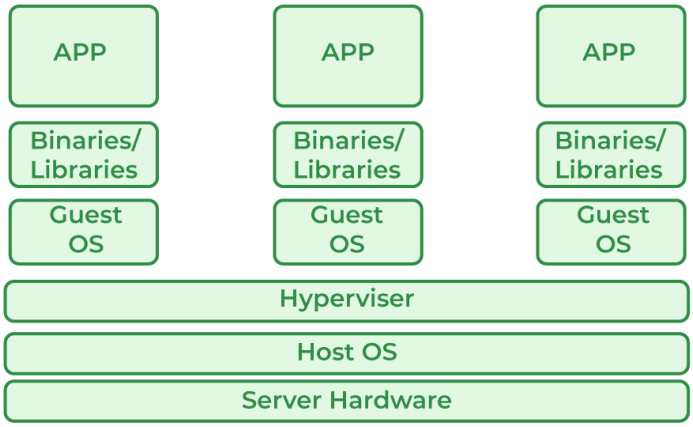
Since cloud computing deals with provisioning resources in real-time, it deals with enormous amounts of data transfer to and from the servers. This is only made possible due to the availability of the high-speed network. Although these data and resources are exchanged over the network, this can prove to be highly vulnerable in case of limited bandwidth or cases when there is a sudden outage. Even when the enterprises can cut their hardware costs, they need to ensure that the internet bandwidth is high as well there are zero network outages, or else it can result in a potential business loss. It is therefore a major challenge for smaller enterprises that have to maintain network bandwidth that comes with a high cost.

Lack of Knowledge and Expertise

Due to the complex nature and the high demand for research working with the cloud often ends up being a highly tedious task. It requires immense knowledge and wide expertise on the subject. Although there are a lot of professionals in the field they need to constantly update themselves. Cloud computing is a highly paid job due to the extensive gap between demand and supply. There are a lot of vacancies but very few talented cloud engineers, developers, and professionals. Therefore, there is a need for upskilling so these professionals can actively understand, manage and develop cloud-based applications with minimum issues and maximum reliability.

Virtualization

Virtualization is a technique how to separate a service from the underlying physical delivery of that service. It is the process of creating a virtual version of something like computer hardware. It was initially developed during the mainframe era. It involves using specialized software to create a virtual or software-created version of a computing resource rather than the actual version of the same resource. With the help of Virtualization, multiple operating systems and applications can run on the same machine and its same hardware at the same time, increasing the utilization and flexibility of hardware.



Virtualization

Host Machine: The machine on which the virtual machine is going to be built is known as Host Machine.
Guest Machine: The virtual machine is referred to as a Guest Machine.

Virtualization has a prominent impact on Cloud Computing. In the case of cloud computing, users store data in the cloud, but with the help of Virtualization, users have the extra benefit of sharing the infrastructure. Cloud Vendors take care of the required physical resources, but these cloud providers charge a huge amount for these services which impacts every user or organization. Virtualization helps Users or Organisations in maintaining those services which are required by a company through external (third-party) people, which helps in reducing costs to the company. This is the way through which Virtualization works in Cloud Computing.

Benefits of Virtualization

- More flexible and efficient allocation of resources.

- Enhance development productivity.
- It lowers the cost of IT infrastructure.
- Remote access and rapid scalability.
- High availability and disaster recovery.
- Pay peruse of the IT infrastructure on demand.
- Enables running multiple operating systems.

Drawback of Virtualization

- **High Initial Investment:** Clouds have a very high initial investment, but it is also true that it will help in reducing the cost of companies.
- **Learning New Infrastructure:** As the companies shifted from Servers to Cloud, it requires highly skilled staff who have skills to work with the cloud easily, and for this, you have to hire new staff or provide training to current staff.
- **Risk of Data:** Hosting data on third-party resources can lead to putting the data at risk, it has the chance of getting attacked by any hacker or cracker very easily.

Characteristics of Virtualization

- **Increased Security:** The ability to control the execution of a guest program in a completely transparent manner opens new possibilities for delivering a secure, controlled execution environment. All the operations of the guest programs are generally performed against the virtual machine, which then translates and applies them to the host programs.
- **Managed Execution**: In particular, sharing, aggregation, emulation, and isolation are the most relevant features.
- **Sharing:** Virtualization allows the creation of a separate computing environment within the same host.
- **Aggregation:** It is possible to share physical resources among several guests, but virtualization also allows aggregation, which is the opposite process.

Types of Virtualization

1. Application Virtualization
2. Network Virtualization
3. Desktop Virtualization

4. Storage Virtualization
5. Server Virtualization
6. Data virtualization

1. **Application Virtualization**:

Application virtualization helps a user to have remote access to an application from a server. The server stores all personal information and other characteristics of the application but can still run on a local workstation through the internet. An example of this would be a user who needs to run two different versions of the same software. Technologies that use application virtualization are hosted applications and packaged applications.

2. Network Virtualization:

The ability to run multiple virtual networks with each having a separate control and data plan. It co-exists together on top of one physical network. It can be managed by individual parties that are potentially confidential to each other. Network virtualization provides a facility to create and provision virtual networks, logical switches, routers, firewalls, load balancers, Virtual Private Networks (VPN), and workload security within days or even weeks.

3. Desktop Virtualization:

Desktop virtualization allows the users' OS to be remotely stored on a server in the data center. It allows the user to access their desktop virtually, from any location by a different machine. Users who want specific operating systems other than Windows Server will need to have a virtual desktop. The main benefits of desktop virtualization are user mobility, portability, and easy management of software installation, updates, and patches.

4. Storage Virtualization:

Storage virtualization is an array of servers that are managed by a virtual storage system. The servers aren't aware of exactly where their data is stored and instead function more like worker bees in a hive. It makes managing storage from multiple sources be managed and utilized as a single repository. storage virtualization software maintains smooth operations, consistent performance, and a continuous suite of advanced functions despite changes, breaks down, and differences in the underlying equipment.

5. Server Virtualization:

This is a kind of virtualization in which the masking of server resources takes place. Here, the central server (physical server) is divided into multiple different virtual servers by changing the identity number, and processors. So, each system can operate its operating systems in an isolated manner. Where each sub-server knows the identity of the central server. It causes an increase in performance and reduces the operating cost by the deployment of main server resources into a sub-server resource. It's beneficial in virtual migration, reducing energy consumption, reducing infrastructural costs, etc.

6. Data Virtualization:

This is the kind of virtualization in which the data is collected from various sources and managed at a single place without knowing more about the technical information like how data is collected, stored & formatted then arranged that data logically so that its virtual view can be accessed by its interested people and stakeholders, and users through the various cloud services remotely. Many big giant companies are providing their services like Oracle, IBM, At scale, Cdata, etc.

Load Balancing

Load balancing is the method that allows you to have a proper balance of the amount of work being done on different pieces of device or hardware equipment. Typically, what happens is that the load of the devices is balanced between different servers or between the CPU and hard drives in a single cloud server.

Load balancing was introduced for various reasons. One of them is to improve the speed and performance of each single device, and the other is to protect individual devices from hitting their limits by reducing their performance.

Cloud load balancing is defined as dividing workload and computing properties in cloud computing. It enables enterprises to manage workload demands or application demands by distributing resources among multiple computers, networks or servers. Cloud load balancing involves managing the movement of workload traffic and demands over the Internet.

Traffic on the Internet is growing rapidly, accounting for almost 100% of the current traffic annually. Therefore, the workload on the servers is increasing so rapidly, leading to

overloading of the servers, mainly for the popular web servers. There are two primary solutions to overcome the problem of overloading on the server-

First is a single-server solution in which the server is upgraded to a higher-performance server. However, the new server may also be overloaded soon, demanding another upgrade. Moreover, the upgrading process is arduous and expensive.

The second is a multiple-server solution in which a scalable service system on a cluster of servers is built. That's why it is more cost-effective and more scalable to build a server cluster system for network services.

Cloud-based servers can achieve more precise scalability and availability by using farm server load balancing. Load balancing is beneficial with almost any type of service, such as HTTP, SMTP, DNS, FTP, and POP/IMAP.

It also increases reliability through redundancy. A dedicated hardware device or program provides the balancing service.

Different Types of Load Balancing Algorithms in Cloud Computing:

1. Static Algorithm

Static algorithms are built for systems with very little variation in load. The entire traffic is divided equally between the servers in the static algorithm. This algorithm requires in-depth knowledge of server resources for better performance of the processor, which is determined at the beginning of the implementation.

However, the decision of load shifting does not depend on the current state of the system. One of the major drawbacks of static load balancing algorithm is that load balancing tasks work only after they have been created. It could not be implemented on other devices for load balancing.

2. Dynamic Algorithm

The dynamic algorithm first finds the lightest server in the entire network and gives it priority for load balancing. This requires real-time communication with the network which can help increase the system's traffic. Here, the current state of the system is used to control the load.

The characteristic of dynamic algorithms is to make load transfer decisions in the current system state. In this system, processes can move from a highly used machine to an underutilized machine in real time.

3. Round Robin Algorithm

Round robin load balancing algorithm uses round-robin method to assign jobs. First, it randomly selects the first node and assigns tasks to other nodes in a round-robin manner. This is one of the easiest methods of load balancing.

Processors assign each process circularly without defining any priority. It gives fast response in case of uniform workload distribution among the processes. All processes have different loading times. Therefore, some nodes may be heavily loaded, while others may remain under-utilised.

4**. Weighted Round Robin Load Balancing Algorithm**

Weighted Round Robin Load Balancing Algorithms have been developed to enhance the most challenging issues of Round Robin Algorithms. In this algorithm, there are a specified set of weights and functions, which are distributed according to the weight values.

Processors that have a higher capacity are given a higher value. Therefore, the highest loaded servers will get more tasks. When the full load level is reached, the servers will receive stable traffic.

5. Opportunistic Load Balancing Algorithm

The opportunistic load balancing algorithm allows each node to be busy. It never considers the current workload of each system. Regardless of the current workload on each node, OLB distributes all unfinished tasks to these nodes.

The processing task will be executed slowly as an OLB, and it does not count the implementation time of the node, which causes some bottlenecks even when some nodes are free.

6. Minimum to Minimum Load Balancing Algorithm

Under minimum to minimum load balancing algorithms, first of all, those tasks take minimum time to complete. Among them, the minimum value is selected among all the functions. According to that minimum time, the work on the machine is scheduled.

Other tasks are updated on the machine, and the task is removed from that list. This process will continue till the final assignment is given. This algorithm works best where many small tasks outweigh large tasks.

Load balancing solutions can be categorized into two types -

Software-based load balancers: Software-based load balancers run on standard hardware (desktop, PC) and standard operating systems.

Hardware-based load balancers: Hardware-based load balancers are dedicated boxes that contain application-specific integrated circuits (ASICs) optimized for a particular use. ASICs allow network traffic to be promoted at high speeds and are often used for transport-level load balancing because hardware-based load balancing is faster than a software solution.

Major Examples of Load Balancers

Direct Routing Request Despatch Technique: This method of request dispatch is similar to that implemented in IBM's NetDispatcher. A real server and load balancer share a virtual IP address. The load balancer takes an interface built with a virtual IP address that accepts request packets and routes the packets directly to the selected server.

Dispatcher-Based Load Balancing Cluster: A dispatcher performs smart load balancing using server availability, workload, capacity and other user-defined parameters to regulate where TCP/IP requests are sent. The dispatcher module of a load balancer can split HTTP requests among different nodes in a cluster. The dispatcher divides the load among multiple servers in a cluster, so services from different nodes act like a virtual service on only one IP address; Consumers interconnect as if it were a single server, without knowledge of the back-end infrastructure.

Linux Virtual Load Balancer: This is an open-source enhanced load balancing solution used to build highly scalable and highly available network services such as HTTP, POP3, FTP,

SMTP, media and caching, and Voice over Internet Protocol (VoIP) is done. It is a simple and powerful product designed for load balancing and fail-over. The load balancer itself is the primary entry point to the server cluster system. It can execute Internet Protocol Virtual Server (IPVS), which implements transport-layer load balancing in the Linux kernel, also known as layer-4 switching.

Types of Load Balancing

Network Load Balancing

Cloud load balancing takes advantage of network layer information and leaves it to decide where network traffic should be sent. This is accomplished through Layer 4 load balancing, which handles TCP/UDP traffic. It is the fastest local balancing solution, but it cannot balance the traffic distribution across servers.

HTTP(S) load balancing

HTTP(s) load balancing is the oldest type of load balancing, and it relies on Layer 7. This means that load balancing operates in the layer of operations. It is the most flexible type of load balancing because it lets you make delivery decisions based on information retrieved from HTTP addresses.

Internal Load Balancing

It is very similar to network load balancing, but is leveraged to balance the infrastructure internally.

Load balancers can be further divided into hardware, software and virtual load balancers.

Hardware Load Balancer

It depends on the base and the physical hardware that distributes the network and application traffic. The device can handle a large traffic volume, but these come with a hefty price tag and have limited flexibility.

Software Load Balancer

It can be an open source or commercial form and must be installed before it can be used. These are more economical than hardware solutions.

Virtual Load Balancer

It differs from a software load balancer in that it deploys the software to the hardware load-balancing device on the virtual machine.

WHY CLOUD LOAD BALANCING IS IMPORTANT IN CLOUD COMPUTING?

Here are some of the importance of load balancing in cloud computing.

Offers better performance

The technology of load balancing is less expensive and also easy to implement. This allows companies to work on client applications much faster and deliver better results at a lower cost.

Helps Maintain Website Traffic

Cloud load balancing can provide scalability to control website traffic. By using effective load balancers, it is possible to manage high-end traffic, which is achieved using network equipment and servers. E-commerce companies that need to deal with multiple visitors every second use cloud load balancing to manage and distribute workloads.

Can Handle Sudden Bursts in Traffic

Load balancers can handle any sudden traffic bursts they receive at once. For example, in case of university results, the website may be closed due to too many requests. When one uses a load balancer, he does not need to worry about the traffic flow. Whatever the size of the traffic, load balancers will divide the entire load of the website equally across different servers and provide maximum results in minimum response time.

Greater Flexibility

The main reason for using a load balancer is to protect the website from sudden crashes. When the workload is distributed among different network servers or units, if a single node

fails, the load is transferred to another node. It offers flexibility, scalability and the ability to handle traffic better. Because of these characteristics, load balancers are beneficial in cloud environments. This is to avoid heavy workload on a single server.

Scalability and Elasticity

Cloud Elasticity

Elasticity refers to the ability of a cloud to automatically expand or compress the infrastructural resources on a sudden up and down in the requirement so that the workload can be managed efficiently. This elasticity helps to minimize infrastructural costs. This is not applicable for all kinds of environments, it is helpful to address only those scenarios where the resource requirements fluctuate up and down suddenly for a specific time interval. It is not quite practical to use where persistent resource infrastructure is required to handle the heavy workload.

The Flexibility in cloud is a well-known highlight related with scale-out arrangements (level scaling), which takes into consideration assets to be powerfully added or eliminated when required. It is for the most part connected with public cloud assets which is generally highlighted in pay-per-use or pay-more only as costs arise administrations.

The Flexibility is the capacity to develop or contract framework assets (like process, capacity or organization) powerfully on a case by case basis to adjust to responsibility changes in the applications in an autonomic way.

Example: Consider an online shopping site whose transaction workload increases during festive season like Christmas. So for this specific period of time, the resources need a spike up. In order to handle this kind of situation, we can go for a Cloud-Elasticity service rather than Cloud Scalability. As soon as the season goes out, the deployed resources can then be requested for withdrawal.

Cloud Scalability

Cloud scalability is used to handle the growing workload where good performance is also needed to work efficiently with software or applications. Scalability is commonly used where the persistent deployment of resources is required to handle the workload statically.

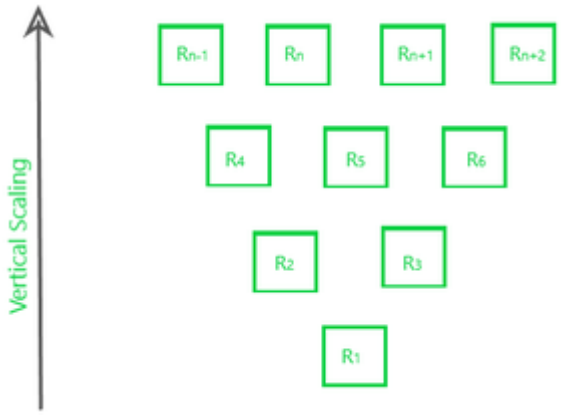
Example: Consider you are the owner of a company whose database size was small in earlier days but as time passed your business does grow and the size of your database also increases, so in this case you just need to request your cloud service vendor to scale up your database capacity to handle a heavy workload.

It is totally different from what you have read above in Cloud Elasticity. Scalability is used to fulfill the static needs while elasticity is used to fulfill the dynamic need of the organization. Scalability is a similar kind of service provided by the cloud where the customers have to pay-per-use. So, in conclusion, we can say that Scalability is useful where the workload remains high and increases statically.

Types of Scalability

1. Vertical Scalability (Scale-up)

In this type of scalability, increase the power of existing resources in the working environment in an upward direction.

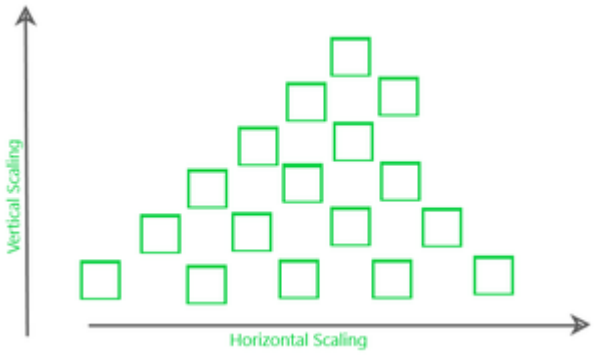


2. **Horizontal Scalability**

In this kind of scaling, the resources are added in a horizontal row.

3. **Diagonal Scalability**

It is a mixture of both Horizontal and Vertical scalability where the resources are added both vertically and horizontally.



Difference Between Cloud Elasticity and Scalability

	Cloud Elasticity	**Cloud Scalability**
1	Elasticity is used just to meet the sudden up and down in the workload for a small period of time.	Scalability is used to meet the static increase in the workload.
2	Elasticity is used to meet dynamic changes, where the resources need can increase or decrease.	Scalability is always used to address the increase in workload in an organization.
3	Elasticity is commonly used by small companies whose workload and demand increases only for a specific period of time.	Scalability is used by giant companies whose customer circle persistently grows in order to do the operations efficiently.
4	It is a short term planning and adopted just to deal with an unexpected increase in demand or seasonal demands.	Scalability is a long term planning and adopted just to deal with an expected increase in demand.

Replication

The simplest form of data replication in cloud computing environment is to store a copy of a file (copy), in expanded form, the copying and pasting in any modern operating systems. Replication is the reproduction of the original data in unchanged form. Changing data accesses are expensive in general through replication. In the frequently encountered master / slave replication, a distinction between the original data (primary data) and the dependent copies. In

peer copies (version control) there must be merging of data sets (synchronization). Sometimes it is important to know which data sets must have the replicas. Depending on the type of replication it is located between the processing and creation of the primary data and their replication in a certain period of time. This period is usually referred to as latency.

Array-Based Data Replication

An array-based data replication strategy uses built-in software to automatically replicate data. With this type of data replication, the software is used in compatible storage arrays to copy data between each. Using this method has several advantages and disadvantages.

Advantages:

- More robust
- Requires less coordination when deployed
- The work gets offloaded from the servers to the storage device

Disadvantages:

- Requires homogenous storage environments: the source and target array have to be similar
- It is costly to implement

Host-Based Data Replication

Host-based data replication uses the servers to copy data from one site to another site. Host-based replication software usually includes options like compression, encryption and, throttling, as well as failover. Using this method has several advantages and disadvantages.

Advantages:

- Flexible: It can leverage existing IP networks
- Can be customized to your business' needs: You can choose what data to replicate
- Can create a schedule for sending data: allows you to throttle bandwidth
- Can use any combination of storage devices on each end

Disadvantages:

- Difficult to manage with a large group of servers if there is no centralized management console
- Consumes host resources during replication
- Both storage devices on each end need to be active, which means you will need to purchase dedicated hardware and OS
- Not all applications can support this type of data replication
- Can be affected by viruses or application failure
- Host-based replication offers the safest option if a business is looking for close to zero impact on operations after a disaster.

Network-Based Data Replication

Network-based data replication uses a device or appliance that sits on the network in the path of the data to manage replication. The data is then copied to a second device. These devices usually have proprietary replication technology but can be used with any host server and storage hardware.

Advantages

- Effective in large, heterogeneous storage and server environments
- Supports any host platform and works with any array
- Works separately from the servers and the storage devices
- Allows replication between multi-vendor products

Disadvantages:

- Higher initial set-up cost because it requires proprietary hardware, as well as ongoing operational and management costs
- Requires implementation of a storage area network (SAN)

Monitoring

Cloud monitoring is a method of reviewing, observing, and managing the operational workflow in a cloud-based IT infrastructure. Manual or automated management techniques confirm the availability and performance of websites, servers, applications, and other cloud

infrastructure. This continuous evaluation of resource levels, server response times, and speed predicts possible vulnerability to future issues before they arise.

This technique tracks multiple analytics simultaneously, monitoring storage resources and processes that are provisioned to virtual machines, services, databases, and applications. This technique is often used to host infrastructure-as-a-service (IaaS) and software-as-a-service (SaaS) solutions. For these applications, you can configure monitoring to track performance metrics, processes, users, databases, and available storage. It provides data to help you focus on useful features or to fix bugs that disrupt functionality.

Monitoring is a skill, not a full-time job. In today's world of cloud-based architectures that are implemented through DevOps projects, developers, site reliability engineers (SREs), and operations staff must collectively define an effective cloud monitoring strategy. Such a strategy should focus on identifying when service-level objectives (SLOs) are not being met, likely negatively affecting the user experience. So, then what are the benefits of leveraging cloud monitoring tools? With cloud monitoring:

Benefits of cloud monitoring

- Scaling for increased activity is seamless and works in organizations of any size
- Dedicated tools (and hardware) are maintained by the host
- Tools are used across several types of devices, including desktop computers, tablets, and phones, so your organization can monitor apps from any location
- Installation is simple because infrastructure and configurations are already in place
- Your system doesn't suffer interruptions when local problems emerge, because resources aren't part of your organization's servers and workstations
- Subscription-based solutions can keep your costs low

Cloud monitoring is primarily part of cloud security and management processes. It is normally implemented through automated monitoring software that provides central access and control over cloud infrastructure.

Cloud Services and Platforms

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

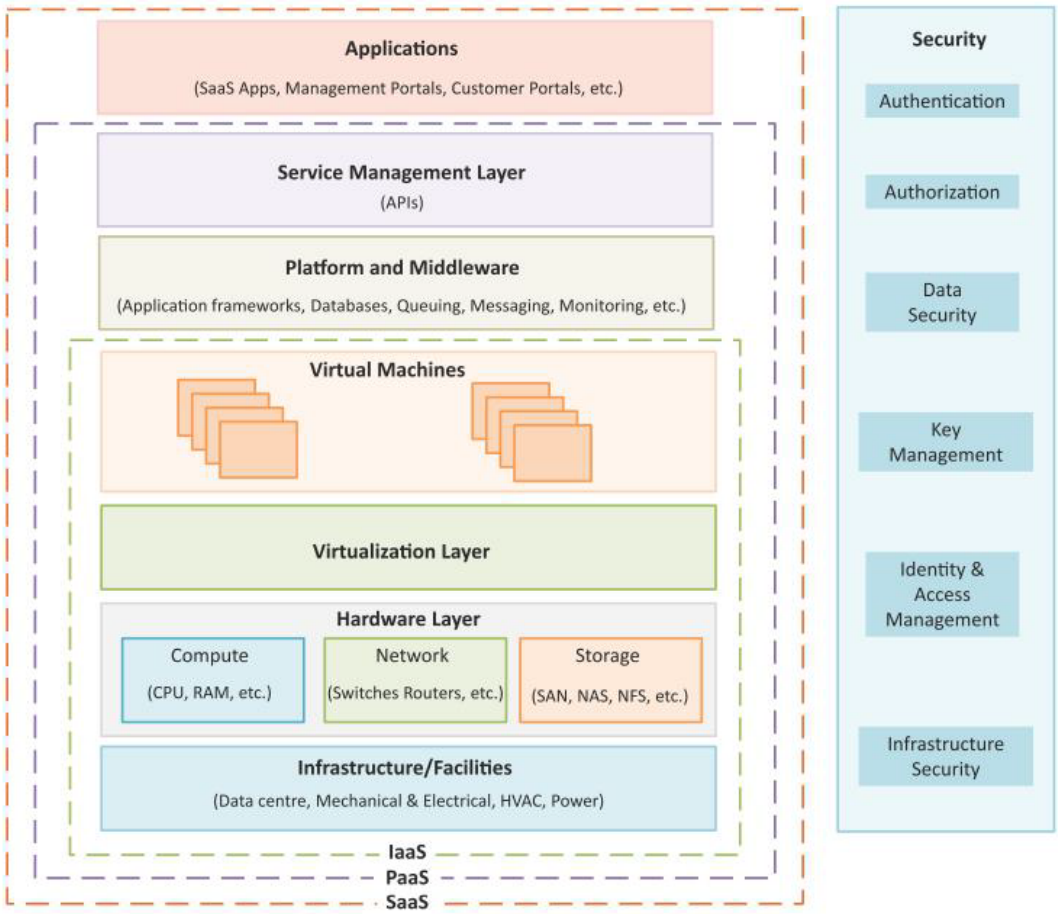
Includes SaaS applications such as Email, cloud storage application, productivity applications, management portals, customer self-service portals, etc.

• 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.



Compute Service

- 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 these services.
- Cloud service providers also provide APIs for various programming languages that allow developers to access and manage these services programmatically.

Compute Service - 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 launch the 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

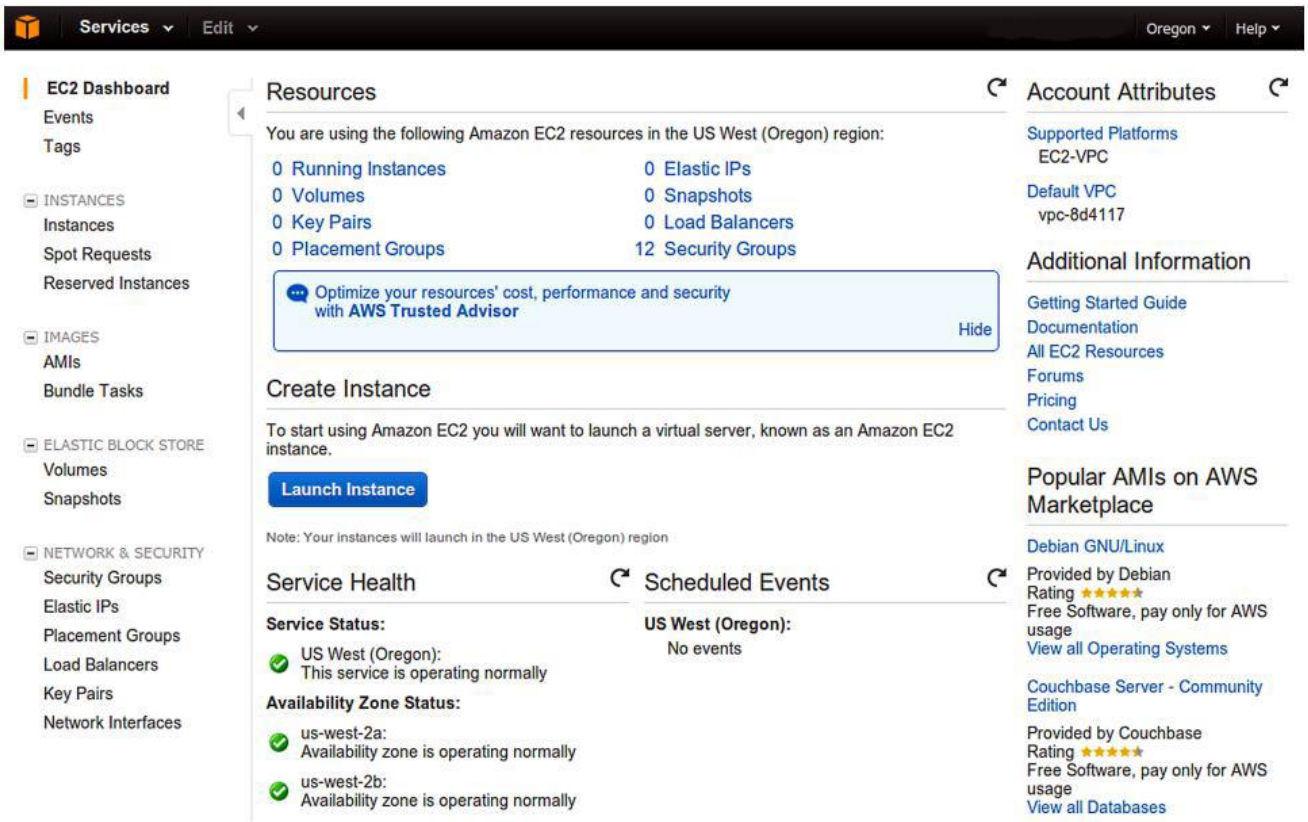
  When you launch an instance you specify the instance type (micro, small, medium, large, extra-large, etc.), the number of instances to launch based on the selected AMI and availability zones for the instances.

- Key-pairs

  When launching a new instance, the user selects a key-pair from existing keypairs or creates 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 network port for the launched instances.



Compute Services – Google Compute Engine

- Google Compute Engine is a compute service provided by Google.

- **Launching Instances**

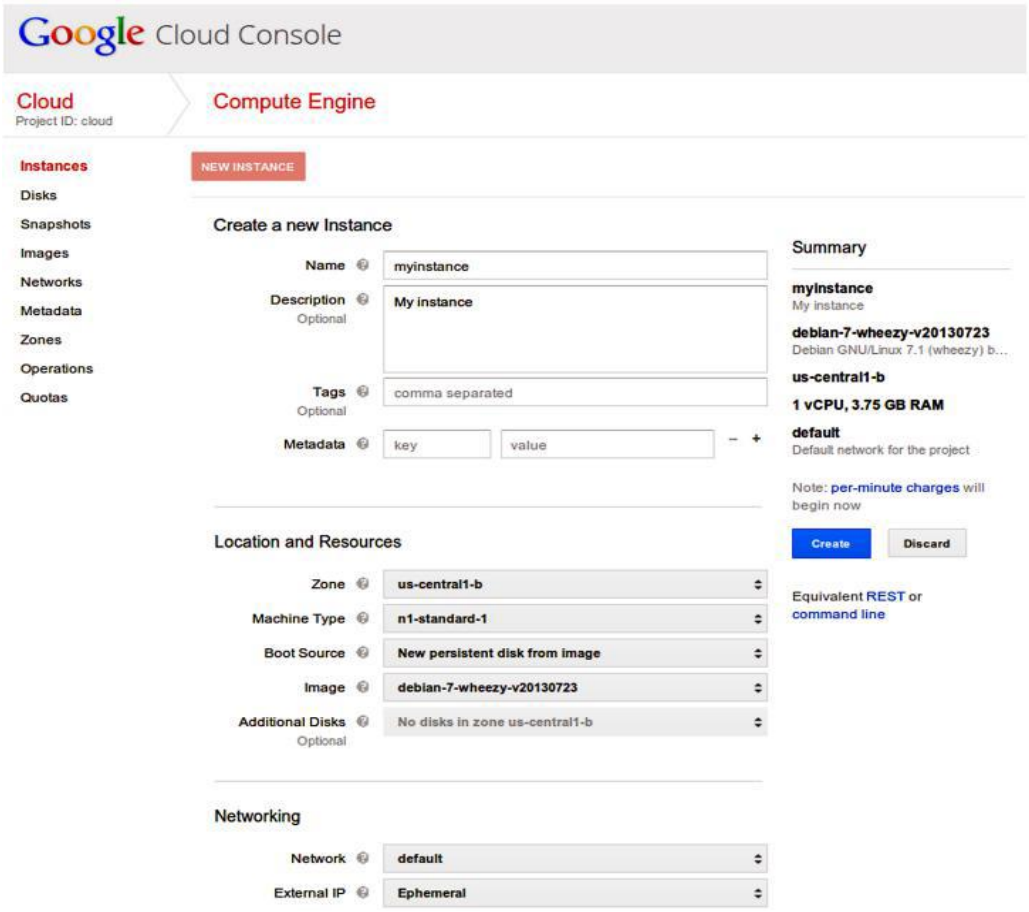
To create a new instance, the user selects an instance machine type, a zone in which the instance will be launched, a machine image for the instance and provides an instance name, instance tags and meta-data.

- **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.

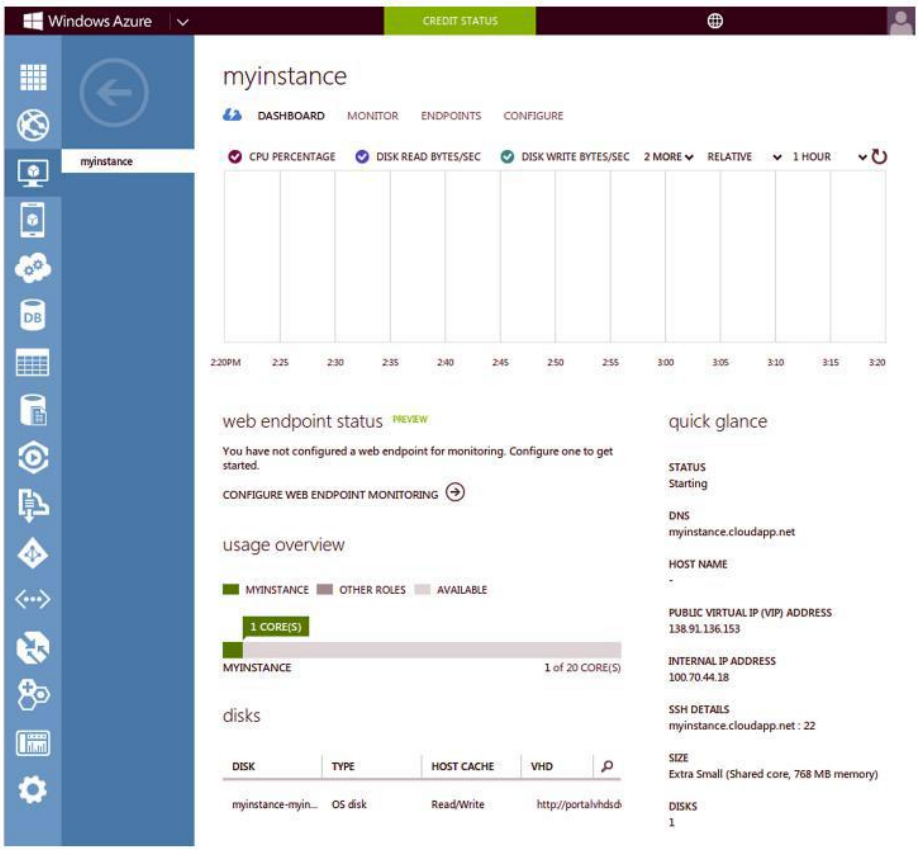
- **Network Options**

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.



Compute Services – 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.



Storage Services

- Cloud storage services allow storage and retrieval of any amount of data, at any time from anywhere on the web.
- 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 and multiple 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 be immediately 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 and affordable 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.

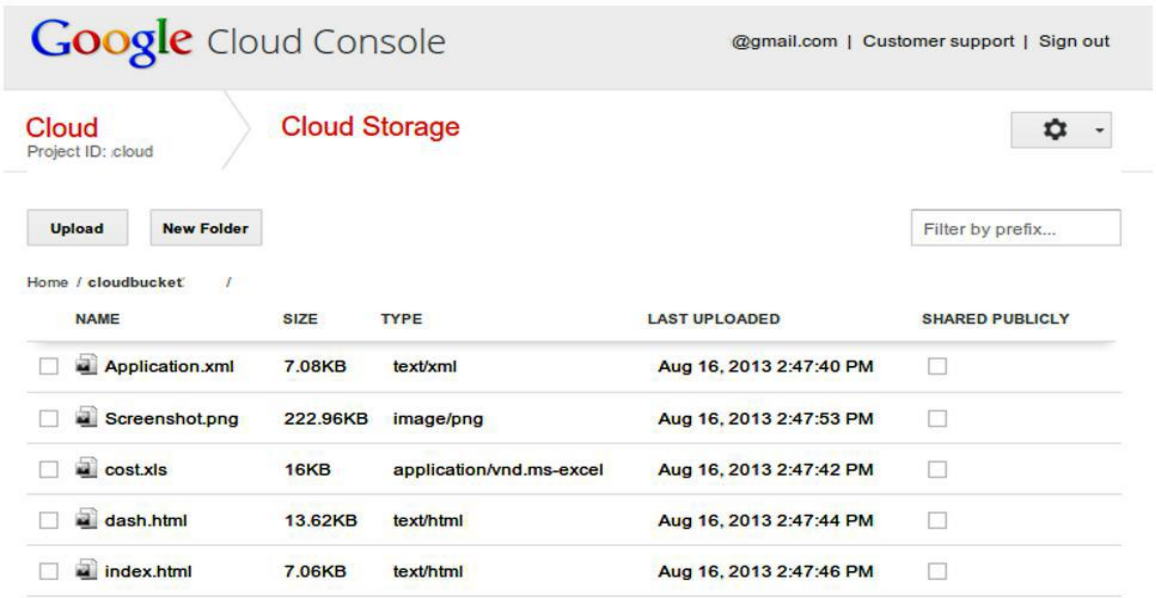
- **Uploading Files to Buckets**
  - S3 console provides simple wizards for creating a new bucket and uploading files.
  - You can upload any kind of file to S3.
  - While uploading a file, you can specify the redundancy and encryption options and access permissions.

Storage Services – Google Cloud Storage

- GCS is the Cloud storage service from Google
- Buckets

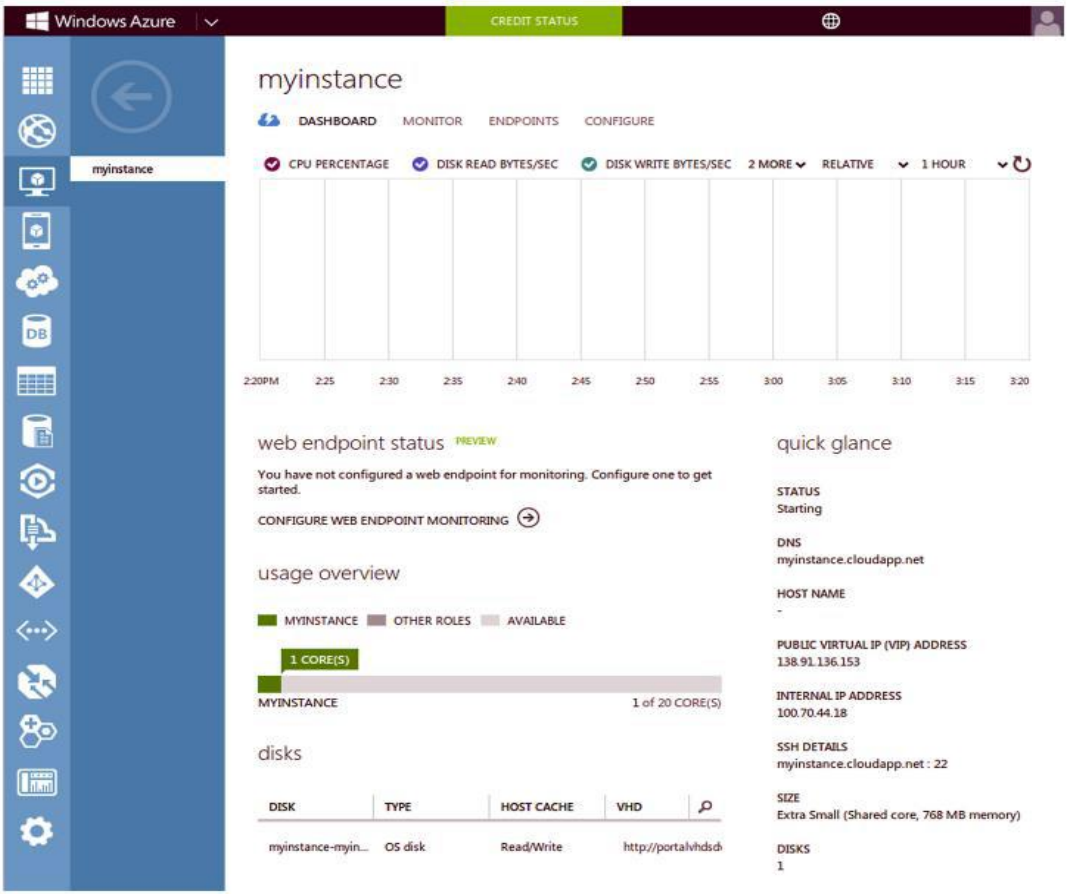
  Objects in GCS are organized into buckets.
- Access Control Lists

  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 from Microsoft.
- 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 binary large 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.



Application Runtimes & Frameworks

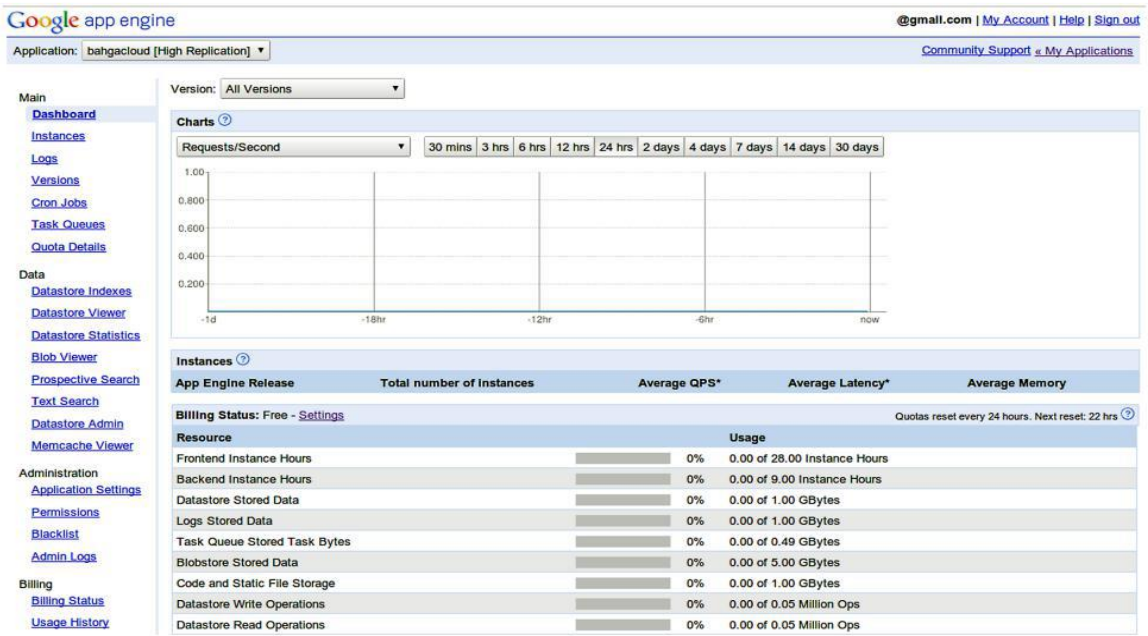
- Cloud-based application runtimes and frameworks allow developers to develop and host applications in the cloud.
- Support for various programming languages

  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 environment isolated from other applications.
  - The sandbox environment provides a limited access to the underlying operating system.



- 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 Web Sites

- 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 web sites 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.