INTRODUCTION TO DISTRIBUTED SYSTEMS

INTRODUCTION

The process of computation was started from working on a single processor. This uni-processor computing can be termed as **centralized computing**. As the demand for the increased processing capability grew high, multiprocessor systems came to existence. The advent of multiprocessor systems, led to the development of distributed systems with high degree of scalability and resource sharing. The modern day parallel computing is a subset of distributed computing

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

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. Distributed computing is widely used due to advancements in machines; faster and cheaper networks. In distributed systems, the entire network will be viewed as a computer. The multiple systems connected to the network will appear as a single system to the user. Thus the distributed systems hide the complexity of the underlying architecture to the user. Distributed computing is a special version of parallel computing where the processors are in different computers and tasks are distributed to computers over a network.

The definition of distributed systems deals with two aspects that:

Deals with hardware: The machines linked in a distributed system are autonomous.

Deals with software: A distributed system gives an impression to the users that they are dealing with a single system.

Features of Distributed Systems:

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 - A key feature that requires message-passing for communication. This feature implies the absence of the common physical clock.

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

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

Issues in distributed systems

Heterogeneity

Openness

Security

Scalability

Failure handling

Concurrency

Transparency

Quality of service

QOS parameters

The distributed systems must offer the following QOS:

- Performance
- Reliability
- Availability
- Security

Differences between centralized and distributed systems

Centralized Systems	**Distributed Systems**
In Centralized Systems, several jobs are done on a particular central processing unit(CPU)	In Distributed Systems, jobs are distributed among several processors. The Processor are interconnected by a computer network
They have shared memory and shared variables.	They have no global state (i.e.) no shared memory and no shared variables.
Clocking is present.	No global clock.

RELATION TO COMPUTER SYSTEM COMPONENTS

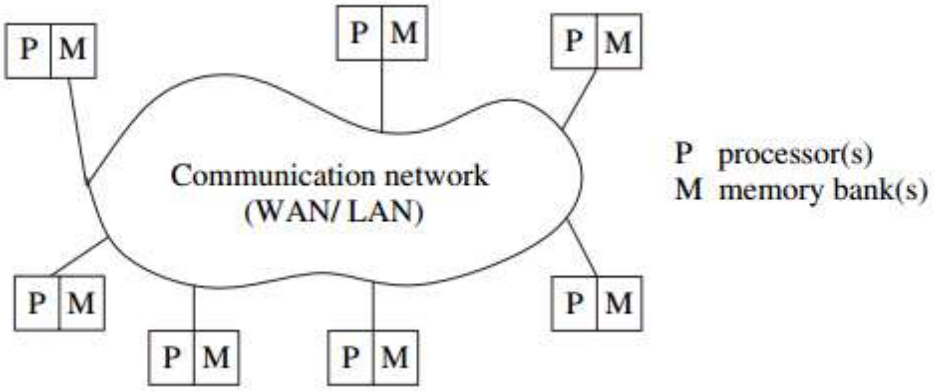


Fig : Example of a Distributed System

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

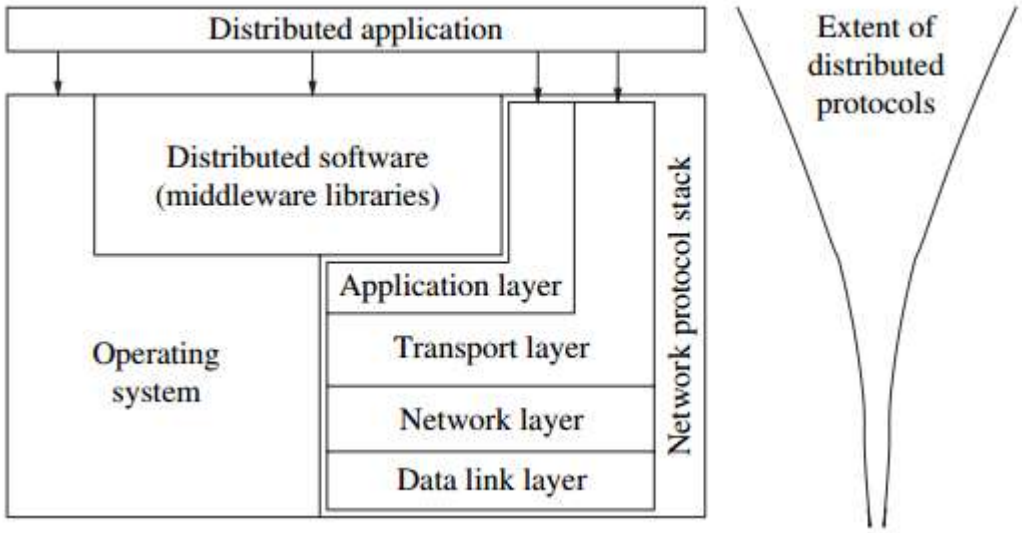


Fig : Interaction of layers of network

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

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

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

MOTIVATION

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

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

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

Access to geographically remote data and resources: As mentioned previously, computations may happen at remote locations. Resources such as centralized servers can also be accessed from distant locations.

Enhanced reliability: DS provides enhanced reliability, since they run on multiple copies of resources. The distribution of resources at distant locations makes them less susceptible for faults.

The term reliability comprises of:

1. **Availability:** the resource/ service provided by the resource should be accessible at all times
2. **Integrity:** the value/state of the resource should be correct and consistent.
3. **Fault-Tolerance:** the ability to recover from system failures

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

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

RELATION TO PARALLEL SYSTEMS

> ***Parallel Processing Systems divide the program into multiple segments and process them simultaneously.***

The main objective of parallel systems is to improve the processing speed. They are sometimes known as **multiprocessor or multi computers or tightly coupled systems.** They refer to simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.

Characteristics of parallel systems

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

1.* A *multiprocessor system

2.* A *multicomputer parallel system

3. Array processors

1.* A *multiprocessor system

A *multiprocessor system* is a parallel system in which the multiple processors have *direct access to shared memory* which forms a common address space. The architecture is shown in Figure (a). Such processors usually do not have a common clock.

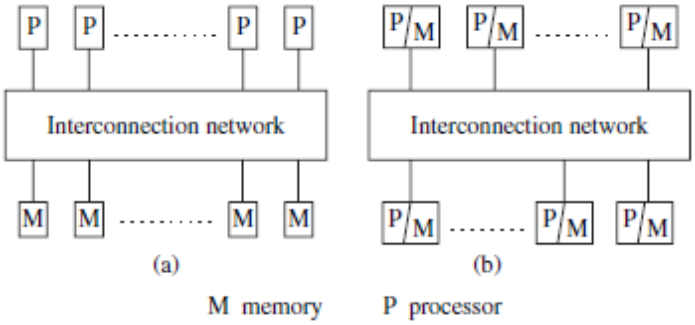


Figure: Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor.

i) Uniform Memory Access (UMA)

- Here, all the processors share the physical memory in a centralized manner with equal access time to all the memory words.
- Each processor may have a private cache memory. Same rule is followed for peripheral devices.

- When all the processors have equal access to all the peripheral devices, the system is called a **symmetric multiprocessor.**
- When only one or a few processors can access the peripheral devices, the system is called an **asymmetric multiprocessor.**
- When a CPU wants to access a memory location, it checks if the bus is free, then it sends the request to the memory interface module and waits for the requested data to be available on the bus.
- Multicore processors are small UMA multiprocessor systems, where the first shared cache is actually the communication channel.

ii) Non-uniform Memory Access (NUMA)

- In NUMA multiprocessor model, the access time varies with the location of the memory word.
- Here, the shared memory is physically distributed among all the processors, called local memories.
- The collection of all local memories forms a global address space which can be accessed by all the processors.
- NUMA systems also share CPUs and the address space, but each processor has a local memory, visible to all other processors.
- In NUMA systems access to local memory blocks is quicker than access to remote memory blocks.

Figure shows two popular interconnection networks – the Omega network and the Butterfly network, each of which is a multi-stage network formed of 2×2 switching elements. Each 2×2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire. In a single step, however, only one data unit can be sent on an output wire. So if the data from both the input wires is to be routed to the same output wire in a single step, there is a collision.

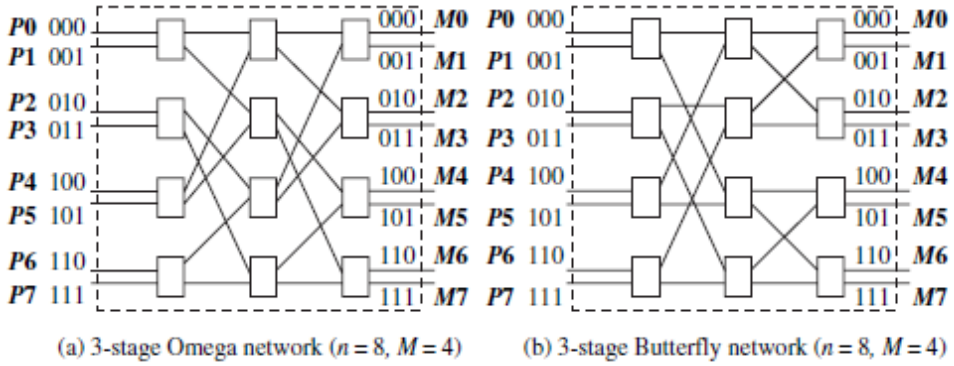


Figure: Interconnection networks for shared memory multiprocessor systems

Omega interconnection function

The Omega network which connects n processors to n memory units has n/2log2 n switching elements of size 2✖2 arranged in log2 n stages. Between each pair of adjacent stages of the Omega network, a link exists between output i of a stage and the input j to the next stage according to the following perfect shuffle pattern which is a left-rotation operation on the binary representation of i to get j. The generation function is given as:

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

The routing function from input line i to output line j considers only j and the stage number s, where s ∈ [0, log n – 1]. In a stage s switch, if the s + 1th most significant bit of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

Butterfly network

A butterfly network links multiple computers into a high-speed network. For a butterfly network with n processor nodes, there need to be n (log n + 1) switching nodes. The generation of the interconnection pattern between a pair of adjacent stages depends not only on n but also on the stage numbers.In a stage (s) switch, if the s + 1th MSB of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

2.* A *multicomputer parallel system

It is a parallel system in which the multiple processors *do not have direct access to shared memory*. The memory of the multiple processors may or may not form a common

address space. Such computers usually do not have a common clock. The architecture is shown in Figure (b).

Torus or 2D Mesh Topology

A k X k mesh will contain k^2processor with maximum path length as 2*(k/2 -1). Every unit in the torus topology is identified using a unique label, with dimensions distinguished as bit positions.

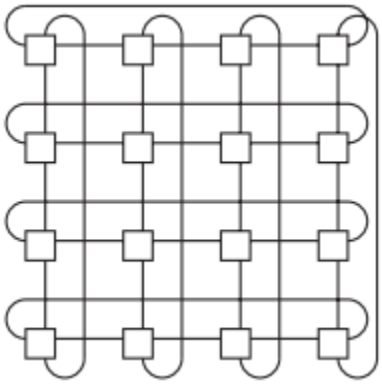


Fig : 2-D Mesh

Hypercube

The path between any two nodes in 4-D hypercube is found by Hamming distance. Routing is done in hop to hop fashion with each adjacent node differing by one bit label. This topology has good congestion control and fault tolerant mechanism.

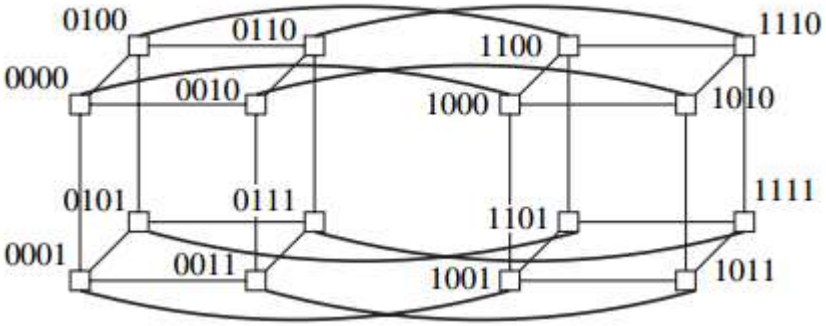


Fig : 4-D Hypercube

3. ArrayProcessors

They are a class of processors that executes one instruction at a time in an array or table of data at the same time rather than on single data elements on a common clock.

They are also known as vector processors. An array processor implement the instruction set where each instruction is executed on all data items associated and then move on the other instruction. Array elements are incapable of operating autonomously, and must be driven by the control unit.

Flynn's Taxonomy

> ***Flynn's taxonomy is a specific classification of parallel computer architectures that are based on the number of concurrent instruction (single or multiple) and data streams (single or multiple) available in the architecture.***

Flynn's taxonomy based on the number of instruction streams and data streams are the following:

1. (SISD) single instruction, single data
2. (MISD) multiple instruction, single data
3. (SIMD) single instruction, multiple data
4. (MIMD) multiple instruction, multiple data

1. SISD (Single Instruction, Single Data stream)

- Single Instruction, Single Data (SISD) refers to an Instruction Set Architecture in which a single processor (one CPU) executes exactly one instruction stream at a time.
- It also fetches or stores one item of data at a time to operate on data stored in a single memory unit.
- Most of the CPU design is based on the von Neumann architecture and the follow SISD.
- The SISD model is a non-pipelined architecture with general-purpose registers, Program Counter (PC), the Instruction Register (IR), Memory Address Registers (MAR) and Memory Data Registers (MDR).

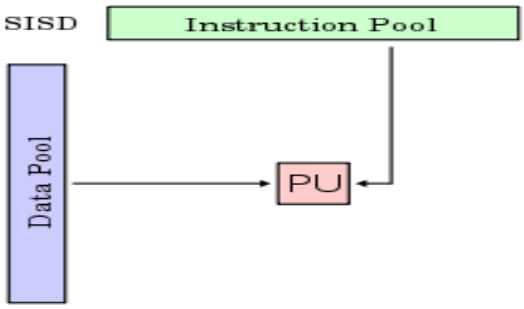


Fig : Single Instruction, Single Data Stream

SIMD (Single Instruction, Multiple Data streams)

- Single Instruction, Multiple Data (SIMD) is an Instruction Set Architecture that have a single control unit (CU) and more than one processing unit (PU) that operates like a von Neumann machine by executing a single instruction stream over PUs, handled through the CU.
- The CU generates the control signals for all of the PUs and by which executes the same operation on different data streams. The SIMD architecture is capable of achieving data level parallelism.

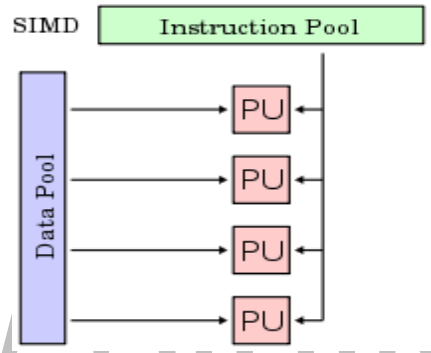


Fig :Single Instruction, Multiple Data streams

MISD (Multiple Instruction, Single Data stream)

- Multiple Instruction, Single Data (MISD) is an Instruction Set Architecture for parallel computing where many functional units perform different operations by executing different instructions on the same data set.
- This type of architecture is common mainly in the fault-tolerant computers executing the same instructions redundantly in order to detect and mask errors.

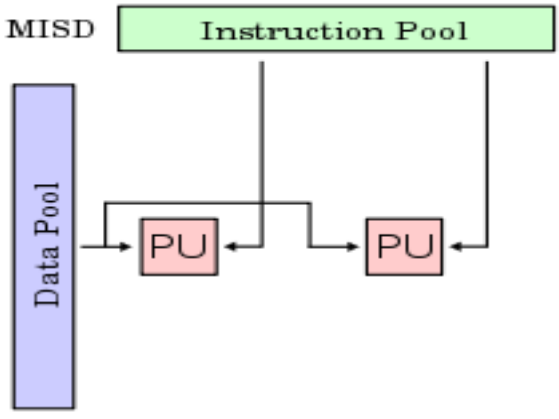


Fig : Multiple Instruction, Single Data stream

MIMD (Multiple Instruction, Multiple Data streams)

- Multiple Instruction stream, Multiple Data stream (MIMD) is an Instruction Set Architecture for parallel computing that is typical of the computers with multiprocessors.
- Using the MIMD, each processor in a multiprocessor system can execute asynchronously different set of the instructions independently on the different set of data units.
- The MIMD based computer systems can used the shared memory in a memory pool or work using distributed memory across heterogeneous network computers in a distributed environment.
- The MIMD architectures is primarily used in a number of application areas such as computer-aided design/computer-aided manufacturing, simulation, modelling, communication switches etc.

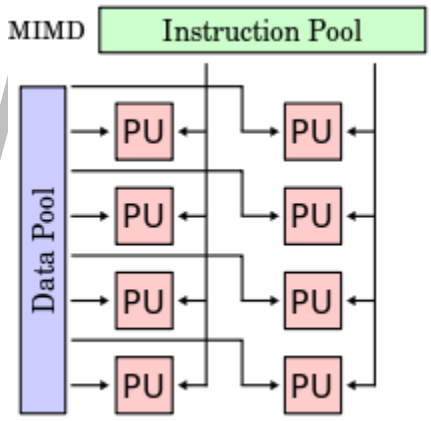


Fig :Multiple Instruction, Multiple Data streams

	Single	**Multiple**
Single	**SISD** Von Neumann Single computer	**MISD** May be pipelined computers
Multiple	**SIMD** Vector processors Fine grained data Parallel computers	**MIMD** Multi computers Multiprocessors

Coupling, parallelism, concurrency, and granularity

Coupling

The term **coupling** is associated with the configuration and design of processors in a multiprocessor system.

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

The multiprocessor systems are classified into two types based on coupling:

1. Loosely coupled systems
2. Tightly coupled systems

Tightly Coupled systems:

- Tightly coupled multiprocessor systems contain multiple CPUs that are connected at the bus level with both local as well as central shared memory.
- Tightly coupled systems perform better, due to faster access to memory and intercommunication and are physically smaller and use less power. They are economically costlier.
- Tightly coupled multiprocessors with UMA shared memory may be either switch-based (e.g., NYU Ultracomputer, RP3) or bus-based (e.g., Sequent, Encore).
- Some examples of tightly coupled multiprocessors with NUMA shared memory or that communicate by message passing are the SGI Origin 2000

Loosely Coupled systems:

- Loosely coupled multiprocessors consist of distributed memory where each processor has its own memory and IO channels.
- The processors communicate with each other via message passing or interconnection switching.
- Each processor may also run a different operating system and have its own bus control logic.
- Loosely coupled systems are less costly than tightly coupled systems, but are physically bigger and have a low performance compared to tightly coupled systems.

- The individual nodes in a loosely coupled system can be easily replaced and are usually inexpensive.
- The extra hardware required to provide communication between the individual processors makes them complex and less portable.
- Loosely coupled multicomputers without shared memory are physically co-located. These may be bus-based (e.g., NOW connected by a LAN or Myrinet card) or using a more general communication network.
- These processors neither share memory nor have a common clock.
- Loosely coupled multicomputers without shared memory and without common clock and that are physically remote, are termed as distributed systems.

Parallelism or speedup of a program on specific system

- It is the use of multiple processing elements simultaneously for solving any problem.
- Problems are broken down into instructions and are solved concurrently as each resource which has been applied to work is working at the same time.
- This is a measure of the relative speedup of a specific program, on a given machine. The speedup depends on the number of processors and the mapping.
- It is expressed as the ratio of the time T(1) witha single processor, to the time T(n) with n processors.

Parallelism within a parallel/distributed program

- This is an aggregate measure of the percentage of time that all the processors are executing CPU instructions productively, as opposed to waiting for communication operations.

Concurrency

- Concurrent programming refer to techniques for decomposing a task into subtasks that can execute in parallel and managing the risks that arise when the program executes more than one task at the same time.
- The parallelism or concurrency in a parallel or distributed program can be measured by the ratio of the number of local non-communication and non-shared memory access operations to the total number of operations, including the communication or shared memory access operations.

Granularity

> ***Granularity or grain size is a measure of the amount of work or computation that is performed by that task.***

- Granularity is also the communication overhead between multiple processors or processing elements.
- In this case, granularity as the ratio of computation time to communication time, wherein, the computation time is the time required to perform the computation of a task and communication time is the time required to exchange data between processors.

Parallelism can be classified into three categories based on work distribution among the parallel tasks:

1. **Fine-grained:** Partitioning the application into small amounts of work done leading to a low computation to communication ratio.
2. **Coarse-grained parallelism:** This has high computation to communication ratio.
3. **Medium-grained:** Here the task size and communication time greater than fine-grained parallelism and lower than coarse-grained parallelism.

   Programs with fine-grained parallelism are best suited for tightly coupled systems.

Classes of OS of Multiprocessing systems:

- **Network Operating Systems:** The operating system running on loosely coupled processors which are themselves running loosely coupled software
- **Distributed Operating systems**: The OS of the system running on loosely coupled processors, which are running tightly coupled software.
- **Multiprocessor Operating Systems:** The OS will run on tightly coupled processors, which are themselves running tightly coupled software.

A MODEL OF DISTRIBUTED COMPUTATIONS: DISTRIBUTED PROGRAM

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

A MODEL OF DISTRIBUTED EXECUTIONS

- The execution of a process consists of a sequential execution of its actions.
- The actions are atomic and the actions of a process are modeled as three types of events: internal events, message send events, and message receive events.
- The occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- The execution of process pi produces a sequence of events e1, e2, e3, …, and it is denoted by H_i: $H_i = (h_i \rightarrow_i)$. Here h_i are states produced by p_i and $\rightarrow$ are the casual dependencies among events p_i.
- $\rightarrow_{msg}$ indicates the dependency that exists due to message passing between two events.
- $send(m) \rightarrow_{msg} rec(m)$

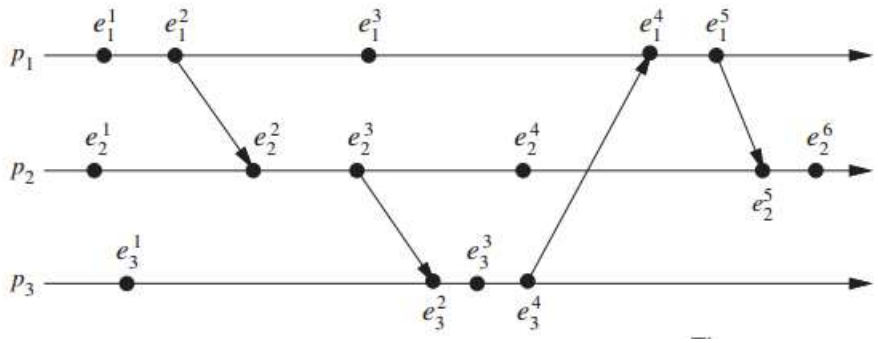


Fig: Space time distribution of distributed systems

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

Casual Precedence Relations

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

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

Happen Before Relation

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

- ✓ **Local ordering:**A and B occur on same process and A occurs before B.
- ✓ **Messages:** send(m) → receive(m) for any message m

✓ **Transitivity:** e → e'' if e → e' and e' → e''

- Ordering can be based on two situations:

1. If two events occur in same process then they occurred in the order observed.
2. During message passing, the event of sending message occurred before the event of receiving it.

Lamports ordering is happen before relation denoted by →

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

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

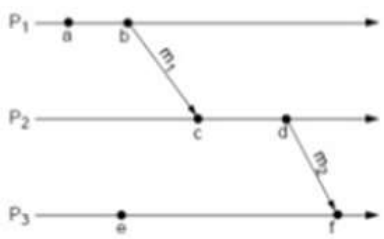


Fig: Communication between processes

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

Logical vs physical concurrency

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

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

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

Differences between logical and physical concurrency

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

LOGICAL TIME

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

Precise physical clocking is not possible in distributed systems. The asynchronous distributed systems spans logical clock for coordinating the events.Three types of logical clock are maintained in distributed systems:

- Scalar clock
- Vector clock
- Matrix clock

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

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

Differences between physical and logical clock

Physical Clock	Logical Clock
A physical clock is a physical procedure combined with a strategy for measuring that procedure to record the progression of time.	A logical clock is a component for catching sequential and causal connections in a dispersed framework.
The physical clocks are based on cyclic processes such as a celestial rotation.	A logical clock allows global ordering on events from different processes.

A Framework for a system of logical clocks

A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation <. This relation is usually called the happened before or causal precedence.

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

$$C : H \mapsto T \text{ such that}$$

for any 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,

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

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

Implementing logical clocks

The two major issues in implanting logical clocks are:

Data structures: representation of each process

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

Data structures:

Each process $p_i$maintains data structures with the given capabilities:

• A local logical clock (lc_i), that helps process p_i measure itsown progress.

• A logical global clock (gc_i), that is a representation of process p_i's local view of the logical global time. It allows this process to assignconsistent timestamps to its local events.

Protocol:

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

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

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

SCALAR TIME

Scalar time is designed by Lamport to synchronize all the events in distributed systems. A Lamport logical clock is an incrementing counter maintained in each process. This logical clock has meaning only in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with that sender maintainingcausal relationship.

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

- The algorithm of Lamport Timestamps can be captured in a few rules:
- All the process counters start with value 0.
- A process increments its counter for each event (internal event, message sending, message receiving) in that process.

- When a process sends a message, it includes its (incremented) counter value with the message.
- On receiving a message, the counter of the recipient is updated to the greater of its current counter and the timestamp in the received message, and then incremented by one.

- If C_i is the local clock for process P_i then,
- if a and b are two successive events in P_i, then $C_i(b) = C_i(a) + d1$, where $d1 > 0$
- if a is the sending of message m by P_i, then m is assigned timestamp $t_m = C_i(a)$
- if b is the receipt of m by P_j, then $C_j(b) = \max\{C_j(b), t_m + d2\}$, where $d2 > 0$

Rules of Lamport's clock

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

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

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

b) Execute Rule 1

c) deliver the message

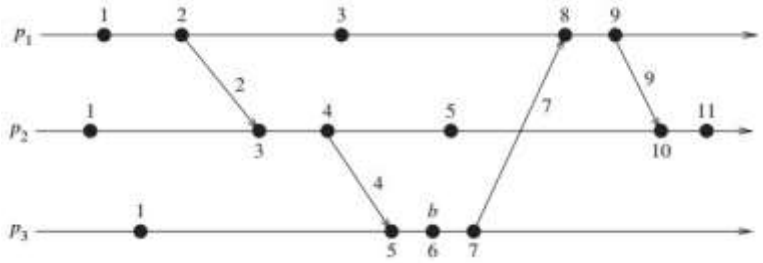


Fig : Evolution of scalar time

Basic properties of scalar time:

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

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

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

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

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

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

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

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

3. Event Counting

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

4. No strong consistency

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

VECTOR TIME

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

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

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

Rules of Vector Time

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

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

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

1. update its global logical time

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

2. execute R1

3. deliver the message m

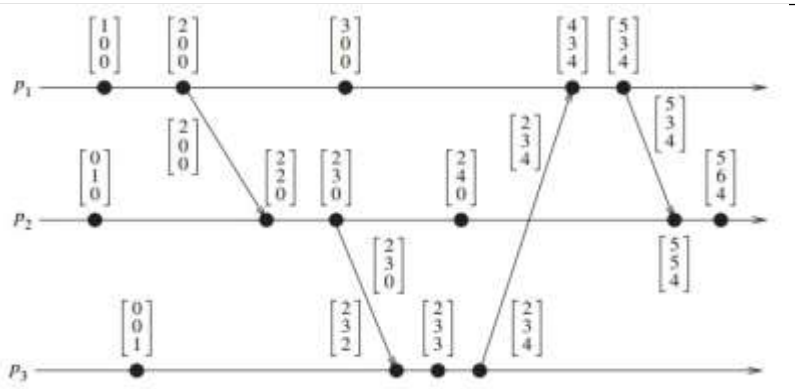


Fig : Evolution of vector scale

Basic properties of vector time

1. Isomorphism:

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

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

- $$x \parallel y \Leftrightarrow vh \parallel vk.$$
- There is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps.
- If the process at which an event occurred is known, the test to compare two timestamps can be simplified as:

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

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

2. Strong consistency

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

3. Event counting

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

Vector clock ordering relation

$$t = t' \Leftrightarrow \forall i\ t[i] = t'[i]$$
$$t \neq t' \Leftrightarrow \exists i\ t[i] \neq t'[i]$$
$$t \leq t' \Leftrightarrow \forall i\ t[i] \leq t'[i]$$
$$t < t' \Leftrightarrow (t \leq t' \text{ and } t \neq t')$$
$$t \,||\, t' \Leftrightarrow \text{not } (t < t' \text{ or } t' < t)$$

t[i]- timestamp of process i.

MESSAGE-PASSING SYSTEMS VERSUS SHARED MEMORY SYSTEMS

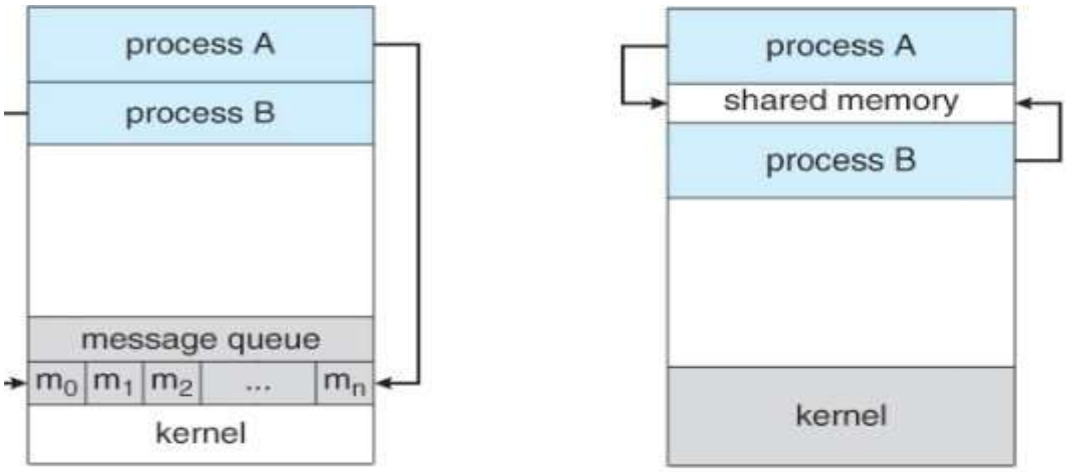
Communication among processors takes place via shared data variables, and control variables for synchronization among the processors**.** The communications between the tasks in multiprocessor systems take place through two main modes:

Message passing systems:

- This allows multiple processes to read and write data to the message queue without being connected to each other.
- Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for inter process communication and are used by most operating systems.

Shared memory systems:

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



a) Message Passing Model **b) Shared Memory Model**

Fig : Inter-process communication models

Differences between message passing and shared memory models

Message Passing	Distributed Shared Memory
Services Offered: Variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process.	The processes share variables directly, so no marshalling and unmarshalling. Shared variables can be named, stored and accessed in DSM.
Processes can communicate with other processes. They can be protected from one another by having private address spaces.	Here, a process does not have private address space. So one process can alter the execution of other.
This technique can be used in heterogeneous computers.	This cannot be used to heterogeneous computers.
Synchronization between processes is through message passing primitives.	Synchronization is through locks and semaphores.
Processes communicating via message passing must execute at the same time.	Processes communicating through DSM may execute with non-overlapping lifetimes.
Efficiency: All remote data accesses are explicit and therefore the programmer is always aware of whether a particular operation is in-process or involves the expense of communication.	Any particular read or update may or may not involve communication by the underlying runtime support.

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

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

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

- This is also implemented through read and write operations. Each shared location can be modeled as a separate process. Write to a shared location is emulated by sending an update message tothe corresponding owner process and read operation to a shared location is emulated by sending a query message to the owner process.
- This emulation is expensive as the processes have to gain access to other process memory location. The latencies involved in read and write operations may be high even when using shared memory emulation because the read and write operations are implemented by using network-wide communication.

PRIMITIVES FOR DISTRIBUTED COMMUNICATION

Blocking / Non blocking / Synchronous / Asynchronous

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

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

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

Blocking primitives

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

- The receipt is determined by polling common buffer or interrupt
- This is a form of synchronization or synchronous communication.
- A primitive is blocking if control returns to the invoking process after the processing for the primitive completes.

Non Blocking primitives

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

Synchronous

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

- 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.
- Implementing non -blocking operations are tricky.

- For non-blocking primitives, a return parameter on the primitive call returns a system-generated **handle** which can be later used to check the status of completion of the call.
- The process can check for the completion:
    - checking if the handle has been flagged or posted
    - issue a Wait with a list of handles as parameters: usually blocks until one of the parameter handles is posted.

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

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

Four modes of send operation

Blocking synchronous Send:

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

Non-blocking synchronous Send:

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

Blocking asynchronous Send:

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

Non-blocking asynchronous Send:

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

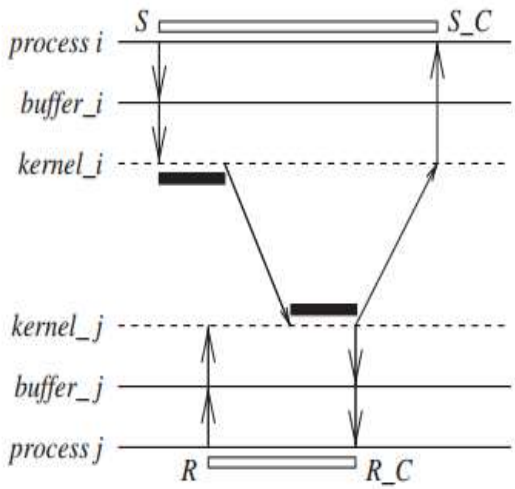


Fig a) Blocking synchronous send and blocking receive

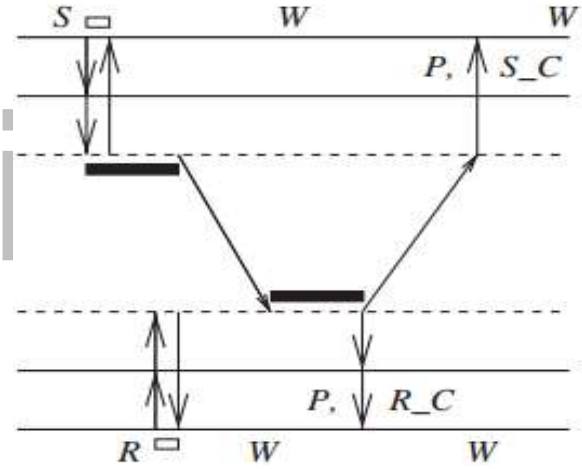


Fig b) Non-blocking synchronous send and blocking receive

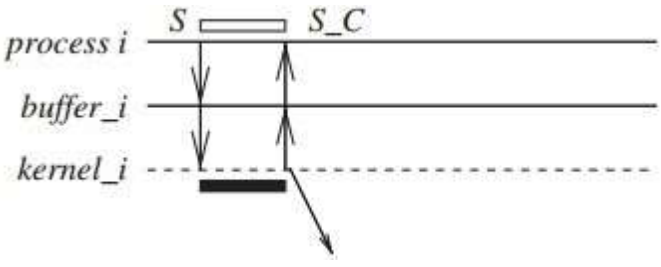


Fig c) Blocking asynchronous send

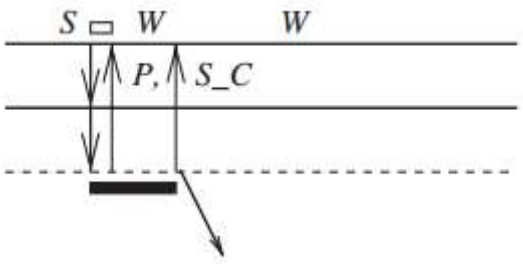


Fig d) Non-blocking asynchronous send

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

Modes of receive operation

Blocking Receive:

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

Non-blocking Receive:

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

Processor Synchrony

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

Since distributed systems do not follow a common clock, 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. The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community

- **Message Passing Interface (MPI):** This is a standardized and portable message-passing system to function on a wide variety of parallel computers. MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.
- The primary goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.
- **Parallel Virtual Machine (PVM):** It is a software tool for parallel networking of computers. It is designed to allow a network of heterogeneous Unix and/or Windows machines to be used as a single distributed parallel processor.
- **Remote Procedure Call (RPC):** The Remote Procedure Call (RPC) is a common model of request reply protocol. In RPC, the procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.
- **Remote Method Invocation (RMI):** RMI (Remote Method Invocation) is a way that a programmer can write object-oriented programming in which objects on different computers can interact in a distributed network. It is a set of protocols being developed by Sun's JavaSoft division that enables Java objects to communicate remotely with other Java objects.
- **Remote Procedure Call (RPC):** RPC is a powerful technique for constructing distributed, client-server based applications. In RPC, the procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. RPC makes the client/server model of computing more powerful and easier to program.

Differences between RMI and RPC

RMI	RPC
RMI uses an object oriented paradigm where the user needs to know the object and the method of the object he needs to invoke.	RPC is not object oriented and does not deal with objects. Rather, it calls specific subroutines that are already established

With RPC looks like a local call. RPC handles the complexities involved with passing the call from the local to the remote computer.	RMI handles the complexities of passing along the invocation from the local to the remote computer. But instead of passing a procedural call, RMI passes a reference to the object and the method that is being called.

The commonalities between RMI and RPC are as follows:

- ✓ They both support programming with interfaces.
- ✓ They are constructed on top of request-reply protocols.
- ✓ They both offer a similar level of transparency.

- **Common Object Request Broker Architecture (CORBA):** CORBA describes a messaging mechanism by which objects distributed over a network can communicate with each other irrespective of the platform and language used to develop those objects. The data representation is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages.

MODELS OF COMMUNICATION NETWORK

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

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

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

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

- The relation between the three models is given by CO ⊂ FIFO ⊂ N-FIFO.

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

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

GLOBAL STATE

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

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

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

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

A distributed snapshot should reflect a consistent state. A global state is consistent if it could have been observed by an external observer. For a successful Global State, all states must be consistent:

- If we have recorded that a process P has received a message from a process Q, then we should have also recorded that process Q had actually send that message.
- Otherwise, a snapshot will contain the recording of messages that have been received but never sent.

- The reverse condition (Q has sent a message that P has not received) is allowed.

The notion of a global state can be graphically represented by what is called a **cut**. A cut represents the last event that has been recorded for each process.

The history of each process if given by:

$history(p_i) = h_i = <e_i^0, e_i^1, e_i^2, \ldots>$

Each event either is an internal action of the process. We denote by s_i^k the state of process p immediately before the k^{th} event occurs. The state s_i in the global state S corresponding to the cut C is that of p_i immediately after the last event processed by p_i in the cut – ei^{ci} . The set of events ei^{ci} is called the frontier of the cut.

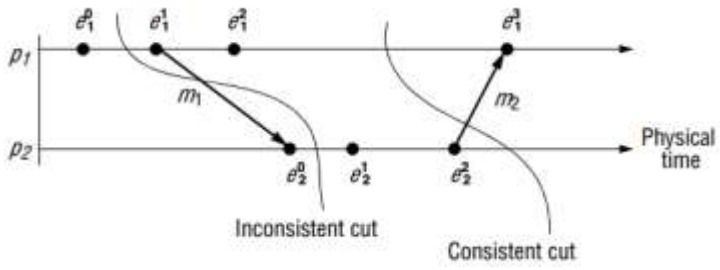


Fig : Types of cuts

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

Inconsistent global states: They are not meaningful in the sense that a distributed system can never be in an inconsistent state.

CUTS OF A DISTRIBUTED COMPUTATION

A cut is a set of cut events, one per node, each of which captures the state of the node on which it occurs.

Cut is pictorially a line slices the space–time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE. The PAST contains all the events to the left of the cut and the FUTURE contains all the events to theright of the cut. For a cut C, let PAST(C) and FUTURE(C) denote the set ofevents in the PAST and FUTURE of C, respectively.

Consistent cut: A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.

Inconsistent cut: A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST.

PAST AND FUTURE CONES OF AN EVENT

In a distributed computation, an event e_j could have been affected only by all events e_i, such that $e_i \rightarrow e_j$ and all the information available at e_i could be made accessible at e_j. In other word e_i and e_j should have a causal relationship. Let Past(e_j) denote all events in the past of e_j in any computation.

$$Past(e_j) = \{e_i | \forall e_i \in H,\ e_i \rightarrow e_j\}$$

The term max(past(e_i)) denotes the latest event of process p_i that has affected e_j. This will always be a message sent event.

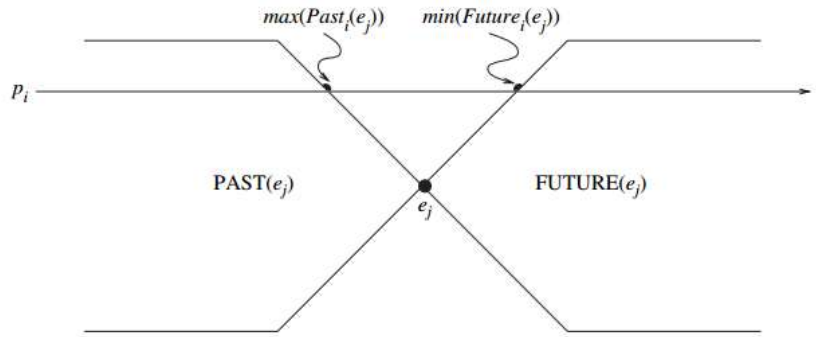


Fig : Past and future cones of event

A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE. A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.

The future of an event $e_j$denoted by Future(e_j) contains all the events e_i that are casually affected by e_j.

$$Future(e_j) = \{e_i | \forall e_i \in H,\ e_j \rightarrow e_i\}.$$

Future$_i$(e_i) is the set of those events of Future (e_j) are the process p_i and min(Future$_i$(e_j)) as the first event on process p_i that is affected by e_j. All events at a process p_i that occurred afterMax(Past(e_j)) but before min(Future$_i$(e_j)) are concurrent with e_j.

MODELS OF PROCESS COMMUNICATIONS

There are two basic models of process communications

Synchronous: The sender process blocks until the message has been received by the receiver process. The sender process resumes executiononly after it learns that the receiver process has accepted the message. The sender and the receiver processes must synchronize to exchange a message.

Asynchronous: It is non- blocking communication where the sender and the receiver do not synchronize to exchange a message. The sender process does not wait for the message to be delivered to the receiver process. The message is buffered by the system and is delivered to the receiver process when it is ready to accept the message. A buffer overflow may occur if a process sends a large number of messages in a burst to another process, thus causing a message burst.

Asynchronous communication achieves high degree of parallelism and non- determinism at the cost of implementation complexity with buffers. On the other hand, synchronization is simpler with low performance. The occurrence of deadlocks and frequent blocking of events prevents it from reaching higher performance levels.

PHYSICAL CLOCK SYNCHRONIZATION: NEWTWORK TIME PROTOCOL (NTP)

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

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

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

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

Basic terminologies:

If C_a and C_b are two different clocks, then:

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

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

Clocking Inaccuracies

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

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

ρ- maximum skew rate.

1. Offset delay estimation

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

2. Clock offset and delay estimation

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

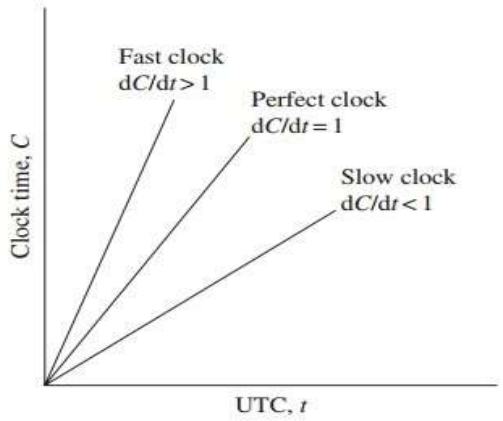
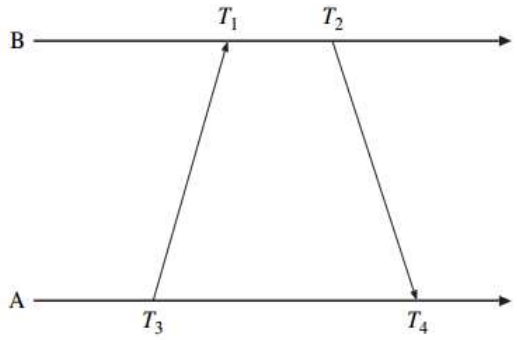


Fig : Behavior of clocks



Figa) Offset and delay estimation between processes from same server

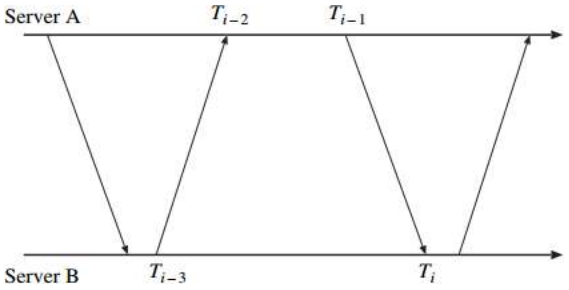


Fig b) Offset and delay estimation between processes from different servers

Let T1, T2, T3, T4 be the values of the four mostrecent timestamps. The clocks A and B are stable andrunning at the same speed. Let a = T1 − T3 and b = T2 − T4. If the networkdelay difference from A to B and from B to A, called **differential delay**, is small, the clock offset θand roundtrip delayδ of B relative to A at time T4are approximately given by the following:

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

Each NTP message includes the latest three timestamps T1, T2, andT3, while T4 is determined upon arrival.

SYNCHRONOUS VS ASYNCHRONOUS EXECUTIONS

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

Asynchronous Execution:

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

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

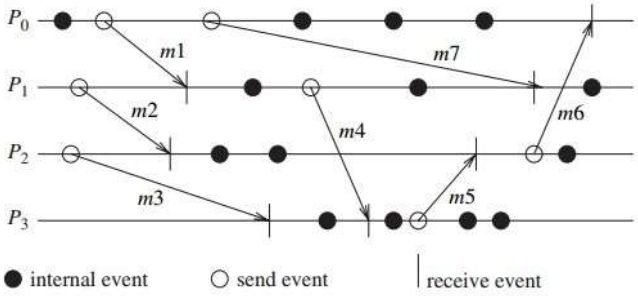


Fig : Asynchronous execution in message passing system

Synchronous Execution:

A communication among processes is considered synchronous when every process observes the same order of messages within the system. In the same manner, the execution is considered synchronous, when every individual process in the system observes the same total order of all the processes which happen within it. In an synchronous execution:

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

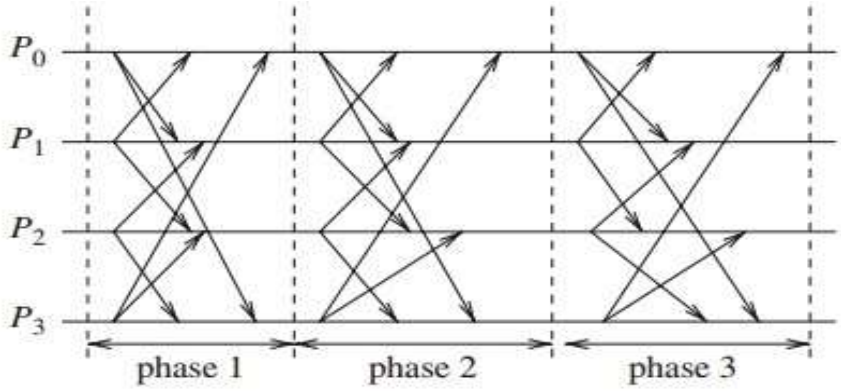


Fig: Synchronous execution

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

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

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

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

Emulation for a fault free system

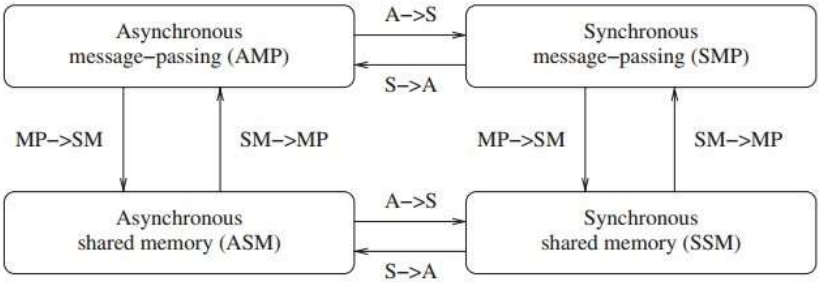


Fig : Emulations in a failure free message passing system

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

DESIGN ISSUES AND CHALLENGES IN DISTRIBUTED SYSTEMS

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

- Issues related to system and operating systems design

- Issues related to algorithm design
- Issues arising due to emerging technologies

The above three classes are not mutually exclusive.

Issues related to system and operating systems design

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

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

Examples: RPC, RMI

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

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

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

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

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

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

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

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

- **Access transparency:** hides differences in data representation
- **Location transparency:** hides differences in locations y providing uniform access to data located at remote locations.
- **Migration transparency:** allows relocating resources without changing names.
- **Replication transparency:** Makes the user unaware whether he is working on original or replicated data.
- **Concurrency transparency:** Masks the concurrent use of shared resources for the user.
- **Failure transparency**: system being reliable and fault-tolerant.

➢ **Scalability and modularity:** The algorithms, data and services must be as distributed as possible. Various techniques such as replication, aching and cache management, and asynchronous processing help to achieve scalability.

Algorithmic challenges in distributed computing

➢ **Designing useful execution models and frameworks**

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

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

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

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

- The geographically remote resources demands the synchronization based on logical time.

- Logical time is relative and eliminates the overheads of providing physical time for applications .Logical time can

(i) capture the logic and inter-process dependencies

(ii) track the relative progress at each process

- Maintaining the global state of the system across space involves the role of time dimension for consistency. This can be done with extra effort in a coordinated manner.
- Deriving appropriate measures of concurrency also involves the time dimension, as the execution and communication speed of threads may vary a lot.

➢ **Synchronization/coordination mechanisms**

- Synchronization is essential for the distributed processes to facilitate concurrent execution without affecting other processes.
- The synchronization mechanisms also involve resource management and concurrency management mechanisms.
- Some techniques for providing synchronization are:

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

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

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

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

✓ **Termination detection**: cooperation among the processes to detect the specific global state of quiescence.

✓ **Garbage collection:** Detecting garbage requires coordination among the processes.

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

- A group is a collection of processes that share a common context and collaborate on a common task within an application domain. Group management protocols are needed for group communication wherein processes can join and leave groups dynamically, or fail.

- The concurrent execution of remote processes may sometimes violate the semantics and order of the distributed program. Hence, a formal specification of the semantics of ordered delivery need to be formulated, and then implemented.

➢ **Monitoring distributed events and predicates**

- Predicates defined on program variables that are local to different processes are used for specifying conditions on the global system state.
- On-line algorithms for monitoring such predicates are hence important.
- An important paradigm for monitoring distributed events is that of event streaming, wherein streams of relevant events reported from different processes are examined collectively to detect predicates.
- The specification of such predicates uses physical or logical time relationships.

➢ **Distributed program design and verification tools**

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

➢ **Debugging distributed programs**

Debugging distributed programs is much harder because of the concurrency and replications. Adequate debugging mechanisms and tools are need of the hour.

➢ **Data replication, consistency models, and caching**

- Fast access to data and other resources is important in distributed systems.
- Managing replicas and their updates faces concurrency problems.
- Placement of the replicas in the systems is also a challenge because resources usually cannot be freely replicated.

➢ **World Wide Web design – caching, searching, scheduling**

- WWW is a commonly known distributed system.
- The issues of object replication and caching, pre fetching of objects have to be done on WWW also.
- Object search and navigation on the web are important functions in the operation of the web.

➢ **Distributed shared memory abstraction**

- A shared memory is easier to implement since it does not involve managing the communication tasks.

- The communication is done by the middleware by message passing.
- The overhead of shared memory is to be dealt by the middleware technology.
- Some of the methodologies that does the task of communication in shared memory distributed systems are:

✓ **Wait-free algorithms:** The ability of a process to complete its execution irrespective of the actions of other processes is wait free algorithm. They control the access to shared resources in the shared memory abstraction. They are expensive.

✓ **Mutual exclusion:** Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner. Only one process is allowed to execute the critical section at any given time. In a distributed system, shared variables or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion.

✓ **Register constructions:** Architectures must be designed in such a way that, registers allows concurrent access without any restrictions on the concurrency permitted.

➢ **Reliable and fault-tolerant distributed systems**

The following are some of the fault tolerant strategies:

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

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

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

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

✓ **Self-stabilizing systems:** All system executions have associated good(or legal) states and bad (or illegal) states; during correct functioning, the system makes transitions among the good states. A self-stabilizing algorithm guarantee to take the system to a good state even if a bad state were to arise due to some error. Self-stabilizing algorithms require some in-built redundancy to track additional variables of the state and do extra work.

✓ **Checkpointing and recovery algorithms:** Check pointing is periodically recording the current state on secondary storage so that, in case of a failure. The entire computation is not lost but can be recovered from one of the recently taken checkpoints. Check pointing in a distributed environment is difficult because if the checkpoints at the different processes are not coordinated, the local checkpoints may become useless because they are inconsistent with the checkpoints at other processes.

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

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

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

✓ **Computation migration**: The ability to relocate processes in order to perform a redistribution of the workload.

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

➢ **Real-time scheduling**

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

➢ **Performance**

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

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

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

Applications of distributed computing and newer challenges

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

➢ **Mobile systems**

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

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

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

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

➢ **Sensor networks**

- A sensor is a processor with an electro-mechanical interface that is capable of sensing physical parameters.

- They are low cost equipment with limited computational power and battery life. They are designed to handle streaming data and route it to external computer network and processes.
- They are susceptible to faults and have to reconfigure themselves.
- These features introduces a whole new set of challenges, such as position estimation and time estimation when designing a distributed system .

➢ **Ubiquitous or pervasive computing**

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

➢ **Peer-to-peer computing**

- Peer-to-peer (P2P) computing is computing over an application layer network where all interactions among the processors are at a same level.
- This is a form of symmetric computation against the client sever paradigm.
- They are self-organizing with or without regular structure to the network.
- Some of the key challenges include: object storage mechanisms, efficient object lookup, and retrieval in a scalable manner; dynamic reconfiguration with nodes as well as objects joining and leaving the network randomly; replication strategies to expedite object search; tradeoffs between object size latency and table sizes; anonymity, privacy, and security.

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

- The users in present day require only the information of interest.
- In a dynamic environment where the information constantly fluctuates there is great demand for
- **Publish:** an efficient mechanism for distributing this information
- **Subscribe:** an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information

- An efficient mechanism for aggregating large volumes of published information and filtering it as per the user's subscription filter.
- Content distribution refers to a mechanism that categorizes the information based on parameters.
- The publish subscribe and content distribution overlap each other.
- Multimedia data introduces special issue because of its large size.

➢ **Distributed agents**

- Agents are software processes or sometimes robots that move around the system to do specific tasks for which they are programmed.
- Agents collect and process information and can exchange such information with other agents.
- Challenges in distributed agent systems include coordination mechanisms among the agents, controlling the mobility of the agents, their software design and interfaces.

➢ **Distributed data mining**

- Data mining algorithms process large amount of data to detect patterns and trends in the data, to mine or extract useful information.
- The mining can be done by applying database and artificial intelligence techniques to a data repository.

➢ **Grid computing**

- Grid computing is deployed to manage resources. For instance, idle CPU cycles of machines connected to the network will be available to others.
- The challenges includes: scheduling jobs, framework for implementing quality of service, real-time guarantees, security.

➢ **Security in distributed systems**

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