R.M.K GROUP OF ENGINEERING INSTITUTIONS

R.M.K
GROUP OF INSTITUTIONS

R.M.K
GROUP OF
INSTITUTIONS

Please read this disclaimer before proceeding:

22CS401
Distributed and Cloud Computing

Department: AI&DS

Batch/Year: 2022-2026/II

Table of Contents

R.M.K GROUP OF INSTITUTIONS

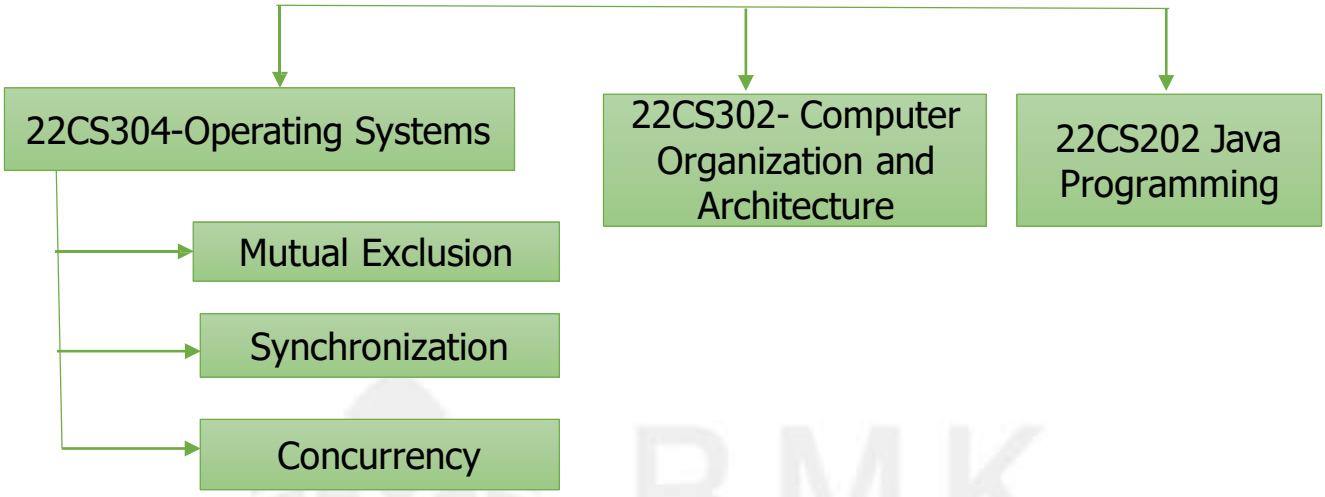
Course Objectives

Course Objectives

- To articulate the concepts and models underlying distributed computing
- To maintain consistency and perform efficient coordination in distributed systems through the use of logical clocks, global states, and snapshot recording algorithms.
- To learn different distributed mutual exclusion algorithms.
- To develop the ability to understand the cloud infrastructure and virtualization that help in the development of cloud.
- To explain the high-level automation and orchestration systems that manage the virtualized infrastructure

Pre Requisites

Prerequisites



Syllabus

22CS401 **DISTRIBUTED AND CLOUD COMPUTING** **L T P C 2 0 2 3**

Unit I : INTRODUCTION **6+6**

Definition - Relation to computer system components - Message-passing systems versus shared memory systems - Primitives for distributed communication - Synchronous versus asynchronous executions. A model of distributed computations: A distributed program - A model of distributed executions - Models of communication networks - Global state of a distributed system.

List of Exercise/Experiments:

1. Implement a simple distributed program that communicates between two nodes using Java's RMI (Remote Method Invocation) API.
2. Develop a distributed program that uses Java's messaging API (JMS) to communicate between nodes. Explore the different messaging paradigms (pub/sub, point-to-point) and evaluate their performance and scalability.
3. Develop a model of a distributed program using Java's concurrency and synchronization primitives.

Unit II : LOGICAL TIME, GLOBAL STATE, AND SNAPSHOT ALGORITHMS **6+6**

Logical time–Scalar Time–Vector Time-Efficient implementations of vector clocks–Virtual Time. Global state and snapshot recording algorithms: System model-Snapshot algorithms for FIFO channels and non-FIFO channels.

List of Exercise/Experiments:

1. Develop a program in Java that implements vector clocks to synchronize the order of events between nodes in a distributed system.
2. Implement a snapshot algorithm for recording the global state of the distributed system using vector clocks, for both FIFO and non-FIFO channels. Test the algorithm by recording snapshots at various points in the system's execution and analyzing the resulting global state.

Unit III : DISTRIBUTED MUTUAL EXCLUSION ALGORITHMS **6+6**

Introduction-Lamport's algorithm-Ricart–Agrawala algorithm-Quorum-based mutual exclusion algorithms-Maekawa's algorithm-Suzuki–Kasami's broadcast algorithm.

List of Exercise/Experiments:

1. Implement Lamport's algorithm for mutual exclusion in a distributed system using Java's RMI API.
2. Develop a program in Java that implements Maekawa's algorithm for mutual exclusion in a distributed system.
3. Implement Suzuki-Kasami's broadcast algorithm in Java to achieve reliable message delivery in a distributed system.

Unit IV : CLOUD INFRASTRUCTURE AND VIRTUALIZATION **6+6**

Data Center Infrastructure and Equipment – Virtual Machines – Containers – Virtual Networks - Virtual Storage.

List of Exercise/Experiments:

1. Set up a virtualized data center using a hypervisor like VMware or VirtualBox and create multiple virtual machines (VMs) on it. Configure the VMs with different operating systems, resources, and network configurations, and test their connectivity and performance.
2. Deploy a containerized application on a virtual machine using Docker or Kubernetes.

Unit V : AUTOMATION AND ORCHESTRATION **6+6**

Automation - Orchestration: Automated Replication and Parallelism - The MapReduce Paradigm: The MapReduce Programming Paradigm – Splitting Input – Parallelism and Data size – Data access and Data Transmission – Apache Hadoop – Parts of Hadoop – HDFS Components – Block Replication and Fault Tolerance – HDFS and MapReduce - Microservices.

List of Exercise/Experiments:

1. Set up and configure a single-node Hadoop cluster.
2. Run the word count program in Hadoop.
3. Deploy a microservices architecture using a container orchestration tool like Kubernetes or Docker Swarm.

Course Outcomes

Course Outcomes

CO#	COs	K Level
CO1	Articulate the main concepts and models underlying distributed computing.	K3
CO2	Learn how to maintain consistency and perform efficient coordination in distributed systems through the use of logical clocks, global states, and snapshot recording algorithms.	K2
CO3	Learn different distributed mutual exclusion algorithms	K2
CO4	Develop the ability to understand the cloud infrastructure and virtualization that help in the development of cloud	K3
CO5	Explain the high-level automation and orchestration systems that manage the virtualized infrastructure.	K2

Knowledge Level	Description
K6	Evaluation
K5	Synthesis
K4	Analysis
K3	Application
K2	Comprehension
K1	Knowledge

CO – PO/PSO Mapping

CO – PO /PSO Mapping Matrix

CO #	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2	PS0 3
CO1	3	3	2	1	1	-	-	2	2	2	-	2	3	2	-
CO2	3	2	2	2	2	-	-	2	2	2	-	2	3	2	-
CO3	3	3	2	2	2	-	-	2	2	2	-	2	3	2	-
CO4	3	2	2	2	2	-	-	2	2	2	-	2	3	3	-
CO5	3	2	2	2	2	-	-	2	2	2	-	2	3	2	-

Lecture Plan

Unit I

R.M.K
GROUP OF
INSTITUTIONS

Lecture Plan - Unit 1

Sl. No.	Topic	Number of Periods	Proposed Date	Actual Lecture Date	CO	Taxonomy Level	Mode of Delivery
1	Definition - Relation to computer system components	1	03.01.24	03.01.24	CO1		Chalk & Talk
2	Message-passing systems versus shared memory systems - Primitives for distributed communication	1	06.01.24	06.01.24	CO1		Chalk & Talk
3	Synchronous versus asynchronous executions	1	09.01.24	09.01.24	CO1		Chalk & Talk
4	A model of distributed computations: A distributed program	1	10.01.24	10.01.24	CO1		Chalk & Talk
5	A model of distributed executions - Models of communication networks	1	22.01.24	22.01.24	CO1		Chalk & Talk
6	Global state of a distributed system	1	24.01.24	24.01.24	CO1		Chalk & Talk

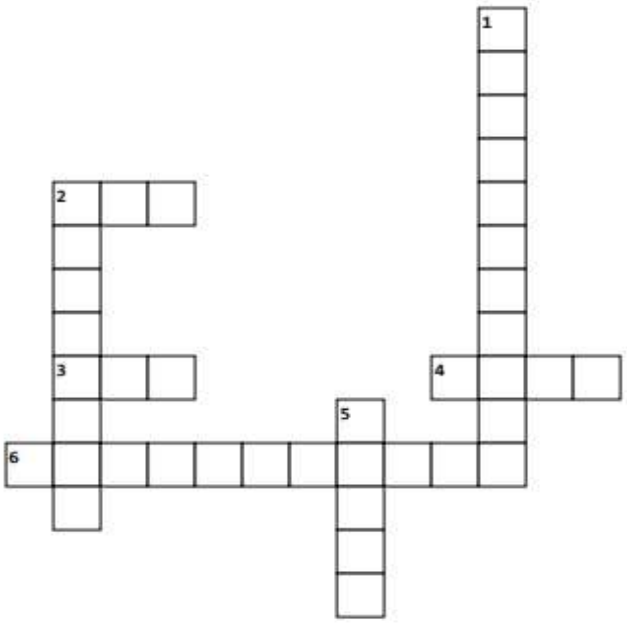
Activity Based Learning

Unit I

R.M.K
GROUP OF
INSTITUTIONS

1.Live assessment using RMK NEXTGEN

2.Cross Word Puzzle



Across

2. COMMERCIAL MIDDLEWARE

3. MEMORY ACCESS TIME IS SAME FOR ALL PROCESSOR

4. PROCESSOR EXECUTING MULTIPLE INSTRUCTIONS AND DATA

6. SYSTEM TO ADAPT THE INCREASED SERVICE LOAD

Down

1. AMOUNT OF COMPUTATION/COMMUNICATION

2. DISTRIBUTED SYSTEM MUST HAVE BETTER

5. DISTRIBUTED SYSTEM HAVE NO PHYSICAL

Lecture Notes – Unit I

R.M.K
GROUP OF
INSTITUTIONS

Unit I INTRODUCTION

Definition

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

A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system. Typically the computers are semi-autonomous and are loosely coupled while they cooperate to address a problem collectively

A collection of independent computers that appears to the users of the system as a single coherent computer

A wide range of computers, from weakly coupled systems such as wide-area networks, to strongly coupled systems such as local area networks, to very strongly coupled systems such as multiprocessor systems.

Features

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

No shared memory-Key feature that requires message-passing for communication.

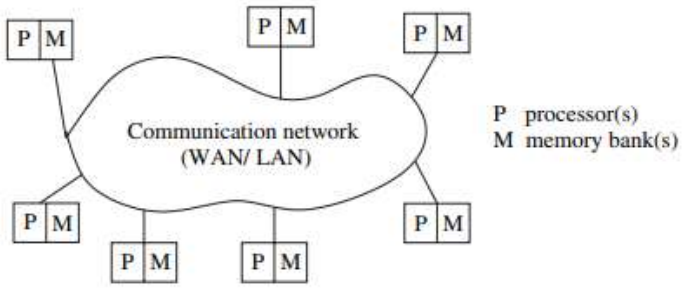
Geographical separation-The network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system. This NOW configuration is becoming popular because of the low-cost high-speed off-the-shelf processors now available. The Google search engine is based on the NOW architecture.

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.

Unit I INTRODUCTION

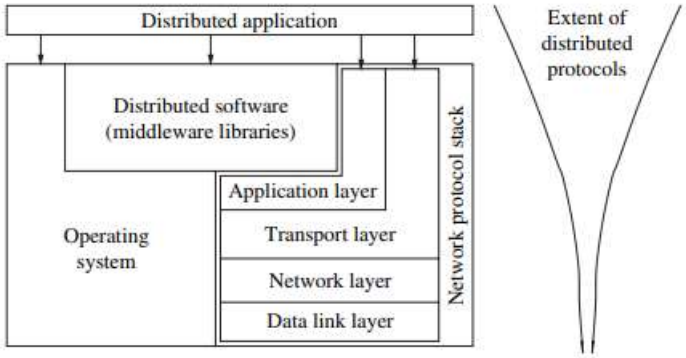
Relation to computer system components

A typical distributed system is shown in the figure. Each computer has a memory-processing unit and the computers are connected by a communication network.



A distributed system connects processors by a communication network

The figure below shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning. The distributed software is also termed as middleware. A distributed execution is the execution of processes across the distributed system to collaboratively achieve a common goal. An execution is also sometimes termed a computation or a run. The distributed system uses a layered architecture to break down the complexity of system design. The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level



Interaction of the software components at each processor

The middleware layer does not contain the traditional application layer functions of the network protocol stack, such as http, mail, ftp, and telnet. Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program code

There 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.

Motivation for using Distributed Systems

- Inherently distributed computations
- Resource sharing
- Access to geographically remote data and resources
- Enhanced reliability
- Increased performance/cost ratio
- Scalability
- Modularity and incremental expandability

Characteristics of parallel systems

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

1. A **multiprocessor system** is a parallel system in which the multiple processors have direct access to shared memory which forms a common address space. Such processors usually do not have a common clock. A multiprocessor system usually corresponds to a uniform memory access (UMA) architecture in which the access latency, i.e., waiting time, to complete an access to any memory location from any processor is the same.

The processors are in very close physical proximity and are connected by an interconnection network. Inter-process communication across processors is traditionally through read and write operations on the shared memory and message-passing primitives. All the processors usually run the same operating system, and both the hardware and software are very tightly coupled.

The architecture is shown in Figure 1.3 (a).

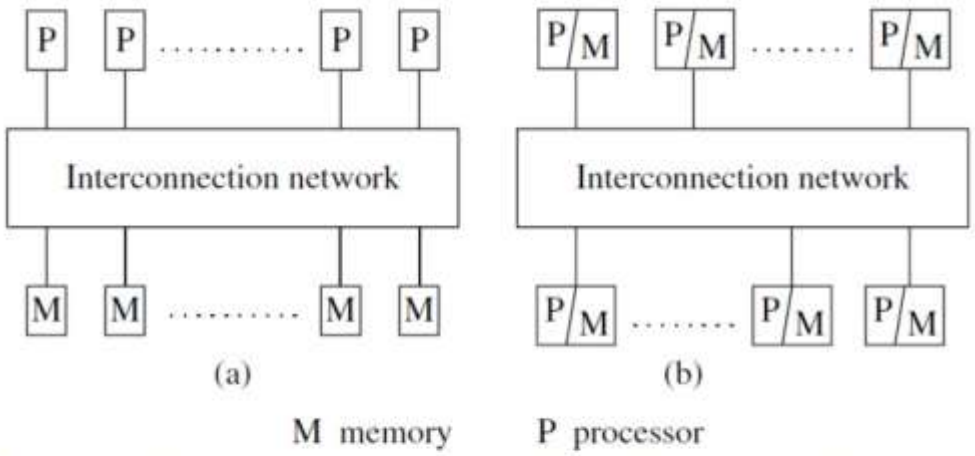


Figure 1.3 Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.

The processors are usually of the same type, and are housed within the same box/container with a shared memory. The interconnection network to access the memory may be a bus, it is usually a multistage switch with a symmetric and regular design.

2. A multicomputer parallel system is a parallel system in which the multiple processors do not have direct access to shared memory. The memory of the multiple processors may or may not form a common address space. Such computers usually do not have a common clock.

3. Array processors belong to a class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock.

Flynn's taxonomy

• **Single instruction stream, single data stream (SISD)** This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.

• **Single instruction stream, multiple data stream (SIMD)** This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items. Applications that involve operations on large arrays and matrices, such as scientific applications, can best exploit systems that provide the SIMD mode of operation because the data sets can be partitioned easily.

- **Multiple instruction stream, single data stream (MISD)** This mode corresponds to the execution of different operations in parallel on the same data. This is a specialized mode of operation with limited but niche applications, e.g., visualization.

- **Multiple instruction stream, multiple data stream (MIMD)** In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems. There is no common clock among the system processors.

Coupling: The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the interdependency and binding and/or homogeneity among the modules. When the degree of coupling is high (low), the modules are said to be tightly (loosely) coupled. SIMD and MISD architectures generally tend to be tightly coupled.

Parallelism or speedup of a program on a specific system

This is a measure of the relative speedup of a specific program, on a given machine. The speedup depends on the number of processors and the mapping of the code to the processors. It is expressed as the ratio of the time T1 with a single processor, to the time Tn 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 (either via shared memory or message-passing) operations to complete.

Concurrency

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

Granularity of a program

The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as granularity. If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions, compared to the number of times the processors communicate either via shared memory or message passing and wait to get synchronized with the other processors.

Unit I INTRODUCTION

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. For a distributed system, this abstraction is called distributed shared memory.

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. 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. This emulation is expensive. The latencies involved in read and write operations may be high.

Within the multiprocessor system, the processors communicate via shared memory. Between two computers, the communication is by message passing.

Unit I INTRODUCTION

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.

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:

- **Buffered option**
- **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.

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.

Blocking primitives A primitive is blocking if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.

Non-blocking primitives A primitive is non-blocking if control returns back to the invoking process immediately after invocation, even though the operation has not completed. For a non-blocking Send, control returns to the process even before the data is copied out of the user buffer. For a non-blocking Receive, control returns to the process even before the data may have arrived from the sender.

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

After issuing the primitive in non-blocking mode, the process has done whatever actions it could and now needs to know the status of completion of the call, therefore using a blocking Wait() call is usual programming practice.

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.

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

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

There are four versions of the Send primitive

Synchronous blocking

Synchronous non-blocking

Asynchronous blocking

Asynchronous non-blocking

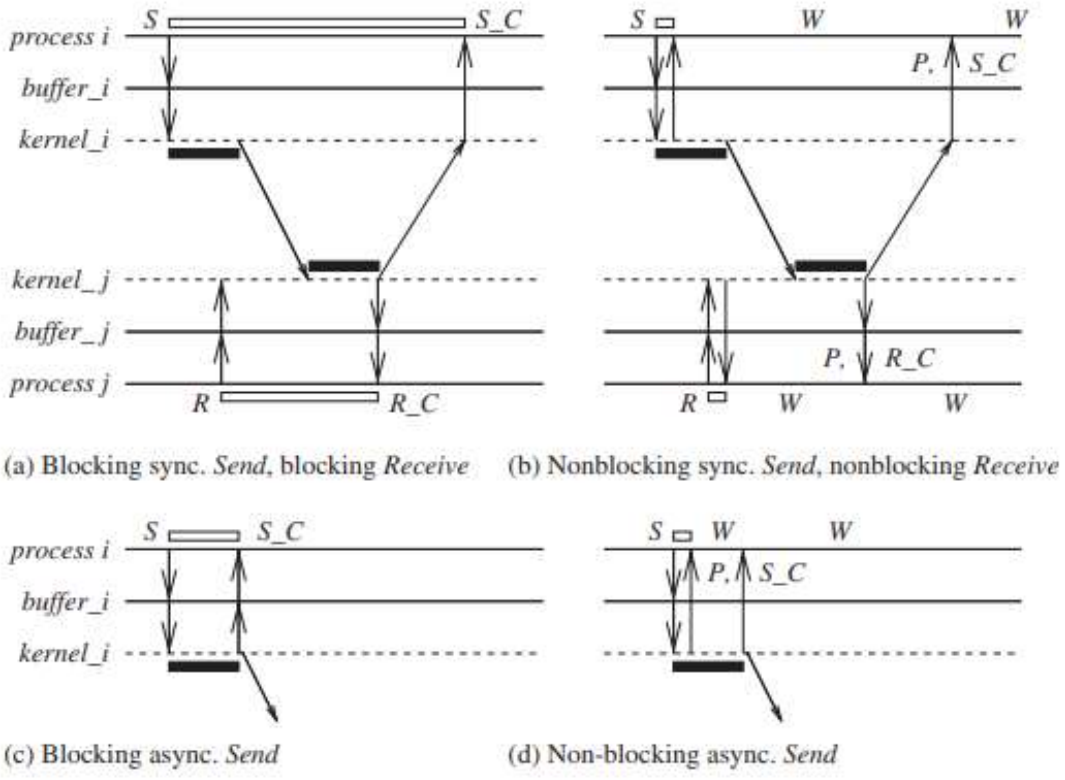
For the Receive primitive, there are blocking synchronous and non-blocking synchronous versions. These versions of the primitives are illustrated in Figure.

Three time lines are shown for each process:

(1) for the process execution

(2) for the user buffer from/to which data is sent/received

(3) for the kernel/communication subsystem.



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

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

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

Blocking Receive - Figure (a) 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 - Figure (b) 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.

- A synchronous Send lowers the efficiency within process Pi.
- The non-blocking asynchronous Send is useful when a large data item is being sent because it allows the process to perform other instructions in parallel with the completion of the Send.
- The non-blocking synchronous Send avoids the potentially large delays for handshaking, particularly when the receiver has not yet issued the Receive call.
- The non-blocking Receive is useful when a large data item is being received and/or when the sender has not yet issued the Send call, because it allows the process to perform other instructions in parallel with the completion of the Receive.
- Note that if the data has already arrived, it is stored in the kernel buffer, and it may take a while to copy it to the user buffer specified in the Receive call

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.

Synchronous versus asynchronous executions

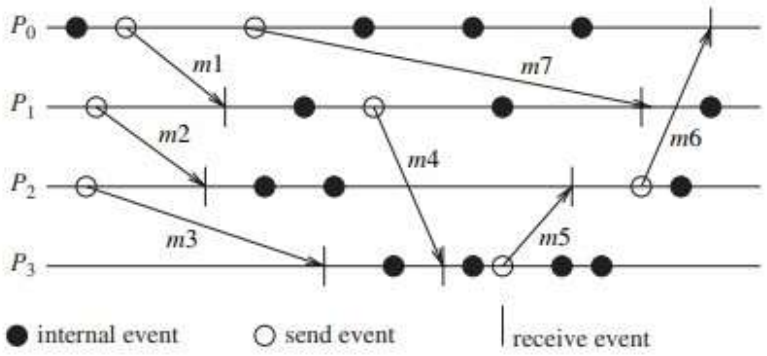
An asynchronous execution is an execution in which

(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.

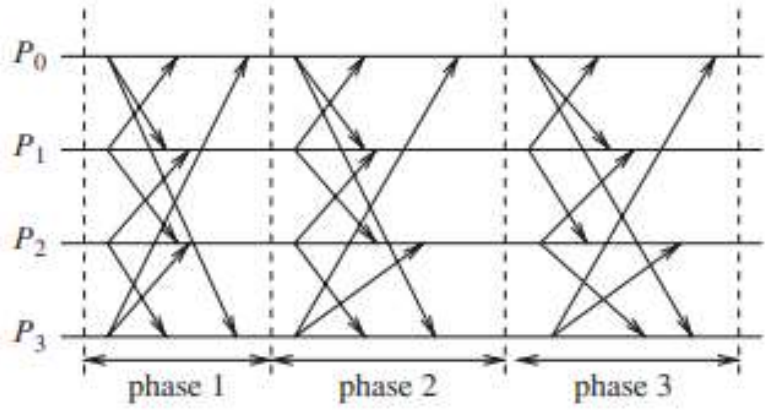


A synchronous execution is an execution in which

(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.



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.

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

Unit I INTRODUCTION

A model of distributed computations

A distributed program

A distributed program is composed of a set of n asynchronous processes p_1, p_2, , p_k,.. , p_n that communicate by message passing over the communication network.

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 p_i to process p_j and let m_{ij} denote a message sent by p_i 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.

A model of distributed executions

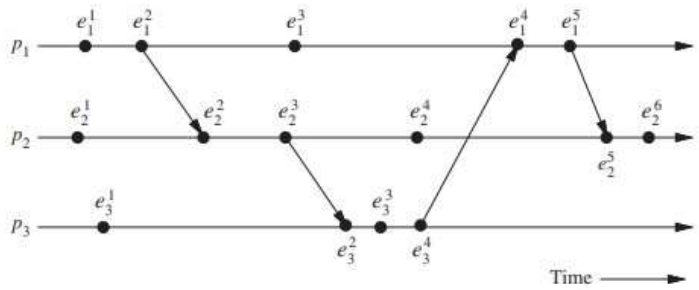
The execution of a process consists of a sequential execution of its actions. The actions are atomic and the actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events. Let $e_{i\ x}$ denote the xth event at process p_i. For a message m, let send(m) and rec(m) denote its send and receive events, respectively.

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 $e_{i\ 1}$, $e_{i\ 2}$,… , $e_{i\ x}$, $e_{i\ x+1}$,

and is denoted by H_i

$H_i = h_i \rightarrow i$ where hi is the set of events produced by pi 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.

For every message m that is exchanged between two processes, we have send(m) → msg rec(m) Relation →msg defines causal dependencies between the pairs of corresponding send and receive events 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, it is justified to denote it as a dot on a process line. In this figure, for process p1, the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.



Causal precedence relation

The execution of a distributed application results in a set of distributed events produced by the processes. Let H =∪$_i$h$_i$ denote the set of events executed in a distributed computation. Next, we define a binary relation on the set H, denoted as →, that expresses causal dependencies between events in the distributed execution

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

any two events e$_i$ and e$_j$, if e$_i$ → e$_j$, then event e$_j$ is directly or transitively dependent on event e$_i$; graphically, it means that there exists a path consisting of message arrows and process-line segments in the space–time diagram that starts at e$_i$ and ends at e$_j$. Note that relation → denotes flow of information in a distributed computation and e$_i$ → e$_j$ dictates that all the information available at e$_i$ is potentially accessible at e$_j$

For any two events e_i and e_j, $e_i \rightarrow e_j$ denotes the fact that event e_j does not directly or transitively dependent on event e_i. That is, event e_i does not causally affect event e_j. Event e_j is not aware of the execution of e_i or any event executed after e_i on the same process.

- for any two events e_i and e_j, $e_i \rightarrow e_j \Rightarrow e_j \rightarrow e_i$
- for any two events e_i and e_j , $e_i \rightarrow e_j \Rightarrow e_j \rightarrow e_i$.

For any two events e_i and e_j , if $e_i \rightarrow e_j$ and $e_j \rightarrow e_i$, then events e_i and e_j are said to be concurrent and the relation is denoted as $e_i \parallel e_j$.

For any two events e_i and ej in a distributed execution, $e_i \rightarrow e_j$ or $e_j \rightarrow e_i$, or $e_i \parallel e_j$.

Logical vs. physical concurrency

In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other. Physical concurrency is that the events occur at the same instant in physical time.

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. The "causal ordering" model is based on Lamport's "happens before" relation.

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

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

Global state of a distributed system

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. But both are impossible.

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.

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:

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

In the distributed execution, a global state GS_1 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 $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all the channels are empty except C_{21} that contains message m_{21}.

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

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

A global state is strongly consistent iff it is transitless as well as consistent.

Assignments

Assignments

1. Compare and contrast a parallel system and a distributed system? (CO1,K4)
2. Organize the challenges of a distributed system (CO1,K4)
3. Discuss the design requirements of a distributed system (CO1,K2)
4. Design an Omega network for 8 processors and memory (CO1,K6)
5. Design a Butterfly network for 8 processors and memory (CO1,K6)

Part A – Q & A

Unit - I

PART - A Questions

1. Define Distributed System? (CO1, K1)

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

2. List out the features of distributed systems. (CO1,K1)

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

3. How a distributed system is characterized? (CO1,K1)

A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system.

4. Differentiate loosely coupled and tightly coupled systems. (CO1,K4)

A multiprocessor is that which contains more than two processors in a system. A system is known as a loosely connected multiprocessor if there is a very low degree of coupling between these CPUs. Every CPU has its local memory, collection of input-output devices.

A tightly coupled system is a system architecture and computing method in which all hardware and software components are linked together so that every component is dependent on the others. Tightly coupled system architecture encourages application and code interdependence.

5. Give examples for loosely coupled and tightly coupled systems. (CO1,K1)

Loosely coupled systems: wide-area networks

Tightly coupled systems: local area networks and multiprocessor systems

6. What is NOW/COW architecture? (CO1,K1)

The network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system. This NOW configuration is becoming popular because of the low-cost high-speed off-the-shelf processors now available. The Google search engine is based on the NOW architecture.

PART - A Questions

7. What is middleware? (CO1 , K1)

The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level. middleware layer does not contain the traditional application layer functions of the network protocol stack, such as http, mail, ftp, and telnet. Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program code.

8. What is distributed execution? (CO1,K1)

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.

9. What is computation or a run? (CO1,K1)

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.

10. List out the middleware standards in distributed systems. (CO1, K1)

There are several standards such as Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Remote Procedure Call (RPC) mechanism, DCOM (Distributed Component Object Model), Java, and RMI (Remote Method Invocation) and message-passing interface (MPI).

11. Sketch the working of RPC mechanism. (CO1, K3)

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.

12. What is distributed shared memory? (CO1,K1)

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. For a distributed system, this abstraction is called distributed shared memory.

PART - A Questions

13. What is the motivation for using Distributed Systems. (CO1,K1)

- Inherently distributed computations
- Resource sharing
- Access to geographically remote data and resources
- Enhanced reliability
- Increased performance/cost ratio
- Scalability
- Modularity and incremental expandability

14. What is shared memory (CO1,K1)

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.

15. Why message passing is required in distributed systems? (CO1,K1)

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.

16. Differentiate shared memory and message passing (CO1,K4)

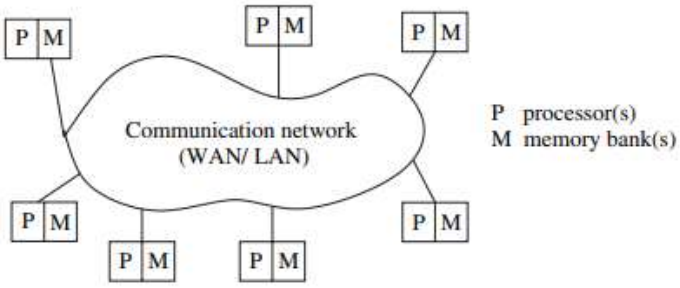
Shared Memory	Message Passing
used to communicate between the single processor and multiprocessor systems	most commonly utilized in a distributed setting when communicating processes are spread over multiple devices linked by a network.
system calls are only required to establish the shared memory	performed via the kernel
The code for reading and writing the data from the shared memory should be written explicitly by the developer	no such code is required in this case because the message passing feature offers a method for communication and synchronization of activities executed by the communicating processes

17. What is resource sharing? (CO1,K1)

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. 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.

PART - A Questions

18. Draw a block diagram of distributed system. (CO1,K1)



19. What is Concurrency? (CO1,K1)

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

20. What is Granularity? (CO1,K1)

The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as granularity. If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions, compared to the number of times the processors communicate either via shared memory or message-passing and wait to get synchronized with the other processors.

21. Define SISD (CO1,K1)

This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.

22. For which applications MISD is suitable? (CO1,K1)

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

23. Define SIMD (CO1,K1)

This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items. Applications that involve operations on large arrays and matrices, such as scientific applications, can best exploit systems that provide the SIMD mode of operation because the data sets can be partitioned easily.

24. Define MIMD (CO1,K1)

In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems. There is no common clock among the system processors. Sun Ultra servers, multicomputer PCs, and IBM SP machines are examples of machines that execute in MIMD mode.

PART - A Questions

25. List out different types of parallel systems. (CO1,K1)

- A multiprocessor system
- A multicomputer parallel system
- Array processors

26. Define Array processors. (CO1,K1)

Array processors belong to a class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock.

27. When two events are said to be logically concurrent and physically concurrent? (CO1,K1)

Two events are logically concurrent if and only if they do not causally affect each other. Physical concurrency has a connotation that the events occur at the same instant in physical time.

28. When a global state is said to be consistent? (CO1,K1)

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:

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

29. What do you mean by transitless? (CO1,K1)

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

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

30. When we say that a global state is strongly consistent? (CO1,K1)

A global state is strongly consistent iff it is transitless as well as consistent.

Part B – Questions

Part-B Questions

Q. No.	Questions	CO Level	K Level
1	Explain the characteristics of distributed systems	CO1	K2
2	List the features of distributed systems	CO1	K2
3	Summarize the distributed computer system components	CO1	K2
4	Discuss the primitives for distributed communication	CO1	K2
5	Explain about the synchronous versus asynchronous executions in a message-passing system with examples.	CO1	K2
6	Explain the characteristics of parallel systems	CO1	K2
7	What are the processing modes of Flynn taxonomy? Examine various MIMD architectures in terms of coupling.	CO1	K4
8	Explain the model of distributed execution	CO1	K2
9	Discuss about the global state in a distributed system	CO1	K2

Supportive online Certification courses (NPTEL, Swayam, Coursera, Udemy, etc.,)

R.M.K GROUP OF INSTITUTIONS

Supportive Online Certification Courses

Sl. No.	Courses	Platform
1	Cloud Computing and Distributed Systems	NPTEL

Real time Applications in day to day life and to Industry

R.M.K GROUP OF INSTITUTIONS

Real Time Applications

1. Game Development
2. Computer Graphics
3. Tele communication network
4. Streaming Media

Content Beyond Syllabus

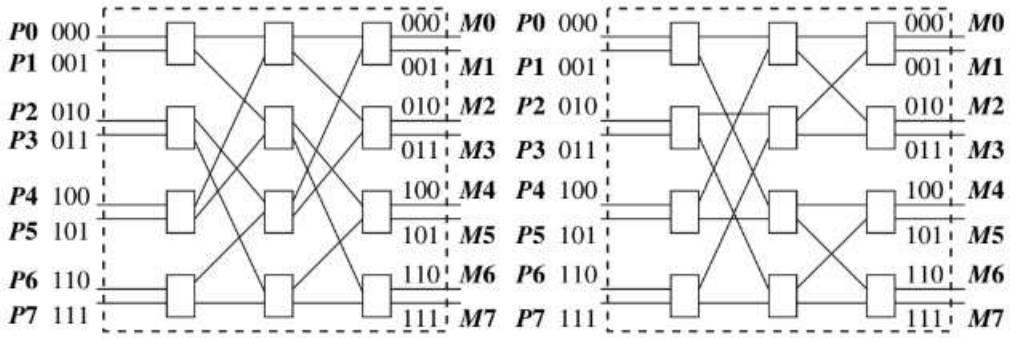
R.M.K
GROUP OF
INSTITUTIONS

Unit I Content Beyond Syllabus

Omega and Butterfly interconnection Networks

Two popular interconnection networks – the Omega network and the Butterfly network. It is a multi-stage network formed of 2×2 switching elements. Each 2×2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire.

In a single step, however, only one data unit can be sent on an output wire. So if the data from both the input wires is to be routed to the same output wire in a single step, there is a collision. Various techniques such as buffering or more elaborate interconnection designs can address collisions.



(a) 3-stage Omega network ($n = 8$, $M = 4$) (b) 3-stage Butterfly network ($n = 8$, $M = 4$)

Interconnection networks for shared memory multiprocessor systems.

Each 2 ×2 switch is represented as a rectangle in the figure. A n-input and n-output network uses log n stages and log n bits for addressing. Routing in the 2 × 2 switch at stage k uses only the kth bit, and hence can be done at clock speed in hardware. The multi-stage networks can be constructed recursively, and the interconnection pattern between any two stages can be expressed using an iterative or a recursive generating function.

Omega interconnection function The Omega network which connects n processors to n memory units has n/2 $\log_2 n$ switching elements of size 2 × 2 arranged in $\log_2 n$ stages. Between each pair of adjacent stages of the Omega network, a link exists between output i of a stage and the input j to the next stage according to the following perfect shuffle pattern which is a left-rotation operation on the binary representation of i to get j. The iterative generation function is as follows:

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

Content Beyond Syllabus

Omega routing function The routing function from input line i to output line j considers only j and the stage number s, where s ∈ [0, $\log_2 n-1$]. In a stage s switch, if the s +1th MSB (most significant bit) of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

Butterfly interconnection function The generation of the interconnection pattern between a pair of adjacent stages depends not only on n but also on the stage number s. The recursive expression is as follows. Let there be M = n/2 switches per stage, and let a switch be denoted by the tuple <x, s>, where x ∈ [0, M −1] and stage s ∈ [0, $\log_2 n-1$].

The two outgoing edges from any switch x s are as follows. There is an edge from switch <x, s> to switch <y, s+1> if (i) x = y or (ii) x XOR y has exactly one 1 bit, which is in the s+1th MSB. For stage s, apply the rule above for $M/2^s$ switches.

Consider the Butterfly network in Figure 1.4(b), n = 8 and M = 4. There are three stages, s = 0 1 2, and the interconnection pattern is defined between s = 0 and s = 1 and between s = 1 and s = 2. The switch number x varies from 0 to 3 in each stage, i.e., x is a 2-bit string. Consider the first stage interconnection (s = 0) of a butterfly of size M, and hence having $\log_2 2M$ stages. For stage s = 0, as per rule (i), the first output line from switch 00 goes to the input line of switch 00 of stage s = 1. As per rule (ii), the second output line of switch 00 goes to input line of switch 10 of stage s = 1. Similarly, x = 01 has one output line go to an input line of switch 11 in stage s = 1.

For stage s = 1 connecting to stage s = 2, we apply the rules considering only $M/2^1 = M/2$ switches, i.e., we build two butterflies of size M/2 – the "upper half" and the "lower half" switches. The recursion terminates for $M/2^s = 1$, when there is a single switch.

Butterfly routing function 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.

Assessment Schedule (Proposed Date & Actual Date)

R.M.K GROUP OF INSTITUTIONS

Assessment Schedule

Assessment Tool	Proposed Date	Actual Date	Course Outcome	Program Outcome (Filled Gap)
Assessment I	10.02.2024		CO1, CO2	
Assessment II	01.04.2024		CO3, CO4	
Model	20.04.2024		CO1, CO2, CO3, CO4, CO5	

Prescribed Text Books & Reference

Text & Reference Books

Sl. No.	Book Name & Author	Book
1	Ajay D. Kshemkalyani, Mukesh Singhal, "Distributed Computing: Principles, Algorithms, and Systems", Cambridge University Press, 2011.	Text Book
2	Douglass E. Comer, "The Cloud Computing Book: The future of computing explained", CRC Press, 2021.	Text Book
3	Arshdeep Bahga, Vijay Madisetti, "Cloud Computing: A Hands-on Approach", Universities Press Private Limited, 2014.	Reference Book
4	Rajkumar Buyya, Christian Vecchiola, S. ThamaraiSelvi, "Mastering Cloud Computing", Tata Mcgraw Hill, 2017.	Reference Book
5	Kai Hwang, Geoffrey C. Fox, Jack G. Dongarra, "Distributed and Cloud Computing, From Parallel Processing to the Internet of Things", Morgan Kaufmann Publishers, 2012.	Reference Book
6	Hagit Attiya, Jennifer Welch, "Distributed Computing: Fundamentals, Simulations and Advanced Topics", John Wiley & Sons, Inc., 2004.	Reference Book
7	http://nptel.ac.in/	Reference Book

Mini Project Suggestions

R.M.K
GROUP OF
INSTITUTIONS

Mini Project

1. A user arrives at a railway station that she has never visited before, carrying a PDA that is capable of wireless networking. Suggest how the user could be provided with information about the local services and amenities at that station, without entering the station's name or attributes. What technical challenges must be overcome? Discuss in detail.
2. In a client server model that is implemented by using a simple RPC mechanism, after making an RPC request ,a client keep waiting until a reply is received from the server for its request. It would be more efficient to allow the client to perform other jobs while the server is processing its request. Develop a mechanism that may be used in this case to allow a client to perform other jobs while the server is processing its request.
3. The Project deals with the management of the occasion cars at the Dealer showroom by Client- Server application. The application must have option for the registration of the new cars and its sales receipt. This information or data is next transferred to the Server by implementing the Car and Receipt objects. The Server has to produce every new information.
4. Create a miniproject using RMI concept to perform deposit and withdrawal from an account..
5. Write a miniproject to perform deposit and withdrawal in an account using synchronized block/method.

Thank you

Disclaimer: