Experiment 01: Study of Distributed Computing system architecture and explain with various application like university, Banking system.

Learning Objective: Student should be able to write case study of Distributed Computing system architecture and explain with various application like university, Banking system.

Tools: Microsoft Word.

Theory:

Distributed computing system refers to a network of interconnected computers that work together to achieve a common goal. Unlike traditional centralized systems, where a single powerful machine handles all tasks, distributed systems distribute the workload across multiple nodes. This decentralized approach brings various advantages, including improved reliability, scalability, and fault tolerance.

Key Characteristics:

1. Decentralization:

In a distributed computing system, there is no central point of control. Instead, tasks and responsibilities are distributed among multiple nodes.

2. Concurrency:

Multiple tasks can be executed simultaneously across different nodes, enhancing overall system performance and efficiency.

3. Intercommunication:

Nodes within the system communicate and coordinate with each other to share information and collaborate on tasks.

4. Fault Tolerance:

Distributed systems are designed to be resilient to failures. If one node fails, the system can continue functioning using other available nodes.

5. Scalability:

Distributed systems can easily scale by adding or removing nodes, allowing for efficient handling of varying workloads.

6. Resource Sharing:

Resources such as processing power, storage, and memory are shared among nodes, optimizing resource utilization.

7. Heterogeneity:

Nodes in a distributed system can have different hardware configurations, operating systems, and software platforms, promoting flexibility.

Types of Architecture of Distributed Computing Systems:

1. Client-Server Architecture:

In this architecture, the system is divided into two main components: clients and servers. Clients request services or resources, and servers fulfill these requests. This model facilitates centralized control and resource management.

2. Peer-to-Peer (P2P) Architecture:

P2P architecture allows nodes to act both as clients and servers. Each node has equal status and can request services or provide resources. This type of architecture is known for its decentralized and self-organizing nature.

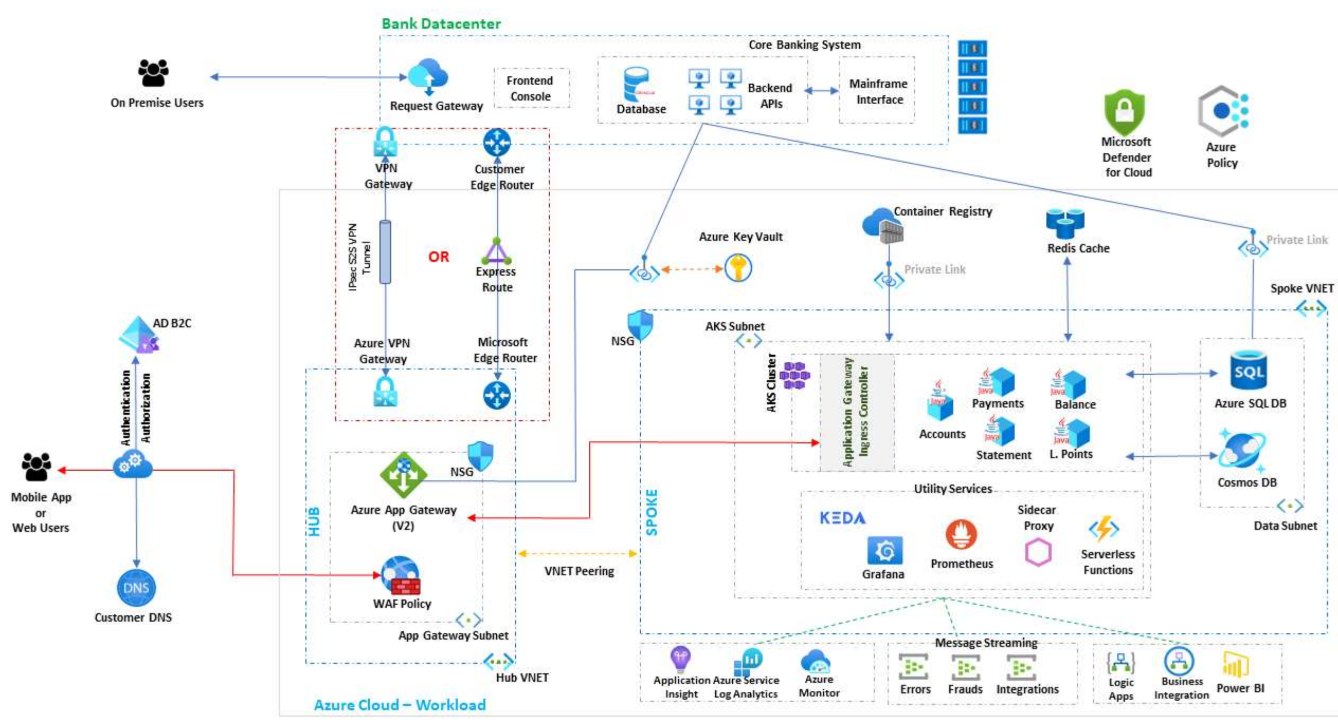
3. Three-Tier Architecture:

This architecture divides the system into three main layers: presentation, logic, and data. The presentation layer handles user interface, the logic layer processes requests, and the data layer manages storage and retrieval of data. This model enhances scalability and maintainability.

4. Microservices Architecture:
In a microservices architecture, the system is composed of small, independent services that communicate with each other through well-defined APIs. This approach promotes modularity, flexibility, and easy scalability.

5. Clustered Architecture:
Clustered architecture involves grouping multiple interconnected computers to work together as a single system. Nodes within a cluster share resources and collaborate on tasks, providing high availability and fault tolerance.

Distributed Computing Architecture of Banking System



Result and Discussion:

• Explain Distributed Computing Architecture of University
• Explain Distributed Computing Architecture of Banking

Learning Outcomes: Students should have the ability to

LO1: Understand the basics of Architecture of Distributed Computing.

LO2: Studied Architecture of University & banking.

Course Outcomes: Upon completion of the course students will be able to create architecture of Distributed Computing

Conclusion: Distributed computing systems play a crucial role in modern computing, providing solutions to complex problems through collaboration and resource sharing. Understanding the different architectures helps in designing systems that meet specific requirements, whether it be scalability, fault tolerance, or efficient resource utilization.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				

Experiment 2 – RPC/RMI

Learning Objective: Student should be able to Built a Program for Client/server using RPC/RMI

Tools : Java

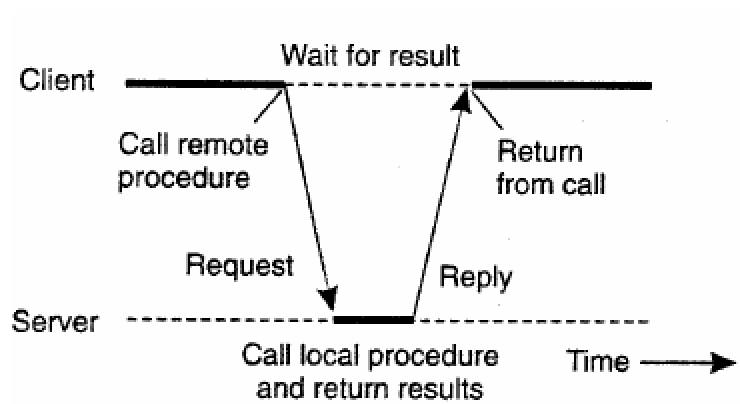
Theory:

Remote Procedure call

A remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

It further aims at hiding most of the intricacies of message RPC allows programs to call procedures located on other machines. But the procedures 'send' and 'receive' do not conceal the communication which leads to achieving access transparency in distributed systems.

Example: when process A calls a procedure on B, the calling process on A is suspended and the execution of the called procedure takes place. (PS: function, method, procedure difference, stub, 5 state process model definition)Information can be transported in the form of parameters and can come back in procedure result. No message passing is visible to the programmer. As calling and called procedures exist on different machines, they execute in different address spaces, the parameters and result should be identical and if machines crash during communication, it causes problems.



Client Stub: Used when read is a remote procedure. Client stub is put into a library and is called using a calling sequence. It calls for the local operating system. It does not ask for the local operating system to give data, it asks the server and then blocks itself till the reply comes.

Server Stub: when a message arrives, it directly goes to the server stub. Server stub has the same functions as the client stub. The stub here unpacks the parameters from the message and then calls the server procedure in the usual way.

Summary of the process:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's as sends the message to the remote as.

4. The remote as gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local as.
8. The server's as sends the message to the client's as.
9. The client's as gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Implementation of RPC

The implementation of an RPC mechanism is based on the concept of stubs, which provide a perfectly normal (local) procedure call abstraction by concealing from programs the interface to the underlying RPC system. We saw that an RPC involves a client process and a server process. Therefore, to conceal the interface of the underlying RPC system from both the client and server processes, a separate stub procedure is associated with each of the two processes. Moreover, to hide the existence and functional details of the underlying network, an RPC communication package (known as RPCRuntime) is used on both the client and server sides. Thus, implementation of an RPC mechanism usually involves the following five elements of program

1. The client
2. The client stub
3. The RPCRuntime
4. The server stub
5. The server

The interaction between them is shown in Figure 4.2. The client, the client stub, and one instance of RPCRuntime execute on the client machine, while the server, the server stub, and another instance of RPCRuntime execute on the server machine. The job of each of these elements is described below.

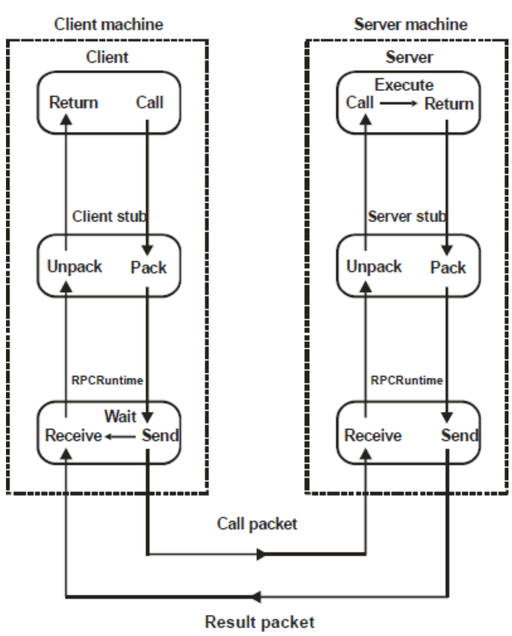


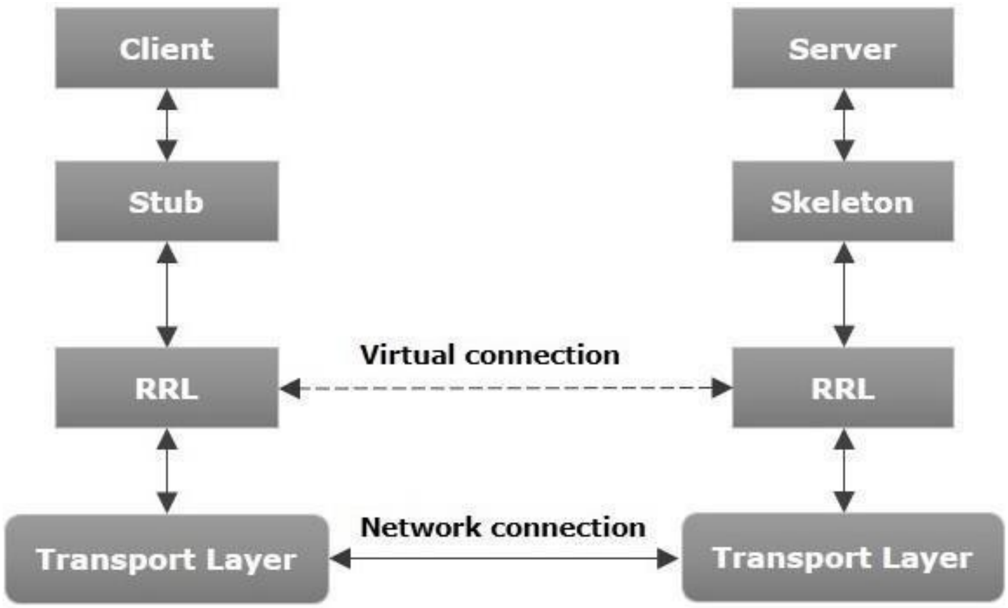
Fig. 3.2 : Implementation of RPC mechanism

Remote Method Invocation (RMI)

RMI stands for Remote Method Invocation. It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

The following diagram shows the architecture of an RMI application.



Working of an RMI Application

The following points summarize how an RMI application works –

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called invoke() of the object remoteRef. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as marshalling.

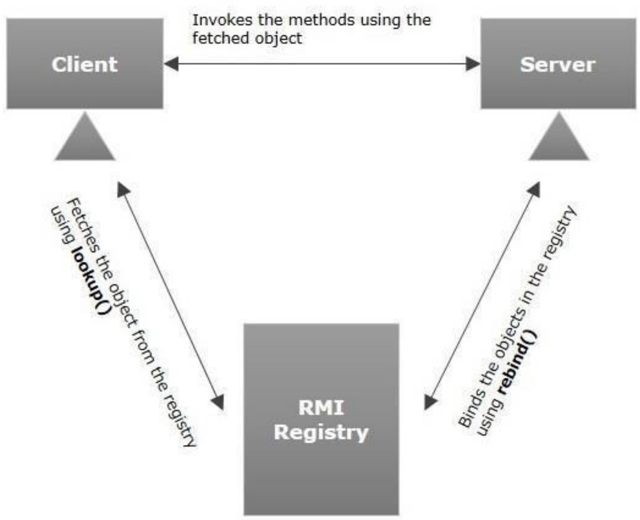
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling.

RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using lookup() method).

The following illustration explains the entire process –



To write an RMI Java application, you would have to follow the steps given below –

- Define the remote interface
- Develop the implementation class (remote object)
- Develop the server program
- Develop the client program
- Compile the application
- Execute the application

Code:

RMI_Interface:

```
package pkg_RMI;



import java.rmi.Remote;

import java.rmi.RemoteException;



public interface RMI_Interface extends Remote {

   void displayMessage() throws RemoteException;

   int factorial(int n) throws RemoteException;

}
```

RMI_Client:

```
package pkg_RMI;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
import java.util.Scanner;
import java.rmi.NotBoundException;
import java.rmi.Naming;
public class RMI_Client {
 public static void main(String[] args) throws
MalformedURLException, RemoteException,
NotBoundException{
		try {
			RMI_Interface helloAPI =
(RMI_Interface)
Naming.lookup("rmi://localhost:1880/hello");
//			helloAPI.displayMessage();
			Scanner sc = new Scanner(System.in);
			System.out.println("Enter a number to
find the factorial: ");
			int n = sc.nextInt();
			int ans = helloAPI.factorial(5);
			System.out.println("Factorial of "+n+" is
"+ans);
		}
		catch(Exception e)
		{
			System.out.println("The RMI APP is not
running...");
			e.printStackTrace();
		}
 }


}
```

RMI_Server:

```
package pkg_RMI;
import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
public class RMI_Server extends UnicastRemoteObject
implements RMI_Interface{
 public RMI_Server() throws RemoteException {
		super();
 }
 public static void main(String[] args)throws
RemoteException, AlreadyBoundException {
		try {
			Registry registry =
LocateRegistry.createRegistry(1880);
			registry.bind("hello", new
RMI_Server());
			System.out.println("The RMI_Server is
running and ready...");
		}
		catch (Exception e) {
		   e.printStackTrace();
		   System.out.println("The RMI_Server is not
running...");
		}
 }
 @Override
 public void displayMessage()throws RemoteException{
		System.out.println("------------------------------------
---------");
		System.out.println("Hello Akash!");
		System.out.println("------------------------------------
---------");

 }

 @Override
 public int factorial(int n)throws RemoteException{
		int fact = 1;
		for(int i=1;i<=n;i++) {
			fact *= i;
		}
		return fact;
 }
}
```

Output:

```
<terminated> RMI_Client [Java Application] C:\Program File
Enter a number to find the factorial:
5
Factorial of 5 is 120
```

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 3 – Inter-process Communication

Learning Objective: Demonstrate a program for Inter-process communication

Tools : Java

Theory:

Inter-process Communication

A process is a program in execution. When we say that two computers of a distributed system are communicating with each other, we mean that two processes, one running on each computer, are in communication with each other. In a distributed system, processes executing on different computers often need to communicate with each other to achieve some common goal. For example, each computer of a distributed system may have a resource manager process to monitor the current status of usage of its local resources, and the resource managers of all the computers might communicate with each other from time to time to dynamically balance the system load among all the computers. Therefore, a distributed operating system needs to provide inter-process communication (lPC) mechanisms to facilitate such communication activities.

Inter-process communication basically requires information sharing among two or more processes. The two basic methods for information sharing are as follows:

1. Original sharing, or shared-data approach

2. Copy sharing, or message-passing approach

In the shared-data approach, the information to be shared is placed in a common memory area that is accessible to all the processes involved in an IPC. The shared-data paradigm gives the conceptual communication pattern illustrated in Figure 3.1(a). On the other hand, in the message-passing approach, the information to be shared is physically copied from the sender process's address space to the address spaces of all the receiver processes, and this is done by transmitting the data to be copied in the form of messages (a message is a block of information). The message-passing paradigm gives the conceptual communication pattern illustrated in Figure 3.1(b). That is, the communicating processes interact directly with each other.

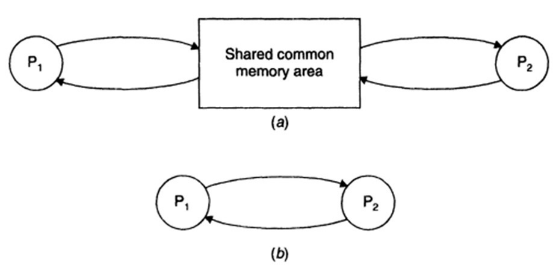


Fig. 3.1 The two basic interprocess communication paradigms: (*a*) The shared-data approach. (*b*) The message-passing approach.

Since computers in a network do not share memory, processes in a distributed system normally communicate by exchanging messages rather than through shared data. Therefore, message passing is the basic IPe mechanism in distributed systems.

A message-passing system is a subsystem of a distributed operating system that provides a set of message-based IPe protocols and does so by shielding the details of complex network protocols and multiple heterogeneous platforms from programmers. It enables processes to communicate by exchanging messages and allows programs to be written by using simple communication primitives, such as send and receive. It serves as a suitable infrastructure for building other higher level lPC systems, such as remote procedure call and distributed shared memory

Model of interprocess communication

The models of interprocess communication are as follows:

Shared Memory Model

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. All POSIX systems, as well as Windows operating systems use shared memory.

1. **Advantage of Shared Memory Model**

   Memory communication is faster on the shared memory model as compared to the message passing model on the same machine.

1. **Disadvantages of Shared Memory Model**

   Some of the disadvantages of shared memory model are as follows:

   - All the processes that use the shared memory model need to make sure that they are not writing to the same memory location.
   - Shared memory model may create problems such as synchronization and memory protection that need to be addressed.

Message Passing Model

Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored on queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

1. **Advantage of Messaging Passing Model**

   The message passing model is much easier to implement than the shared memory model.

2. **Disadvantage of Messaging Passing Model**

   The message passing model has slower communication than the shared memory model because the connection setup takes time.

Characteristics Of Inter-process Communication

There are mainly five characteristics of inter-process communication in a distributed environment/system.

1. **Synchronous System Calls:**

   In the synchronous system calls both sender and receiver use blocking system calls to transmit the data which means the sender will wait until the acknowledgment is received from the receiver and receiver waits until the message arrives.

2. **Asynchronous System Calls:**
   In the asynchronous system calls, both sender and receiver use non-blocking system calls to transmit the data which means the sender doesn't wait from the receiver acknowledgment.
3. **Message Destination:**
   A local port is a message destination within a computer, specified as an integer. Aport has exactly one receiver but many senders. Processes may use multiple ports from which to receive messages. Any process that knows the number of a port can send the message to it.
4. **Reliability:**
   It is defined as validity and integrity.
5. **Integrity:**
   Messages must arrive without corruption and duplication to the destination.
6. **Validity:**
   Point to point message services are defined as reliable, If the messages are guaranteed to be delivered without being lost is called validity.
7. **Ordering:**
   It is the process of delivering messages to the receiver in a particular order. Some applications require messages to be delivered in the sender order i.e the order in which they were transmitted by the sender

Result and Discussion:

WriterProgram:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
public class WriterProgram {
    public static void main(String[] args) {
        String filePath = "message.txt";
        String message = "Hello from Dhiraj!";
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            writer.write(message);
```

```
            System.out.println("Message written to " + filePath);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ReaderProgram:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class ReaderProgram {
    public static void main(String[] args) {
        String filePath = "message.txt";
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println("Message read from " + filePath + ": " + line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

OUTPUT:

```
PS D:\Legasis\Login> javac WriterProgram.java
PS D:\Legasis\Login> javac ReaderProgram.java
PS D:\Legasis\Login> java WriterProgram
Message written to message.txt
PS D:\Legasis\Login>
```

```
PS D:\Legasis\Login> javac WriterProgram.java
PS D:\Legasis\Login> javac ReaderProgram.java
PS D:\Legasis\Login> java WriterProgram
Message written to message.txt
PS D:\Legasis\Login> java ReaderProgram
Message read from message.txt: Hello from Dhiraj!
PS D:\Legasis\Login>
```

Learning Outcomes: The student should have the ability to

LO1: Describe the protocol for Inter process communication.

LO 2: justify that client server are managed properly by the Inter process communication

Course Outcomes: Upon completion of the course students will be able to understand interprocess communication.

Conclusion:……………………………………………………………………………………

……………………………………………………………………………………………………

…………………………………………………………………………………………………....

Viva Questions:

1. What is Inter-process communication?
2. What are the models of IPC?
3. What do you mean by "unicast" and "multicast" IPC?
4. Write operations provided in IPC?
5. Which transport protocol is used by remote procedure call (RPC)?

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 04: Write a program to implement group communication in distributed computing

Learning Objective: Student should be able to implement group communication in distributed computing
Tools: Python
Theory: Group communication refers to the exchange of information and ideas among members of a group. It can take place through various channels and involve multiple participants. In the context of programming, group communication often involves coordinating and exchanging data among different components or processes.

Types of Group Communication:

1. Broadcast Communication:
Definition: In broadcast communication, messages are sent to all members of the group.
Programming Context: In programming, broadcast communication often involves sending information or signals to all components or processes within a system. This is particularly useful for scenarios where global updates or notifications need to be disseminated to all participants.
Example: In a distributed system, a server might broadcast a message to all connected clients to notify them of a system-wide event, such as server maintenance or an important update.

2. Multicast Communication:
Definition: In multicast communication, messages are sent to a specific subset of the group.
Programming Context: Multicast is beneficial when you want to target a specific group of participants rather than the entire set. This is useful for scenarios where certain components or processes share common interests or responsibilities.
Example: In a multiplayer online game, multicast communication could be used to send updates only to players in a specific region or those involved in a particular in-game event.

3. Unicast Communication:
Definition: In unicast communication, messages are sent between two specific members of the group.
Programming Context: Unicast is similar to traditional one-to-one communication. It is commonly used for direct communication between two components or processes within a group.
Example: In a peer-to-peer network, unicast communication might occur between two nodes exchanging specific data, such as file transfers or real-time chat messages.

Additional Considerations:

Message Queues: Many group communication implementations involve the use of message queues. Processes or components can publish messages to a queue, and subscribers receive messages from the queue based on their interest or topic.

Reliability: Depending on the application, you might need to consider the reliability of communication. For instance, using acknowledgment mechanisms or ensuring message delivery order may be crucial in certain scenarios.

Scalability: Group communication mechanisms should be scalable to accommodate a growing number of participants. This involves considerations of system architecture and the chosen communication patterns.

Security: When designing group communication, security measures such as encryption and authentication should be considered, especially if sensitive data is being exchanged.

Code:

```
import multiprocessing

def worker_function(worker_id, shared_data):
    print(f"Worker {worker_id} received: {shared_data.value}")

if __name__ == "__main__":
    # Shared data among processes
    shared_data = multiprocessing.Value('i', 10)

    # Creating multiple processes
    num_workers = 3
    processes = []

    for i in range(num_workers):
        process = multiprocessing.Process(target=worker_function, args=(i, shared_data))
        processes.append(process)
        process.start()

    # Broadcasting data to all processes
    # shared_data.value = 42

    # Waiting for all processes to finish
    for process in processes:
        process.join()
```

Output:

```
Worker 0 received: 10
Worker 1 received: 10
Worker 2 received: 10
```

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 5 –Election Algorithm

Learning Objective: Student should be able to develop a program for Election Algorithm

Tools : Java/Python

Theory:

Several distributed algorithms require that there be a coordinator process in the entire system that performs some type of coordination activity needed for the smooth running of other processes in the system. Two examples of such coordinator processes encountered

1. The coordinator in the centralized algorithm for mutual exclusion.
2. The central coordinator in the centralized deadlock detection algorithm.

Since all other processes in the system have to interact with the coordinator, they all must unanimously agree on who the coordinator is. Furthermore, if the coordinator process fails due to the failure of the site on which it is located, a new coordinator process must be elected to take up the job of the failed coordinator. Election algorithms are meant for electing a coordinator process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system.

Election algorithms are based on the following assumptions:

1. Each process in the system has a unique priority number.
2. Whenever an election is held, the process having the highest priority number among thecurrently active processes is elected as the coordinator.
3. On recovery, a failed process can take appropriate actions to rejoin the set of active processes.

Therefore, whenever initiated, an election algorithm basically finds out which of the currently active processes has the highest priority number and then informs this to all other active processes.

The Bully Algorithm

As a first example, consider the bully algorithm . When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:

1. P sends an ELECTION message to all processes with higher numbers.
2. If no one responds, P wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. P's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Fig. 6-20 we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely 5, 6, and 7. as shown in Fig. 6-20(a). Processes 5 and 6 both respond with OK, as shown in Fig. 6-20(b). Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).

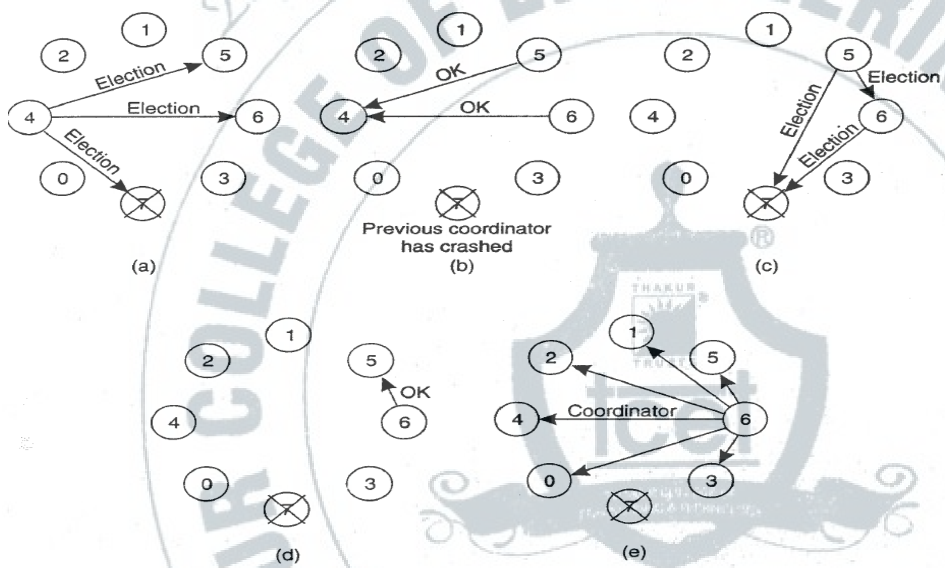


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

In Fig. 6-20(c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Fig. 6-20(d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to takeover, 6 announces this by sending a COORDINATOR message to all running processes.When 4 gets this message, it can now continue with the operation it was trying to do when it was discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue.If process 7 is ever restarted, it will just send an others a COORDINATOR message and bully them into submission.

A Ring Algorithm

Another election algorithm is based on the use of a ring. Unlike some ring algorithms, this one does not use a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor.

If the successor is down, the sender skips over the successor and goes to the next member along the ring. or the one after that, until a running process is located. At each step along the way, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as coordinator.

Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.

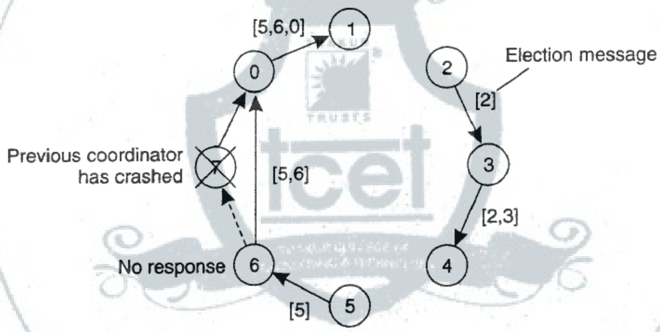


Figure 6-21. Election algorithm using a ring.

Result and Discussion:

BULLY ALGORITHM:

```
import java.io.InputStream;
import java.io.PrintStream;
import java.util.Scanner;

public class Bully {
   static boolean[] state = new boolean[5];
   int coordinator;

   public static void up(int up) {
      if (state[up - 1]) {
         System.out.println("process" + up +
"is already up");
      } else {
         int i;
         Bully.state[up - 1] = true;
         System.out.println("process " + up +
"held election");
            for (i = up; i < 5; ++i) {
               System.out.println("election
message sent from process" + up + "to
process" + (i + 1));
            }
            for (i = up + 1; i <= 5; ++i) {
               if (!state[i - 1]) continue;
               System.out.println("alive message
send from process" + i + "to process" + up);
               break; }}}
   public static void down(int down) {
      if (!state[down - 1]) {
         System.out.println("process " +
down + "is already dowm.");
      } else {
```

```
  Bully.state[down - 1] = false;
      }
   }

   public static void mess(int mess) {
      if (state[mess - 1]) {
         if (state[4]) {
            System.out.println("0K");
         } else if (!state[4]) {
            int i;
            System.out.println("process" +
mess + "election");
            for (i = mess; i < 5; ++i) {
               System.out.println("election
send from process" + mess + "to process " +
(i + 1));
            }
            for (i = 5; i >= mess; --i) {
               if (!state[i - 1]) continue;
               System.out.println("Coordinator
message send from process" + i + "to all");
               break;
            }
         }
      } else {
         System.out.println("Prccess" + mess
+ "is down");
      }
   }

   public static void main(String[] args) {
      int choice;
      Scanner sc = new Scanner(System.in);
      for (int i = 0; i < 5; ++i) {
         Bully.state[i] = true;
      }
      System.out.println("5 active process
are:");
      System.out.println("Process up  = p1 p2
p3 p4 p5");
System.out.println("Process 5 is
coordinator");
        do {
            System.out.println(".........");
            System.out.println("1 up a
process.");
            System.out.println("2.down a
process");
            System.out.println("3 send a
message");
            System.out.println("4.Exit");
            choice = sc.nextInt();
            switch (choice) {
               case 1: {
                  System.out.println("bring
proces up");
                  int up = sc.nextInt();
                  if (up == 5) {
                     System.out.println("process 5
is co-ordinator");
                     Bully.state[4] = true;
                     break;
                  }
                  Bully.up(up);
                  break;
               }
               case 2: {
                  System.out.println("bring down
any process.");
                  int down = sc.nextInt();
                  Bully.down(down);
                  break;
               }
               case 3: {
                  System.out.println("which
process will send message");
                  int mess = sc.nextInt();
                  Bully.mess(mess);
               }
            }
        } while (choice != 4);
```

OUTPUT:

```
java -cp /tmp/OXWi1ezEPF Bully
5 active process are:
Process up  = p1 p2 p3 p4 p5
Process 5 is coordinator
.........
1 up a process.
2.down a process
3 send a message
4.Exit

2
bring down any process.
5
.........
1 up a process.
2.down a process
3 send a message
4.Exit
3
which process will send message
2
process2election
election send from process2to process 3
election send from process2to process 4
election send from process2to process 5
Coordinator message send from process4to all
```

…….

RING ALGORITHM:

```
import java.util.Scanner;

public class Ring {

public static void main(String[] args) {

// TODO Auto-generated method stub

        int temp, i, j;
        char str[] = new char[10];
        Rr proc[] = new Rr[10];

// object initialisation
for (i = 0; i < proc.length; i++)
    proc[i] = new Rr();

// scanner used for getting input from
console
Scanner in = new Scanner(System.in);
System.out.println("Enter the number of
process : ");
        int num = in.nextInt();

// getting input from users
for (i = 0; i < num; i++) {
    proc[i].index = i;
System.out.println("Enter the id of process :
");
proc[i].id = in.nextInt();
proc[i].state = "active";
proc[i].f = 0;
        }
// sorting the processes from on the basis of
id
for (i = 0; i < num - 1; i++) {
for (j = 0; j < num - 1; j++) {
    if (proc[j].id > proc[j + 1].id) {
        temp = proc[j].id;
    proc[j].id = proc[j + 1].id;
    proc[j + 1].id = temp;
                }
            }
        }

for (i = 0; i < num; i++) {
System.out.print("  [" + i + "]" + " " +
```

```
proc[i].id);
                    }
int init;
int ch;
int temp1;
int temp2;
int ch1;
int arr[] = new int[10];

proc[num - 1].state = "inactive";

System.out.println("\n process " + proc[num
- 1].id + "select as co-ordinator");

while (true) {
System.out.println("\n 1.election 2.quit ");
                              ch = in.nextInt();

          for (i = 0; i < num; i++) {
             proc[i].f = 0;
                              }

switch (ch) {
          case 1:
System.out.println("\n Enter the Process
number who initialsied election : ");
          init = in.nextInt();
          temp2 = init;
          temp1 = init + 1;
          i = 0;

while (temp2 != temp1) {
if ("active".equals(proc[temp1].state) &&
proc[temp1].f == 0) {
System.out.println("\nProcess " +
proc[init].id + " send message to " +
proc[temp1].id);
proc[temp1].f = 1;
init = temp1;
arr[i] = proc[temp1].id;
i++;
}
if (temp1 == num) {
temp1 = 0;
} else {
temp1++;
}}
System.out.println("\nProcess " +
proc[init].id + " send message to " +
proc[temp1].id);
arr[i] = proc[temp1].id;
          i++;
int max = -1;

// finding maximum for co-ordinator
selection
for (j = 0; j < i; j++) {
if (max < arr[j]) {
max = arr[j];  }}

// co-ordinator is found then printing on
console
System.out.println("\n process " + max +
"select as co-ordinator");
for (i = 0; i < num; i++) {

if (proc[i].id == max) {

          proc[i].state = "inactive"; } }
                    break;
case 2:
          System.out.println("Program
terminated ...");
          return ;
default:
System.out.println("\n invalid response \n");
          break;
                              }}}}

class Rr {

          public int index;   // to store the
index of process
          public int id;      // to store id/name of
process
          public int f;
          String state;       // indiactes whether
active or inactive state of node }
```

```
Enter the number of process :
4
Enter the id of process :
10
Enter the id of process :
11
Enter the id of process :
12
Enter the id of process :
13
  [0] 10  [1] 11  [2] 12  [3] 13
 process 13select as co-ordinator

 1.election 2.quit
1

 Enter the Process number who initialsied election :
1

Process 11 send message to 12

Process 12 send message to 10

Process 10 send message to 11

 process 12select as co-ordinator
```

Learning Outcomes: The student should have the ability to
LO1: Describe the Election Algorithms.

LO2: Write a Program to Demonstrate Election Algorithms.

Course Outcomes: Upon completion of the course students will be able to understand Election Algorithms

for Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completio nof Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment Number 06

Aim: Develop a program for Clock Synchronization algorithms

Learning Objective: Student should be able to develop Clock Synchronization algorithms

Tools: Java

Theory:

Clock synchronization in distributed computing involves ensuring that the clocks across multiple machines in a network are aligned to a common time reference. This is crucial for coordinating actions, ordering events, and maintaining consistency in distributed systems. One widely used algorithm for clock synchronization is the Network Time Protocol (NTP). NTP employs a hierarchical system of time servers to disseminate accurate time information across the network. Each server synchronizes its clock with a higher-stratum server, ultimately obtaining time from a primary time source such as an atomic clock. Clients periodically query the time from these servers and adjust their clocks accordingly, minimizing clock skew and maintaining synchronized time across the network.

The Berkeley Algorithm

It is a clock synchronization algorithm used in distributed systems. It works by periodically synchronizing the local clocks of all machines in the network with a reference clock known as the master clock. Here's how it works:

1. **Master Selection:** One machine in the network is designated as the master, typically the one with the most accurate clock or an external time source like a GPS receiver.
2. **Measurement:** Periodically, each machine measures the difference between its local clock and the master clock. This difference is known as the clock offset.
3. **Calculation:** After collecting clock offsets from all machines, the average offset is calculated. This average represents the discrepancy between the average time of the machines and the master clock.
4. **Adjustment:** Each machine adjusts its local clock by applying the calculated average offset, thus synchronizing its time with the master clock.

This process helps minimize clock skew and ensures that all machines in the network maintain relatively synchronized time.

Clock Synchronization algorithm Berkeley's Algorithm

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
class Machine {
    private int id;
    private int clock;
    public Machine(int id) {
        this.id = id;
        this.clock = new
Random().nextInt(100); // Initialize
clock with random value}
    public int getId() {
        return id; }
    public int getClock() {
        return clock; }
```

```
    public void setClock(int clock) {
        this.clock = clock;  }}
public class BerkeleyAlgorithm {
    private List<Machine> machines;
    public BerkeleyAlgorithm(int
numMachines) {
        machines = new ArrayList<>();
        for (int i = 0; i < numMachines;
i++) {
            machines.add(new
Machine(i));}}
    public void synchronizeClocks() {
        int sum = 0;
        for (Machine machine : machines)
{            sum += machine.getClock();}
```

```
        int average = sum /
machines.size();
        for (Machine machine : machines)
{
            machine.setClock(average);} }
    public void printClocks() {
        for (Machine machine : machines)
{
            System.out.println("Machine "
+ machine.getId() + ": Clock = " +
machine.getClock());}}
    public static void main(String[]
args) {
        BerkeleyAlgorithm algorithm = new
BerkeleyAlgorithm(5); // Number of
machines
        System.out.println("Before
synchronization:");
        algorithm.printClocks();
        algorithm.synchronizeClocks();
        System.out.println("\nAfter
synchronization:");
        algorithm.printClocks(); }}
```

Output:

```
Before synchronization:
Machine 0: Clock = 33
Machine 1: Clock = 33
Machine 2: Clock = 48
Machine 3: Clock = 93
Machine 4: Clock = 64
```

```
After synchronization:
Machine 0: Clock = 54
Machine 1: Clock = 54
Machine 2: Clock = 54
Machine 3: Clock = 54
Machine 4: Clock = 54
```

Result and Discussion:

The Berkeley Algorithm in Java enables clock synchronization among distributed machines by averaging and adjusting clock values, ensuring coordination and consistency crucial for distributed environments.

Learning Outcomes: The student should have the ability to

LO1: Describe the Clock Synchronization algorithms

LO2: Write a Program to the Clock Synchronization algorithms

Course Outcomes: Upon completion of the course students will be able to understand Clock synchronization algorithms.

Conclusion:

To conclude, the Berkeley Algorithm, implemented in Java, effectively synchronizes clocks among distributed machines by averaging and adjusting their values. This ensures coordination and consistency crucial for distributed environments, demonstrating the algorithm's practical utility in real-world applications.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 7– Token Based Algorithm

Learning Objective: Student should be able to design a program to illustrate token based algorithms.

Tools : Java

Theory:

Mutual Exclusion

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, a file must not be simultaneously updated by multiple processes. Similarly, use of unit record peripherals such as tape drives or printers must be restricted to a single process at a time. Therefore, exclusive access to such a. shared resource by a process must be ensured. This exclusiveness of access is called mutual exclusion between processes. The sections of a program that need exclusive access to shared resources are referred to as critical sections. For mutual exclusion, means are introduced to prevent processes from executing concurrently within their associated critical sections.
An algorithm for implementing mutual exclusion must satisfy the following requirements:

1. Mutual exclusion. Given a shared resource accessed by multiple concurrent processes, at any time only one process should access the resource. That is, a process that has been granted the resource must release it before it can be granted to another process.

2. No starvation. If every process that is granted the resource eventually releases it, every request must be eventually granted.

In single-processor systems, mutual exclusion is implemented using semaphores, monitors, and similar constructs.

Suzuki–Kasami Algorithm

Suzuki–Kasami algorithm is a token-based algorithm for achieving mutual exclusion in distributed systems.This is modification of Ricart–Agrawala algorithm, a permission based (Non-token based) algorithm which uses **REQUEST** and **REPLY** messages to ensure mutual exclusion.

In token-based algorithms, A site is allowed to enter its critical section if it possesses the unique token. Non-token based algorithms uses timestamp to order requests for the critical section where as sequence number is used in token based algorithms.

Data structure and Notations:

- An array of integers **RN[1…N]**

  A site Si keeps **RNi[1…N]**, where **RNi[j]** is the largest sequence number received so far through **REQUEST** message from site Si.
- An array of integer **LN[1…N]**

  This array is used by the token.**LN[J]** is the sequence number of the request that is recently executed by site Sj.
- A queue **Q**
  This data structure is used by the token to keep a record of ID of sites waiting for the token

Algorithm:

- **To enter Critical section:**
  - When a site Si wants to enter the critical section and it does not have the token then it increments its sequence number **RNi[i]** and sends a request message **REQUEST(i, sn)** to all other sites in order to request the token. Here **sn** is update value of **RNi[i]**
  - When a site Sj receives the request message **REQUEST(i, sn)** from site Si, it sets **RNj[i]** to maximum of **RNj[i]** and **sn** i.e **RNj[i]** = max(**RNj[i], sn**).
  - After updating **RNj[i]**, Site Sj sends the token to site Si if it has token and **RNj[i]** = **LN[i]** + 1
- **To execute the critical section:**
  - Site Si executes the critical section if it has acquired the token.
- **To release the critical section:**

  After finishing the execution Site Si exits the critical section and does following:
  - sets **LN[i]** = **RNi[i]** to indicate that its critical section request **RNi[i]** has been executed
  - For every site Sj, whose ID is not prsent in the token queue **Q**, it appends its ID to **Q** if **RNi[j]** = **LN[j]** + 1 to indicate that site Sj has an outstanding request.

- After above updation, if the Queue **Q** is non-empty, it pops a site ID from the **Q** and sends the token to site indicated by popped ID.
- If the queue **Q** is empty, it keeps the token

Message Complexity:

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

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

Drawbacks of Suzuki–Kasami Algorithm:

- **Non-symmetric Algorithm:** A site retains the token even if it does not have requested for critical section. According to definition of symmetric algorithm "No site possesses the right to access its critical section when it has not been requested."

Result and Discussion:

Code:

```
import java.util.concurrent.Semaphore;

public class SuzukiKasami {
    private static final int NUM_NODES = 5; // Number of nodes in the distributed system
    private static final int NUM_REQUESTS = 10; // Number of requests for each node

    private static int[] request = new int[NUM_NODES];
    private static int[] token = new int[NUM_NODES];
    private static int[][] quorum = {
        {1, 2, 4},  // Quorum for node 0
        {0, 2, 3},  // Quorum for node 1
        {0, 1, 3},  // Quorum for node 2
        {1, 2, 4},  // Quorum for node 3
        {0, 3, 4}   // Quorum for node 4
    };

    private static Semaphore mutex = new Semaphore(1);

    public static void main(String[] args) {
        for (int i = 0; i < NUM_NODES; i++) {
            final int nodeId = i;
            new Thread(() -> {
```

```
            for (int j = 0; j < NUM_REQUESTS; j++)
{
                requestCriticalSection(nodeId);
                // Simulate critical section execution
                System.out.println("Node " + nodeId + "
is in critical section");
                try {
                    Thread.sleep((long) (Math.random()
* 1000));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                releaseCriticalSection(nodeId);
            }
        }).start();
    }
}

    private static void requestCriticalSection(int
nodeId) {
        try {
            mutex.acquire();
            request[nodeId] = 1;
            for (int i : quorum[nodeId]) {
                if (i != nodeId) {
                    while (request[i] == 1) {
                        mutex.release();
                        Thread.sleep(10); // Adjust as needed
                        mutex.acquire();
                    }
                }
            }
            token[nodeId] = 1;
            mutex.release();
            for (int i : quorum[nodeId]) {
                if (i != nodeId) {
                    while (token[i] == 0) {
                        Thread.sleep(10); // Adjust as needed
                    }
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private static void releaseCriticalSection(int
nodeId) {
        try {
            mutex.acquire();
            token[nodeId] = 0;
            request[nodeId] = 0;
            mutex.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

The output will consist of lines indicating when each node enters and exits the critical section.

Node 0 is in critical section

Node 1 is in critical section

Node 2 is in critical section

Node 0 is in critical section

Node 3 is in critical section

Node 1 is in critical section

Node 4 is in critical section

Node 2 is in critical section

Node 0 is in critical section

...

The Suzuki-Kasami Algorithm ensures mutual exclusion in a distributed system by coordinating access to a critical section among multiple nodes. It achieves this by following these key steps Request phase, Token phase, Critical Section Execution, Release phase.

Learning Outcomes: The student should have the ability to

LO1: Recall the different token based algorithm.

LO2: Apply the different token based algorithm.

Course Outcomes: Upon completion of the course students will be able to understand token based Algorithm.

Conclusion:

The Suzuki-Kasami Algorithm ensures mutual exclusion in a distributed system by coordinating access to a critical section among multiple nodes. It achieves this by following these key steps Request phase, Token phase, Critical Section Execution, Release phase as implemented in the code.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 8 – Non - Token Based Algorithm

Learning Objective: Student should be able to design a program to illustrate non token based algorithm.

Tools: Java

Theory:

Non-token based approach:

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

Example:

- Lamport's algorithm, Ricart–Agrawala algorithm

Ricart–Agrawala algorithm

Ricart–Agrawala algorithm is an algorithm to for mutual exclusion in a distributed system proposed by Glenn Ricart and Ashok Agrawala. This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm. Like Lamport's Algorithm, it also follows permission based approach to ensure mutual exclusion.

In this algorithm:

- Two type of messages (**REQUEST** and **REPLY**) are used and communication channels are assumed to follow FIFO order.
- A site send a **REQUEST** message to all other site to get their permission to enter critical section.
- A site send a **REPLY** message to other site to give its permission to enter the critical section.
- A timestamp is given to each critical section request using Lamport's logical clock.
- Timestamp is used to determine priority of critical section requests. Smaller timestamp gets high priority over larger timestamp. The execution of critical section request is always in the order of their timestamp.
- **To enter Critical section:**
    - When a site Si wants to enter the critical section, it send a timestamped **REQUEST**
      message to all other sites.
    - When a site Sj receives a **REQUEST** message from site Si, It sends a

REPLY
message to site Si if and only if

- Site Sj is neither requesting nor currently executing the critical section.
- In case Site Sj is requesting, the timestamp of Site Si's request is smaller than its own request.

- Otherwise the request is deferred by site Sj.

- **To execute the critical section:**
  - Site Si enters the critical section if it has received the **REPLY** message from all other sites.
- **To release the critical section:**
  - Upon exiting site Si sends **REPLY** message to all the deferred requests.

Message Complexity:

Ricart–Agrawala algorithm requires invocation of 2(N – 1) messages per critical section execution. These 2(N – 1) messages involves

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

Drawbacks of Ricart–Agrawala algorithm:

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

<u>Code:</u>

```
package ric_agr;
import java.util.*;
public class Ric_Agr {
   private static final int numProcesses = 5;
   private static int processID;
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter process ID (0 to " + (numProcesses - 1) + "): ");
      processID = scanner.nextInt();
      scanner.close();
      for (int i = 0; i < numProcesses; i++) {
         if (i != processID) {
            requestCS(i);      }   }
         enterCS();}
   private static void requestCS(int dest) {
      System.out.println("Process " + processID + " sending request to Process " + dest); }
   private static void enterCS() {
      System.out.println("Process " + processID + " entering critical section");
      try {
         Thread.sleep(2000); // Simulate work in critical section
      } catch (InterruptedException e) {
         e.printStackTrace();  }
      System.out.println("Process " + processID + " exiting critical section");  }}
```

<u>Output:</u>

```
<terminated> Ric_Agr [Java Application] C:\Users\anany\D
Enter process ID (0 to 4): 0
Process 0 sending request to Process 1
Process 0 sending request to Process 2
Process 0 sending request to Process 3
Process 0 sending request to Process 4
Process 0 entering critical section
Process 0 exiting critical section
```

Result and Discussion:

This simplified Ricart-Agrawala algorithm simulation demonstrates a process (identified by process ID) sending requests to all other processes in the system. After sending requests, the process enters the critical section, performs some simulated work, and then exits the critical section. The code omits handling of incoming requests and coordination with other processes for brevity.

Learning Outcomes: The student should have the ability to

LO1: Recall the non token based algorithm.

LO2: Analyze the different non token based algorithm

Course Outcomes: Upon completion of the course students will be able to understand non token based Algorithm.

Conclusion: Hence, we were able to design a program to illustrate non token based algorithm.

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				

Experiment 9– Load Balancing Algorithm

Learning Objective: Student should be able to develop a program for Load Balancing Algorithm.

Tools : Java

Theory:

Load Balancing Algorithm:

The scheduling algorithms using this approach are known as load-balancing algorithms or load-leveling algorithms. These algorithms arc based on the intuition that, for better resource utilization, it is desirable for the load in a distributed system to be balanced evenly. Thus, a load-balancing algorithm tries to balance the total system load by transparently transferring the workload from heavily loaded nodes to lightly loaded nodes in an attempt to ensure good overall performance relative to some specific metric of system performance. When considering performance from the user point of view, the metric involved is often the response time of the processes. However, when performance is considered from the resource point of view, the metric involved is the total system throughput. In contrast to response time, throughput is concerned with seeing that all users are treated fairly and that all are making progress. Notice that the resource view of maximizing resource utilization is compatible with the desire to maximize system throughput. Thus the basic goal of almost all the load-balancing algorithms is to maximize the total system throughput.

Static versus Dynamic

At the highest level, we may distinguish between static and dynamic load-balancing algorithms. Static algorithms use only information about the average behavior of the system, ignoring the current state of the system. On the other hand, dynamic algorithms react to the system state that changes dynamically.

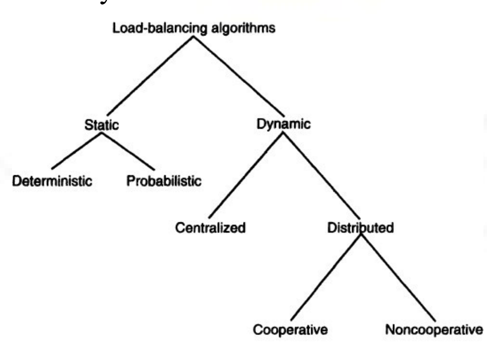


Fig. 7.3 A taxonomy of load-balancing algorithms.

Deterministic versus Probabilistic
Static load-balancing algorithms may be either deterministic or probabilistic. Deterministic algorithms use the information about the properties of the nodes and the characteristics of the processes to be scheduled to deterministically allocate processes to nodes. Notice that the task assignment algorithms basically belong to the category of deterministic static load-balancing algorithms.
Centralized versus Distributed
Dynamic scheduling algorithms may be centralized or distributed. In a centralized dynamic scheduling algorithm, the responsibility of scheduling physically resides on a single node. On the other hand, in a distributed dynamic scheduling algorithm, the work involved in making process assignment decisions is physically distributed among the various nodes of the system. In the centralized approach, the system state information is collected at a single node at which all scheduling decisions are made. This node is called the centralized server node. All requests for process scheduling are handled by the centralized server, which decides about the placement of a new process using the state information stored in it. The centralized approach can efficiently make process assignment decisions because the centralized server knows both the load at each node and the number of processes needing service. In the basic method, the other nodes periodically send status update messages to the central server node. These messages are used to keep the system state information up to date at the centralized server node. One might consider having the centralized server query the other nodes for state information. This would reduce message traffic if state information was used to answer several process assignment requests, but since nodes can change their load any time due to local activities, this would introduce problems of stale state information.
Cooperative versus Noncooperative
Distributed dynamic scheduling algorithms may be categorized as cooperative and noncooperative. In noncooperative algorithms, individual entities act as autonomous entities and make scheduling decisions independently of the actions of other entities. On the other hand, in cooperative algorithms, the distributed entities cooperate with each other to make scheduling decisions. Hence, cooperative algorithms are more complex and involve larger overhead than noncooperative ones. However, the stability of a cooperative algorithm is better than that of a noncooperative algorithm.

Result and Discussion:

Code:

```
import java.util.ArrayList;

public class RoundRobinLoadBalancer {
    // Simulating servers with ArrayLists
    private ArrayList<Integer>[] servers;
    private int numServers;

    public RoundRobinLoadBalancer(int numServers) {
        this.numServers = numServers;
        servers = new ArrayList[numServers];
```

```
        for (int i = 0; i < numServers; i++) {
            servers[i] = new ArrayList<>();
        }
    }

    // Add processes to the servers
    public void addProcesses(int[] processes) {
        int currentIndex = 0;
        for (int process : processes) {
            servers[currentIndex].add(process);
            currentIndex = (currentIndex + 1) % numServers; // Round robin distribution
        }
    }

    // Print processes present in each server
    public void printProcesses() {
        for (int i = 0; i < numServers; i++) {
            System.out.println("Server " + (i + 1) + " Processes: " + servers[i]);
        }
    }

    public static void main(String[] args) {
        // Initial processes in the servers
        int[] initialProcesses = {1, 2, 3, 4, 5, 6, 7};

        // Number of servers
        int numServers = 4;

        RoundRobinLoadBalancer loadBalancer = new
RoundRobinLoadBalancer(numServers);

        System.out.println("Processes before balancing:");
        for (int process : initialProcesses) {
            System.out.print(process + " ");
        }
        System.out.println();

        loadBalancer.addProcesses(initialProcesses);

        System.out.println("\nProcesses after balancing:");
        loadBalancer.printProcesses();
    }
}
```

Output:

```
PS C:\Users\tiwar\OneDrive\Desktop>  c:; cd 'c:\Users\tiwar\O
hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessage
b7984d6b23a\redhat.java\jdt_ws\Desktop_8197b4cc\bin' 'RoundRo
Processes before balancing:
1 2 3 4 5 6 7

Processes after balancing:
Server 1 Processes: [1, 5]
Server 2 Processes: [2, 6]
Server 3 Processes: [3, 7]
Server 4 Processes: [4]
PS C:\Users\tiwar\OneDrive\Desktop>
```

Learning Outcomes: The student should have the ability to

LO1: Comprehend the Load balancing concept

LO2: Analyze different that load balancing methods

Course Outcomes: Upon completion of the course students will be able to understand Load Balancing Algorithm.

Conclusion: We were able to understand Load Balancing Algorithm along with an Example program. We were able to understand issues in Load Balancing Algorithm.

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				

Experiment 10 – Microservices Architecture

Learning Objective: Student should be able to understand micro services architecture of applications.

Tools : Java

Theory:

"Microservices" defines microservices as an approach to building a single application consisting of multiple small services that operate independently, with each service running in its own process and communicating using lightweight mechanisms like HTTP resource APIs. These services are designed around specific business capabilities and can be deployed independently using fully automated deployment tools. There is minimal centralized management of these services, which can be written in different programming languages and use various data storage technologies.

Steps to designing a micro services architecture:

1. Understand the monolith
Study the operation of the monolith and determine the component functions and services it performs. Since all the functions will be mixed together, this may pose a challenge. It is an important part of determining what is needed for the microservices, so it should be the first thing developers focus on.
2. Develop the microservices
Develop each function of the application as an autonomous, independently running microservice. These usually run in a container on a cloud server.
3. Integrate the larger application
Loosely integrate the microservices via API gateways, so they work in concert to form the larger application. An iPaaS like DreamFactory can play an essential role in this step. Each microservice will work with the others to provide the necessary functions
4. Allocate system resources
Use container orchestration tools like Kubernetes to manage the allocation of system resources for each microservice. This step helps keep everything organized and ensures the entire system works as a whole.

Examples of Microservices in Action

Netflix :-

Amazon wasn't the only company to pioneer the world of microservices. Netflix is another company that has found success through the use of microservices connected with APIs.
Similar to Amazon, this microservices example began its journey in 2008 before the term "microservices" had come into fashion. Netflix started its movie-streaming service in 2007. By 2008 it was suffering from service outages and scaling challenges; for three days, it was unable to ship DVDs to members.

At this point, the company was still dealing with physical DVDs, which put a damper on how well it could serve its customers. Streaming was still a dream, and online business, while thriving, was difficult. The monolith design was still not very functional beyond a certain point. Microservices architecture was a much better option, but it didn't truly exist yet.

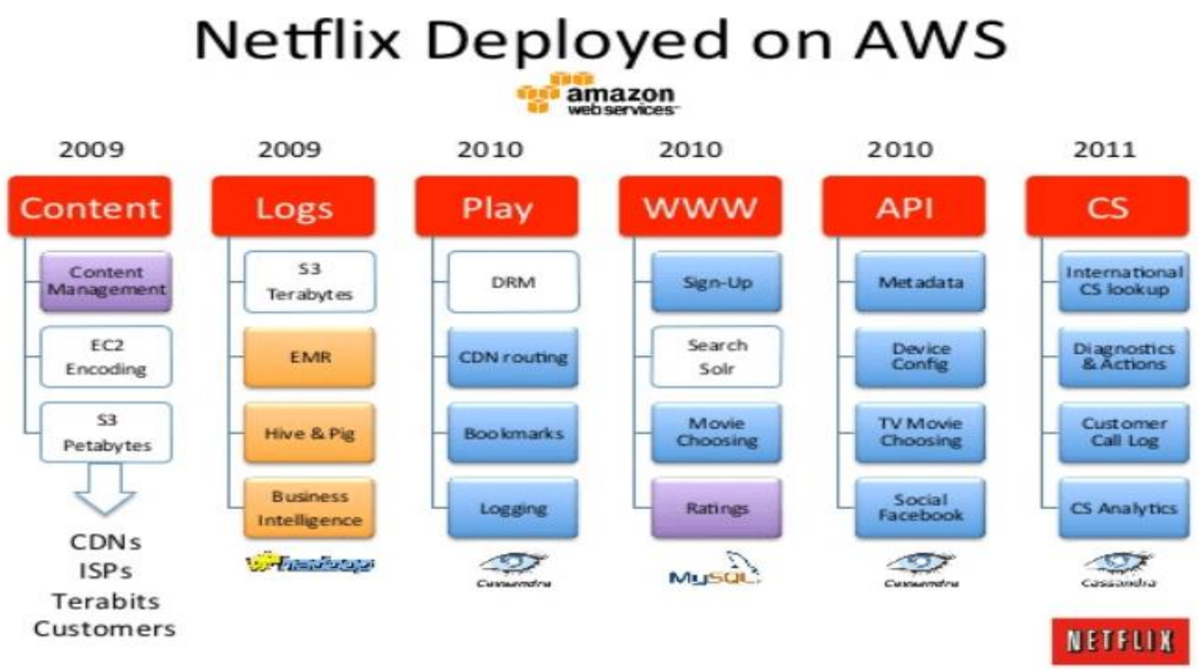
According to a Netflix Vice President:
"Our journey to the cloud at Netflix began in August of 2008, when we experienced a major database corruption and for three days could not ship DVDs to our members. That is when we realized that we had to move away from vertically

scaled single points of failure, like relational databases in our datacenter, towards highly reliable, horizontally scalable, distributed systems in the cloud. We chose Amazon Web Services (AWS) as our cloud provider because it provided us with the greatest scale and the broadest set of services and features." (Source)

In 2009, Netflix began the gradual process of refactoring its monolithic architecture, service by service, into microservices. The first step was to migrate its non-customer-facing, movie-coding platform to run on Amazon AWS cloud servers as an independent microservice. Netflix spent the following two years converting its customer-facing systems to microservices, finalizing the process in 2012.

The first step was to migrate its non-customer-facing, movie-coding platform to run on Amazon AWS cloud servers as an independent microservice. Netflix spent the following two years converting its customer-facing systems to microservices, finalizing the process in 2012.

Here's a diagram of Netflix's gradual transition to microservices:



Uber:-

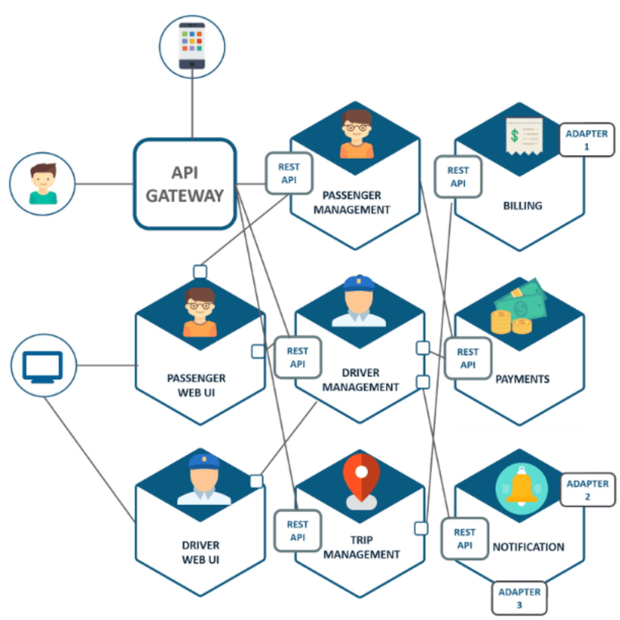
Despite being introduced to the world more recently than either of our previous examples, Uber also began with a monolith design.
This microservice example came not long after the launch of Uber, when the ride-sharing service encountered Growth Hurdles related to its monolithic application structure. The platform struggled to develop and launch new features efficiently, fix bugs, and integrate its rapidly growing global operations. Moreover, the complexity of Uber's monolithic application architecture requires developers to have extensive experience working with the existing system just to make minor updates and changes to the system.

Here's how Uber's monolithic structure worked at the time:

1. Passengers and drivers connected to Uber's monolith through a REST API.
2. There were three adapters – with embedded API for functions like billing, payment, and text messages.
3. There was a MySQL database.
4. All features were contained in the monolith.

This design was clunky and difficult to make changes to. For the developers, the ride share's popularity almost immediately caused problems. The company grew too fast to easily keep up with the app's original design.

Here's a diagram of Uber's **microservices architecture** from Dzone:



Uber transitioned from a monolithic architecture to cloud-based microservices, facilitated by an API gateway. This shift allowed for faster changes due to fewer intertwined functions within the monolith. The microservices architecture enabled specific teams to focus on individual services, enhancing expertise and facilitating faster issue resolution without affecting other services.
As Uber grew rapidly, scaling became easier with microservices, as teams could focus solely on the services needing scaling. The architecture ensured smooth operation without impacting other services, enhancing fault tolerance and reliability.
However, the lack of standardization across microservices posed a challenge, leading to potential chaos. Initially, local standards were implemented for each microservice but caused issues due to differences in availability and coordination during updates. Uber then opted for global standards, analyzing key principles like fault tolerance and scalability to

establish measurable standards. This transition was crucial for maintaining growth and stability amid internal struggles.

Benefits of Microservices:-

Using a Microservices approach offers several benefits, including improved scalability, improved scalability, improved application stability, and faster deployment times. Here, we will explore some of the most significant advantages of using microservices and discuss how they can help organizations build more robust and efficient applications.

1. Easy to Scale:
2. Platform and language agnostic:
3. Improved application stability:
4. Better security and compliance:
5. Easier to organize development teams:

Results and Discussion : - In this experiment, students delved into the intricacies of microservices architecture, grasping its principles through theoretical exploration and practical implementation. Through case studies of industry giants like Netflix and Uber, they gleaned insights into the transformative power of microservices in overcoming scalability and reliability challenges. While some students demonstrated adeptness in designing and integrating microservices, others faced hurdles, indicating the need for further guidance. Nonetheless, the experiment served as a foundational step, equipping students with essential skills for navigating modern software development landscapes and fostering a deeper appreciation for agile and scalable architectures.

Conclusion:-

For Faculty Use

Correction Parameters	Formative Assessment [40%]	Timely completion of Practical [40%]	Attendance / Learning Attitude [20%]	
Marks Obtained				