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:

A distributed computing system is a network of interconnected computers that work together to achieve a common goal. In such a system, tasks are divided among multiple computers, often referred to as nodes or hosts, and these computers communicate and coordinate with each other to accomplish the tasks efficiently. Distributed computing systems are designed to handle large-scale computations, data storage, and processing tasks that would be impractical or impossible for a single computer to handle.

Types of Architecture of Distributed Computing System:

Client-Server Architecture:

- In this architecture, there are two main types of nodes: clients and servers.
- Clients request services or resources from servers, which provide those services or resources.
- Servers typically have higher computational power and resources compared to clients.
- Examples include web servers serving web pages to client browsers, database servers serving data to client applications, etc.

Peer-to-Peer (P2P) Architecture:

- In a peer-to-peer architecture, all nodes have equal roles and responsibilities.
- Each node can act as both a client and a server, providing and consuming resources/services.
- Nodes communicate directly with each other without the need for a central server.
- Examples include file-sharing networks (e.g., BitTorrent), decentralized cryptocurrency networks (e.g., Bitcoin), etc.

Hybrid Architecture:

- Hybrid architectures combine elements of both client-server and peer-to-peer architectures.
- They may have a central server for certain tasks while also allowing peer-to-peer communication among nodes.
- This architecture offers flexibility and scalability by leveraging the benefits of both models.
- Examples include content delivery networks (CDNs) that use a combination of central servers and peer nodes to deliver content efficiently.

Cloud Computing Architecture:

- Cloud computing is a type of distributed computing that relies on virtualized resources provided over the internet.
- It typically involves large-scale data centers that host and manage computing resources (servers, storage, networking, etc.).
- Users access these resources remotely via the internet, paying for usage on a subscription or pay-per-use basis.
- Cloud architectures can be classified based on service models (e.g., Infrastructure as a Service, Platform as a Service, Software as a Service) and deployment models (e.g., public, private, hybrid, multicloud).

Result and Discussion:

Explain Distributed Computing Architecture of University

In a university setting, a distributed computing architecture typically involves various departments, campuses, and administrative offices interconnected through a network. Here's a simplified explanation:

- Client-Server Architecture:
  - Each department or faculty may have its own servers hosting academic resources such as course materials, student databases, and research databases.
  - These servers act as central points for storing and accessing data, while students and faculty members (clients) interact with them to retrieve information and perform tasks.
- Hybrid Architecture:
  - Universities may also employ a hybrid architecture, combining elements of client-server and peer-to-peer models.
  - For instance, collaborative research projects may involve peer-to-peer sharing of data and resources among researchers across different departments or institutions, facilitated by central servers for authentication and coordination.
- Cloud Computing Architecture:
  - Some universities may leverage cloud computing services for hosting virtual learning environments, email systems, and collaborative platforms.
  - Cloud-based infrastructure can provide scalability, flexibility, and cost-effectiveness, allowing universities to adapt to changing demands and support large-scale computing needs.

Explain Distributed Computing Architecture of Banking

In the banking sector, distributed computing architecture is crucial for handling transactions, managing customer data securely, and ensuring high availability. Here's a brief overview:

- Client-Server Architecture:
  - Banks operate a network of branches, ATMs, and online banking platforms where clients interact with banking systems.

- Centralized servers host core banking applications, databases containing customer accounts and transaction records, and security systems for authentication and authorization.

- Cloud Computing Architecture:
  - Many banks are adopting cloud computing to enhance scalability, data analytics capabilities, and disaster recovery mechanisms.
  - Cloud-based solutions allow banks to offload infrastructure management tasks, optimize resource utilization, and rapidly deploy new services while complying with stringent security and regulatory requirements.
- Hybrid Architecture:
  - Banking systems often employ a hybrid architecture to balance the need for centralized control with the benefits of distributed processing.
  - For example, a bank may maintain central servers for core banking operations while using peer-to-peer communication between branches for real-time transaction processing and data synchronization.

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:

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

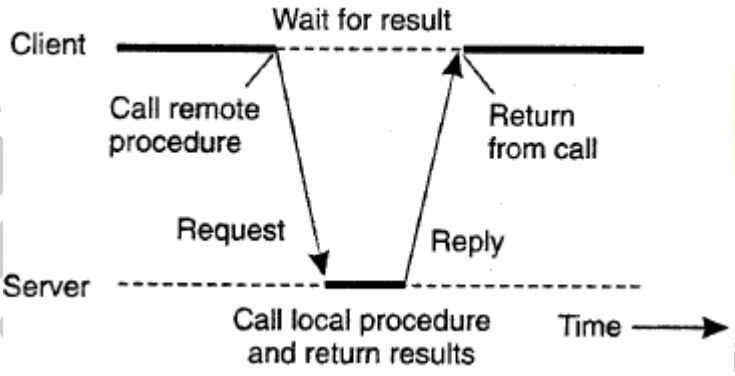
Theory:

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.

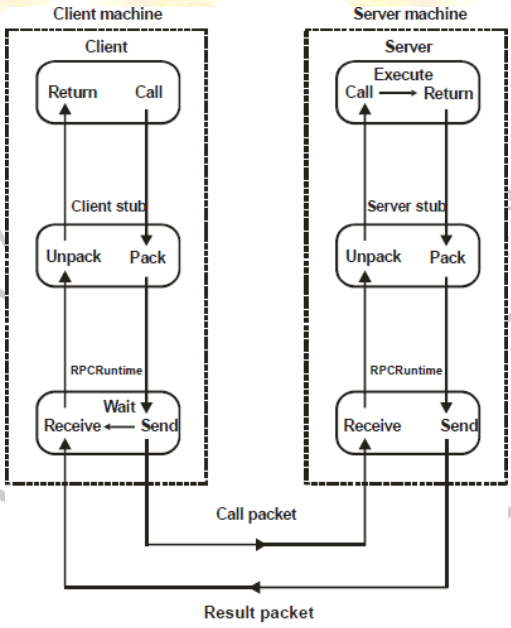


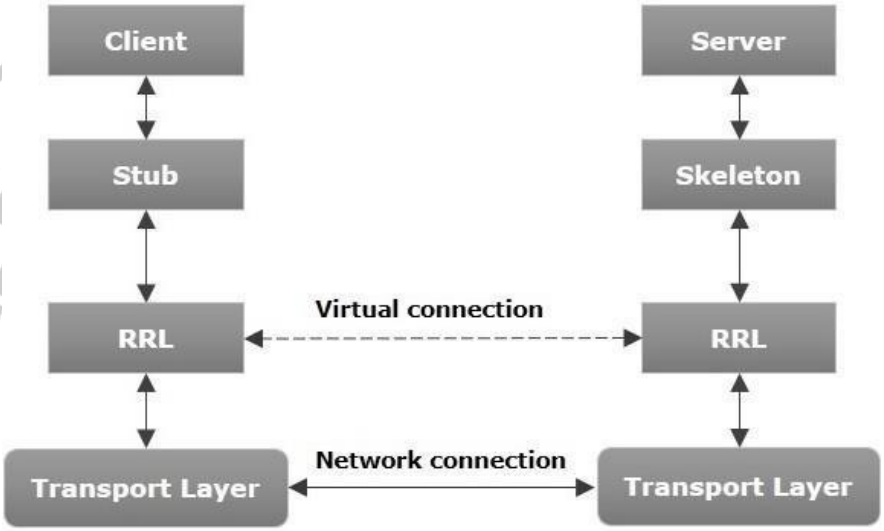
Fig. 3.2 : Implementation of RPC mechanism

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.

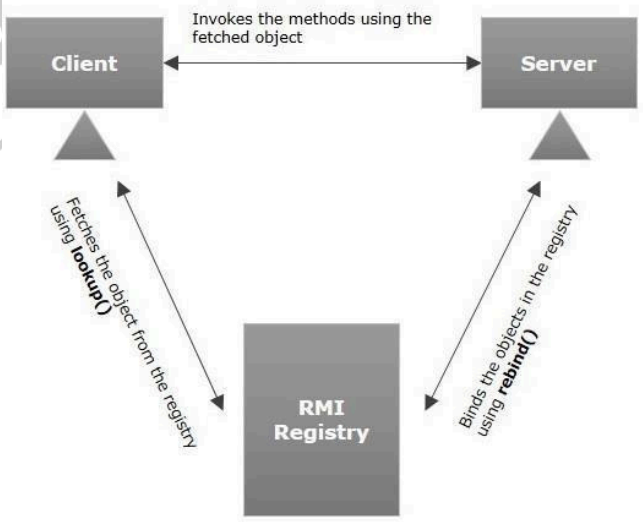
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling.

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

Implementation:

RMI_inteface.java

```
package pkg_RMI;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface RMI_interface {
public void displayMessage() throws RemoteException;
}
```

RMI_Server.java

```
package pkg_RMI;
import java.rmi.server.UnicastRemoteObject;
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 static int i = 0;
public RMI_Server() throws RemoteException {
super();
}
public static void main(String[] args) throws RemoteException, AlreadyBoundException
{
try {
Registry registry = LocateRegistry.createRegistry(1878);
registry.bind("hello", new RMI_Server());
System.out.println("The RMI_Server is running and ready...");
} catch (Exception e) {
System.out.println("The RMI_Server is not running...");
}
}
@Override
public void displayMessage() throws RemoteException {
System.out.println("-----------------------------------------------");
i++;
System.out.println("No. of calls : "+i);
System.out.println("-----------------------------------------------");
}
}
```

RMI_Client.java

```
package pkg_RMI;
import java.net.MalformedURLException;
import java.rmi.RemoteException;
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:1878/hello");
helloAPI.displayMessage();
} catch (Exception e) {
System.out.println("The RMI APP is not running...");
e.printStackTrace();
}
}
}
```

Output

```
[admin@archlinux SE4]$ javac pkg_RMI/RMI_Server.java
^[[A[admin@archlinux SE4]$ java pkg_RMI/RMI_Server
The RMI_Server is running and ready...
-----------------------------------------------
No. of calls : 0
-----------------------------------------------
-----------------------------------------------
No. of calls : 1
-----------------------------------------------
-----------------------------------------------
No. of calls : 2
-----------------------------------------------
```

```
[admin@archlinux SE4]$ javac pkg_RMI/RMI_Client.java
[admin@archlinux SE4]$ java pkg_RMI/RMI_Client
[admin@archlinux SE4]$ java pkg_RMI/RMI_Client
[admin@archlinux SE4]$ java pkg_RMI/RMI_Client
[admin@archlinux SE4]$ java pkg_RMI/RMI_Client
[admin@archlinux SE4]$
```

Learning Outcomes:
LO1: Understand the basics of Architecture of Distributed Computing.
LO2: Study the Architecture of University and Banking.
LO3: Develop the ability to design and implement a client/server program using RPC/RMI in Java.

Course Outcomes:
Upon completion of the course, students will be able to:
CO1: Explain the fundamental concepts and principles of distributed computing architecture.

CO2: Analyze and compare distributed computing architectures in real-world scenarios, such as those found in university and banking systems.
CO3: Design, develop, and deploy a client/server program using Remote Procedure Call (RPC) or Remote Method Invocation (RMI) in Java.

Result and Discussion: The provided theory and implementation details outline the concepts of Remote Procedure Call (RPC) and Remote Method Invocation (RMI) in distributed computing, highlighting their significance in inter-process communication. Through the provided Java code snippets, the practical implementation of RPC and RMI is demonstrated, offering insights into client-server interaction. The discussion showcases the importance of understanding distributed computing architectures in various domains like universities and banking systems, emphasizing the ability to design and deploy robust client/server programs using RPC or RMI in Java. This comprehensive approach enables learners to grasp fundamental concepts, analyze real-world scenarios, and apply practical skills, aligning with the course objectives of understanding, analyzing, designing, and implementing distributed computing solutions.

Conclusion:

For Faculty Use

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

Experiment 3 – Interprocess Communication

Learning Objective: Student should be able to write program to implement interprocess coummunication

Tools: Java/Python

Theory:

Interprocess communication (IPC) refers to the mechanisms used by processes to communicate and synchronize with each other. It allows different processes running on the same computer or different computers to exchange data and coordinate their actions. Here's a brief overview of IPC:

Types of IPC:

- Shared Memory: Processes can share a region of memory, allowing them to read from and write to the same memory locations.
- Message Passing: Processes communicate by sending and receiving messages through various mechanisms such as pipes, sockets, message queues, or signals.
- Synchronization: IPC mechanisms also include synchronization primitives like semaphores, mutexes, and condition variables to coordinate access to shared resources and ensure consistency.

Purpose of IPC:

- Data Sharing: IPC enables processes to share data efficiently, allowing them to collaborate on tasks or exchange information.
- Process Coordination: Processes can use IPC to coordinate their activities, synchronize execution, and avoid conflicts in accessing shared resources.
- Communication: IPC facilitates communication between processes, enabling them to send messages, signals, or other notifications to each other.

Common IPC Mechanisms:

- Shared Memory: Processes map a shared region of memory into their address space, allowing them to read from and write to the same memory locations. This mechanism offers high performance but requires careful synchronization to avoid data corruption.
- Message Passing: Processes exchange messages through various communication channels such as pipes, sockets, message queues, or signals. Message passing provides a more structured approach to communication and is often used in distributed systems.
- Synchronization Primitives: IPC mechanisms also include synchronization primitives like semaphores, mutexes, and condition variables. These primitives help coordinate access to shared resources, prevent race conditions, and ensure consistency in concurrent programs.

Use Cases of IPC:

- Inter-Process Communication: IPC is commonly used in multi-process applications where different processes need to exchange data or coordinate their actions.
- Client-Server Communication: IPC enables communication between clients and servers in distributed systems, allowing them to exchange requests and responses over the network.
- Concurrency Control: IPC mechanisms like semaphores and mutexes are used for concurrency control in multi-threaded and multi-process applications to ensure mutual exclusion and synchronization.

Program:

```
from multiprocessing import Process, Queue
import os
```

```
def child_process(queue):
    message = "Hello from the child process! (PID: {})".format(os.getpid())
    queue.put(message)
if __name__ == '__main__':
    queue = Queue()
    child = Process(target=child_process, args=(queue,))
    child.start()
    child.join()
    message_from_child = queue.get()
    print("Message received from the child process:", message_from_child)
```

Output:

```
Message received from the child process: Hello from the child process! (PID: 26636)

[Done] exited with code=0 in 0.122 seconds
```

Result and Discussion: IPC facilitates efficient communication and synchronization between processes. It ensures concurrency control and is essential for multi-process applications and distributed systems. While offering benefits, IPC also poses challenges like synchronization overhead and deadlock risks. The choice of IPC mechanism impacts scalability and performance, requiring careful consideration. Overall, understanding IPC is crucial for designing reliable and efficient systems.

Learning Outcomes: The student should have the ability to

LO1: Describe the protocol for Inter process communication.

LO 2: justify the 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:

For Faculty Use

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

Experiment 4–Group Communication

Learning Objective: Student should be able to develop a program for Group Communication

Tools : Java/Python

Theory:

Group Communication

The most elementary form of message-based interaction is one-to-one communication (also known as point-to-point, or unicast, communication) in which a single-sender process sends a message to a single-receiver process. However, for performance and ease of programming, several highly parallel distributed applications require that a message passing system should also provide group communication facility.

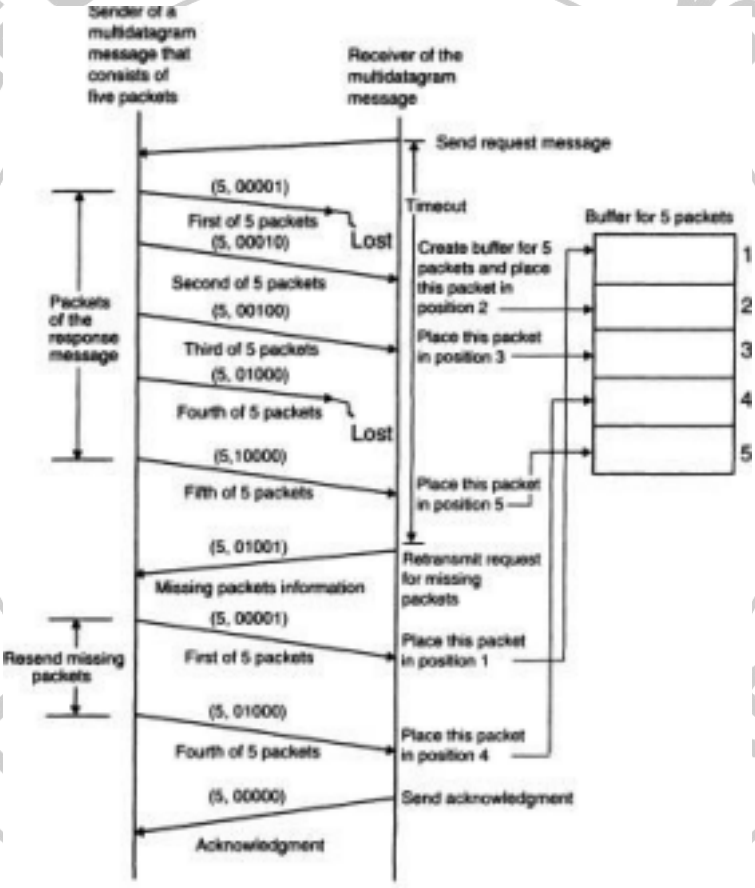


Fig. 3.13 An example of the use of a bitmap to keep track of lost and out of sequence packets in a multidatagram message transmission.

Depending on single or multiple senders and receivers, the following three types of group communication are possible:

1. One to many (single sender and multiple receivers)

2. Many to one (multiple senders and single receiver)

3. Many to many (multiple senders and multiple receivers)

One-to-Many Communication

In this scheme, there are multiple receivers for a message sent by a single sender. One-to-many scheme is also known as multicast communication. A special case of multicast communication is broadcast communication, in which the message is sent to all processors connected to a network. Multicast/broadcast communication is very useful for several practical applications.

For example, consider a server manager managing a group of server processes all providing the same type of service. The server manager can multicast a message to all the server processes, requesting that a free server volunteer to serve the current request. It then selects the first server that responds. The server manager does not have to keep track of the free servers. Similarly, to locate a processor
providing a specific service, an inquiry message may be broadcast. In this case, it is not necessary to receive an answer from every processor; just finding one instance of the desired service is sufficient.

Many-to-One Communication

In this scheme, multiple senders send messages to a single receiver. The single receiver may be selective or nonselective. A selective receiver specifies a unique sender; a message exchange takes place only if that sender sends a message. On the other hand, a nonselective receiver specifies a set of senders, and if anyone sender in the set sends a message to this receiver, a message exchange takes place.

Thus we see that an important issue related to the many-to-one communication scheme is nondeterminism. The receiver may want to wait for information from any of a group of senders, rather than from one specific sender. As it is not known in advance which member (or members) of the group will have its information available first, such behavior is nondeterministic. In some cases it is useful to dynamicalJy control the group of senders from whom to accept message. For example, a buffer process may accept a request from a producer process to store an item in the buffer whenever the buffer is not full; it may accept a request from a consumer process to get an item from the buffer whenever the buffer is not empty. To program such behavior, a notation is needed to express and control nondeterminism. One such construct is the "guarded command" statement introduced by Dijkstra

Many-to-Many Communication

In this scheme, multiple senders send messages to multiple receivers. The one-to-many and many-to one schemes are implicit in this scheme. Hence the issues related to one-to-many and many-to-one schemes, which have already been described above, also apply to the many-to-many communication scheme. In addition, an important issue related to many-to-many communication scheme is that of ordered message delivery.

Ordered message delivery ensures that all messages are delivered to all receivers in an order acceptable to the application. This property is needed by many applications for their correct functioning. For example, suppose two senders send messages to update the same record of a

database to two server processes having a replica of the database. If the messages of the two senders are received by the two servers in different orders, then the final values of the updated record of the database may be different in its two replicas. Therefore, this application requires that all messages be delivered in the same order to all receivers.

Ordered message delivery requires message sequencing. In a system with a single sender and multiple receivers (one-to-many communication), sequencing messages to all the receivers is trivial. If the sender initiates the next multicast transmission only after confirming that the previous multicast message has been received by all the members, the messages win be delivered in the same order. On the other hand. in a system with multiple senders and a single receiver (many-to-one communication),
the messages will be delivered to the receiver in the order in which they arrive at the receiver's machine.

Ordering in this case is simply handled by the receiver. Thus we see that it is not difficult to ensure ordered delivery of messages in many-to-one or one-to-many communication schemes.

However, in many-to-many communication, a message sent from a sender may arrive at a receiver's destination before the arrival of a message from another sender; but this order may be reversed at another receiver's destination (see Fig. 3.14). The reason why messages of different senders may arrive at the machines of different receivers in different orders is that when two processes are contending for access to a LAN, the order in which messages of the two processes are sent over the LAN is nondeterministic. Moreover, in a WAN environment, the messages of different senders may be routed to the same destination using different routes that take different amounts of time (which cannot be correctly predicted) to the destination. Therefore, ensuring ordered message delivery requires a special message-handling mechanism in many-to-many communication scheme.

The commonly used semantics for ordered delivery of multicast messages are absolute ordering, consistent ordering, and causal ordering.

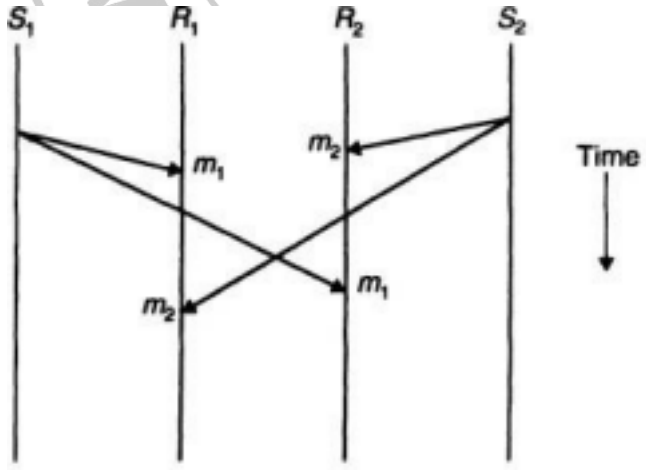


Fig. 3.14 No ordering constraint for message delivery.

Exercise:

1. What is Group Addressing?

Group addressing refers to addressing multiple recipients with a single message, allowing efficient communication within a group of processes or systems.

2. What is Multicasting? List different types of Multicast?

Multicasting involves sending a message from one sender to multiple receivers simultaneously. Types of multicast include source-specific multicast (SSM) and any-source multicast (ASM).

3. Explain Atomic Multicast?

Atomic multicast ensures that either all multicast messages are delivered to all receivers or none at all, ensuring consistency in distributed systems.

4. Explain types of semantics for for one-to-many communications?

Semantics for one-to-many communications include absolute ordering, consistent ordering, and causal ordering, ensuring messages are delivered in a specified order to maintain system integrity.

5. What is Marshalling And Unmarshalling?

Marshalling is the process of converting complex data structures into a format suitable for transmission, while unmarshalling is the reverse process of reconstructing the data structures upon receipt.

Implementation:

```
import threading

import time

class GroupCommunication:

    def __init__(self):

        self.messages = []

        self.lock = threading.Lock()

        self.terminate_flag = threading.Event()

    def send(self, message):

        with self.lock:
```

```
            self.messages.append(message)
    def receive(self):
        while not self.terminate_flag.is_set() or self.messages:
            with self.lock:
                if self.messages:
                    print("Received message:", self.messages.pop(0))
            time.sleep(1)
    def stop_receiver(self):
        self.terminate_flag.set()


def sender(group_comm):
    for i in range(5):
        group_comm.send(f"Message {i}")
        time.sleep(0.5)
    group_comm.stop_receiver()
def main():
    group_comm = GroupCommunication()
    receiver = threading.Thread(target=group_comm.receive)
    sender_thread = threading.Thread(target=sender, args=(group_comm,))


    receiver.start()
    sender_thread.start()


    sender_thread.join()
    receiver.join()
    print("Group communication completed.")
```

```
if __name__ == "__main__":

    main()
```

```
Received message: Message 0
Received message: Message 1
Received message: Message 2
Received message: Message 3
Received message: Message 4
Group communication completed.

[Done] exited with code=0 in 6.063 seconds
```

Result and Discussion:

The study explores group communication types such as one-to-many, many-to-one, and many-to-many schemes, highlighting multicast and broadcast functionalities. It underscores the significance of ordered message delivery and nondeterminism in distributed systems design, crucial for ensuring data consistency and system reliability.

Learning Outcomes: The student should have the ability to

LO1: Describe the protocol for Group communication.

LO2: Compare the different protocol techniques used in group communication.

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

Conclusion:

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 the currently 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).

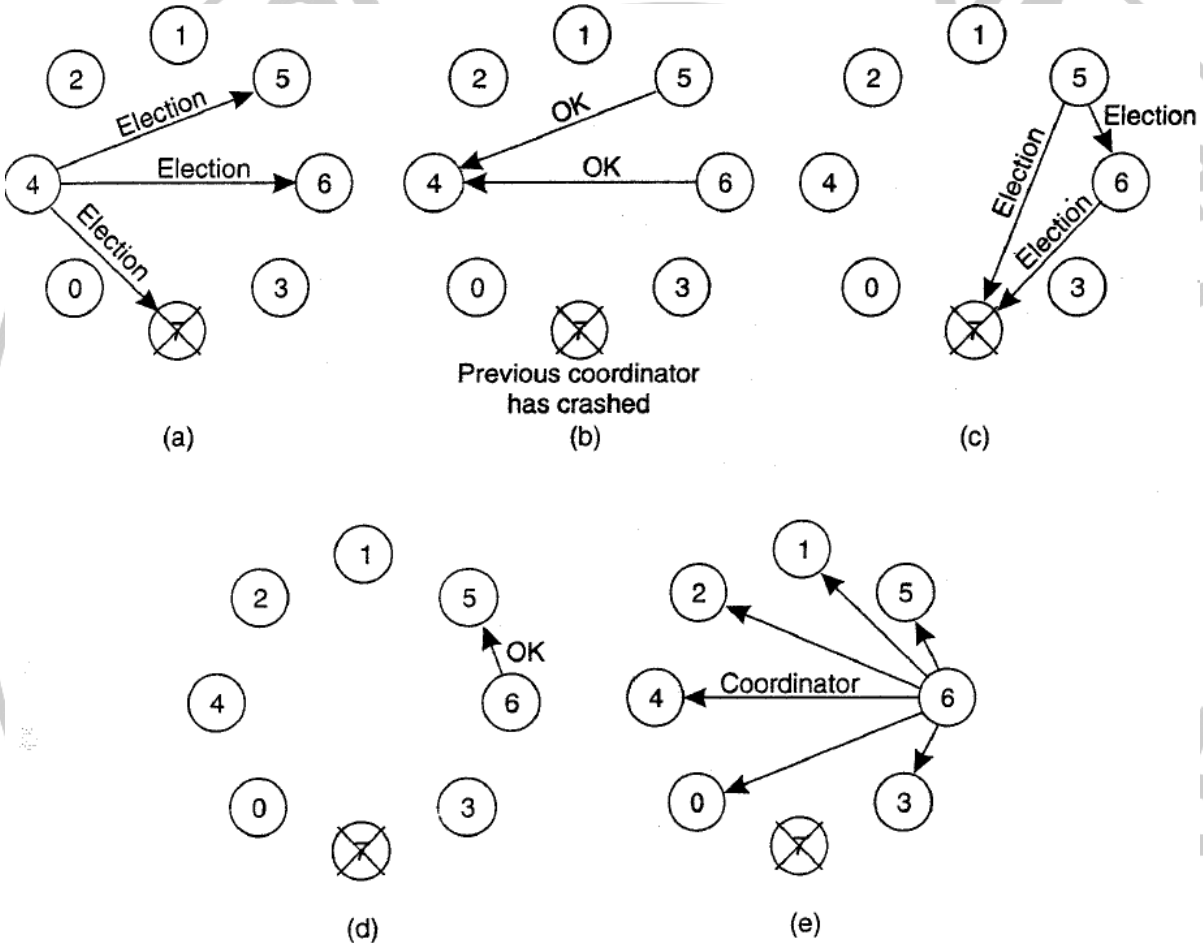


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

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.

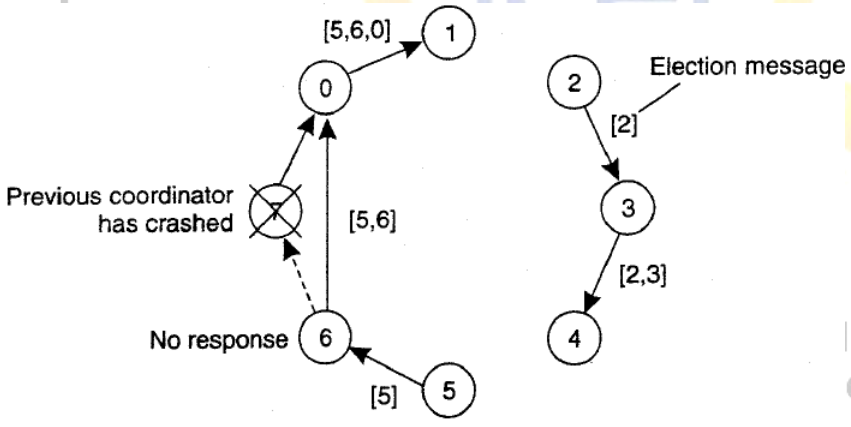


Figure 6-21. Election algorithm using a ring.

Implementation

```
import time
import random

class Node:
    def __init__(self, node_id):
        self.node_id = node_id
```

```
        self.leader_id = None
        self.nodes = set()

    def add_node(self, node):
        self.nodes.add(node)

    def start_election(self):
        print(f"Node {self.node_id} starts election")
        higher_nodes = [node for node in self.nodes if node.node_id > self.node_id]
        if not higher_nodes:
            self.declare_leader()
        elif all(not node.ping() for node in higher_nodes):
            print(f"No higher nodes responded. Node {self.node_id} declares itself as leader")
            self.declare_leader()

    def ping(self):
        return random.choice([True, False])

    def declare_leader(self):
        self.leader_id = self.node_id
        print(f"Node {self.node_id} is the new leader")

def simulate(nodes, rounds):
    print("Simulation starts\n")
    for _ in range(rounds):
        time.sleep(2)
        leader = next((node for node in nodes if node.leader_id is not None), None)
        if leader is None or not leader.ping():
            print("Initiating election process...")
            nodes.sort(key=lambda x: x.node_id, reverse=True)
            for node in nodes:
                node.start_election()
            print("Election process completed")
        else:
            print(f"Leader is Node {leader.leader_id}. Checking leader status...")
        time.sleep(5)

if __name__ == "__main__":
    nodes = [Node(i) for i in range(1, 5)]
    for i, node in enumerate(nodes):
        node.add_node(nodes[(i + 1) % len(nodes)])
    simulate(nodes, 5)  # Running the simulation for 5 rounds
```

Output:

```
Simulation starts

Initiating election process...
Node 4 starts election
Node 4 is the new leader
Node 3 starts election
No higher nodes responded. Node 3 declares itself as leader
Node 3 is the new leader
Node 2 starts election
Node 1 starts election
Election process completed
Initiating election process...
Node 4 starts election
Node 4 is the new leader
Node 3 starts election
Node 2 starts election
Node 1 starts election
Election process completed
Leader is Node 4. Checking leader status...
Leader is Node 4. Checking leader status...
Leader is Node 4. Checking leader status...
```

Result and Discussion:

The experiment implemented the Bully Election Algorithm

in a distributed system simulation with Python threads.

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

Conclusion:……

……

for Faculty Use

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