Experiment 01: Study of Distributed Computing system architecture and Applications of Disputed Computing.

Learning Objective: Student should be able to understand concepts of distributed computing and apply them to explain various applications

Tools: MS Word

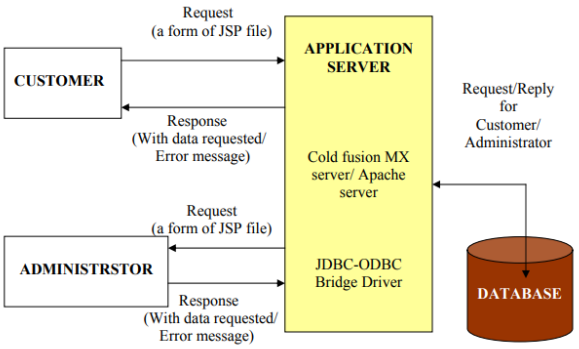
Theory:

Introduction to Distributed Computing: Distributed computing is a model in which components of a software system are shared among multiple computers or nodes. Even though the software components may be spread out across multiple computers in multiple locations, they're run as one system. This is done to improve efficiency and performance. The systems on different networked computers communicate and coordinate by sending messages back and forth to achieve a defined task. Distributed computing can increase performance, resilience and scalability, making it a common computing model in database and application design.

Goals of Distributed Computing: Performance. Distributed computing can help improve performance by having each computer in a cluster handle different parts of a task simultaneously. Scalability. Distributed computing clusters are scalable by adding new hardware when needed. Resilience and redundancy. Multiple computers can provide the same services. This way, if one machine isn't available, others can fill in for the service. Likewise, if two machines that perform the same service are in different data centers and one data center goes down, an organization can still operate. Cost-effectiveness. Distributed computing can use low-cost, off-the-shelf hardware. Efficiency. Complex requests can be broken down into smaller pieces and distributed among different systems. This way, the request is simplified and worked on as a form of parallel computing, reducing the time needed to compute requests. Distributed applications. Unlike traditional applications that run on a single system, distributed applications run on multiple systems simultaneously.

Case study: Distributed Computing Application for Online Banking: In today's world of emerging technologies, enterprises are moving towards the Internet for businesses. People are rushing towards the e-commerce applications for their day-today needs, which in turn are making the Internet very popular. Online Banking has given both an opportunity and a challenge to traditional banking. In the fast growing world, banking is a necessity, which in turn takes a lot of time from our busy schedule. Going to a branch or ATM or paying bills by paper check and mailing them out, and balancing checkbooks are all time-consuming tasks. Banking online automates many of these processes, saving time and money. For all banks, online banking is a powerful tool to gain new customers while it helps to eliminates costly paper handling and manual teller interactions in an increasingly competitive banking environment. Banks have spent generations gaining trust of their customers, and the goal for this project is to develop a user friendly, secure Online Banking Application. The application will be built using Java Server Pages (JSP), tomcat as the application server, and Microsoft Access / SQL Server as a database.

Architecture: The Online Banking Application is based on 3-tiered model. The Enterprise architecture for Online Banking Application is shown below.



The 3-tiered architecture shown above has the following major components:

- Client: There will be two clients for the application. One will be a web-based user-friendly client called bank customers. The other will be for administration purposes.
- Application Server: It takes care of the server script, takes care of JDBC-ODBC driver, and checks for the ODBC connectivity for mapping to the database in order to fulfill client and administrator's request.
- Database: Database Servers will store customer's and bank data.

Simply stated, the application works based on a request/response protocol. A client initiates a request to the server. The server responds by executing the business logic hosted inside the JSP program and if required, communicates with the Database Server to fulfill a client's request.

The Online Banking Application project will be divided into 4 modules namely:

- Bank Account
- Bank Account Administrator
- Credit Card Customer
- Credit Card Account Administrator

Similar banking applications available in the market A considerable amount of research has been done in the past few months on this project. Many banks had migrated from paper based banking system to electronic / online banking. Each bank had its own, user friendly interface, which helps its customers to interact with their account at their ease. A wide variety of online banking applications are available in the market, which in turn help the bank to function smoothly without reducing the quality of service. All banks which are using online banking application use the same basic principle. National City Bank has excellent features, which allows customers to check their accounts and view their statements. The best thing about this bank's system allows us to schedule payments and do online transactions. The security feature is the best; it gains the trust of the customer and allows them to do their transitions in an efficient and secure manner. However, the interface is very complicated for novice users. The interface for credit card customer is confusing. Bank One is the fastest growing bank in United States with millions of customers, who perform their transactions online. The security issues are wonderful, and it allows the customer to view their transactions, pay bills online, ATM/branch locator and provide 4 calculators and educators that will help customers to determine savings, mortgages and loan amounts. Educators are learning materials covering several financial topics which help the customer to learn more about the facilities that the bank has for them. The user interface is not as useful for the novice customers. TCF bank Online banking is a safe, fast and convenient way to access the accounts. It has all functionality but was lacking in user interface.

Advantages of Distributed Computing: Distributed computing includes the following benefits:

- Performance. Distributed computing can help improve performance by having each computer in a cluster handle different parts of a task simultaneously.
- Scalability. Distributed computing clusters are scalable by adding new hardware when needed.
- Resilience and redundancy. Multiple computers can provide the same services. This way, if one machine isn't available, others can fill in for the service. Likewise, if two machines that perform the same service are in different data centers and one data center goes down, an organization can still operate.
- Cost-effectiveness. Distributed computing can use low-cost, off-the-shelf hardware.
- Efficiency. Complex requests can be broken down into smaller pieces and distributed among different systems. This way, the request is simplified and worked on as a form of parallel computing, reducing the time needed to compute requests.
- Distributed applications. Unlike traditional applications that run on a single system, distributed applications run on multiple systems simultaneously.

Learning Outcomes: Students should have the ability to

LO1: Understand the basics of Architecture of Distributed Computing.

LO2: Studied Architecture of an application used in Distributed Computing.

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

Conclusion: We were able to Understand the basics of Architecture of Distributed Computing, its goals, and advantages and also Studied Architecture of an application used in Distributed Computing.

Viva Questions:

1. List and explain Distributed Computing Architecture
2. Explain the steps used to develop architecture of university or banking, etc.

For Faculty Use:

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

Experiment 2

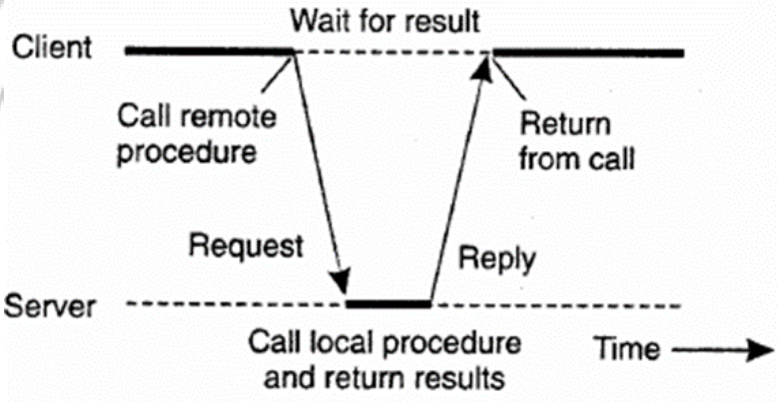
Aim: Build a program for client/server using RPC/RMI

Tools: Java / python

Theory:

Remote Procedure Call (RPC)

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.



RPC implementation steps using python:

A popular choice for RPC in Python is gRPC, which is a high-performance RPC framework. gRPC supports multiple programming languages, making it easier to achieve communication between different platforms.

1. Defining the Service Interface:

    a. Create a .proto file to define the service interface, including remote methods and message types.
    b. Use protobuf syntax to define the service and message formats.

example.proto

```
1   syntax = "proto3";
2
3   package example;
4
5   service RemoteService {
6     rpc RemoteMethod (Request) returns (Response);
7   }
8
9   message Request {
10    string message = 1;
11  }
12
13  message Response {
14    string message = 1;
15  }
```

2. Generating Code:
    a. Use the gRPC protocol compiler (protoc) to generate client and server code in the desired programming languages from the .proto file.
    b. The generated code includes stubs for the client and service implementation for the server.
    c. Command for compilation: python -m grpc_tools.protoc -I. --python_out=. –grpc_python_out=. example.proto
3. Implementing the Server:
    a. Implement the server-side logic by defining the service methods and handling client requests.
    b. Start the gRPC server, specifying the port on which it will listen for incoming requests.
4. Creating the Client:
    a. Implement the client-side logic to communicate with the server using the generated client stub.
    b. Establish a connection to the server and invoke remote methods as needed.
5. Testing the Application:
    a. Start the server and ensure it is listening on the specified port.
    b. Run the client application and observe the communication between client and server.
    c. Verify that the expected data is transmitted between client and server.

Code:

server.py

server.py > RemoteServiceServicer > RemoteMethod > request

```python
import grpc
import example_pb2
import example_pb2_grpc
from concurrent import futures

class RemoteServiceServicer(example_pb2_grpc.RemoteServiceServicer):
    def RemoteMethod(self, request, context):
        return example_pb2.Response(message=f"Received: {request.message}")

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    example_pb2_grpc.add_RemoteServiceServicer_to_server(RemoteServiceServicer(), server)
    server.add_insecure_port("[::]:50051")
    server.start()
    server.wait_for_termination()

if __name__ == "__main__":
    serve()

```

client.py

client.py > ...

```python
import grpc
import example_pb2
import example_pb2_grpc

def run():
    with grpc.insecure_channel("175.175.2.8:50051") as channel:
        stub = example_pb2_grpc.RemoteServiceStub(channel)
        response = stub.RemoteMethod(example_pb2.Request(message="Hello, Server!"))
    print(f"Client received: {response.message}")

if __name__ == "__main__":
    run()

```

Output:

Received message on client

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\comp lal-325\Desktop\implementing-procedure-call> python client.py
Client received: Received: Hello, Server!
PS C:\Users\comp lal-325\Desktop\implementing-procedure-call>
```

Learning Objective: Students should have the ability to

LO1: Understand the difference between RPC/RMI implementation in distributed systems

LO2: Understand the implementation of RPC/RMI, emphasizing abstraction and message exchange in distributed applications

Course Outcomes: Upon Completion of this students will be able to understand the concept of RPC/RMI

Conclusion: By implementing remote method invocation (RMI/RPC), learners have gained insight into the foundational concepts of distributed systems, including message passing and network communication protocols. This practical describes the significance of remote service invocation in building scalable and interconnected applications.

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.

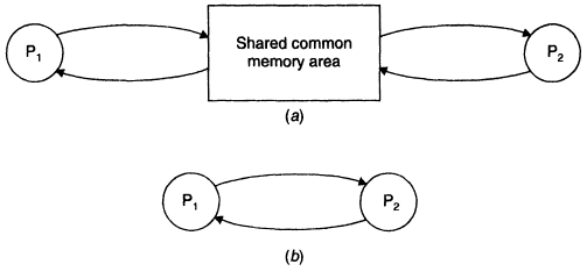


Fig. 3.1 The two basic interprocess communication paradigms: (*a*) The shared-data approach. (*b*) The message-passing approach.

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:

Code:

write.java:

```
package pkg_IPC;

import java.io.*;

public class write {
   public static void main(String[] args) {

   try{
      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
      PrintWriter writer = new PrintWriter(new FileWriter("data.txt"));
      String input;
         while ((input = reader.readLine()) != null) {
            writer.println(input);
```

```
            writer.flush();
        }
        reader.close();
        writer.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

read.java:

```
package pkg_IPC;

import java.io.*;

public class read {
  public static void main(String[] args) {
    try {
      BufferedReader reader = new BufferedReader(new FileReader("data.txt"));

      String input;
      while ((input = reader.readLine()) != null) {
        System.out.println("Received: " + input);
      }

      reader.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Output:

write.java:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\LAB-326\Desktop\dc exp 3>  & 'C:\Program Files\Java\jdk-21\bin\java.exe' '--enable-preview
LAB-326\AppData\Roaming\Code\User\workspaceStorage\f2fbec4cd1cdef8052bc747c5057295c\redhat.java\jdt_ws'
Hi This is Aditya Tiwari
```

read.java:

```
PS C:\Users\LAB-326\Desktop\dc exp 3>  c:; cd 'c:\Users\LAB-326\Desktop\dc exp 3'; & 'C:\Program Fil
DetailsInExceptionMessages' '-cp' 'C:\Users\LAB-326\AppData\Roaming\Code\User\workspaceStorage\f2fbe
79\bin' 'pkg_IPC.read'
Received: Hi This is Aditya Tiwari
PS C:\Users\LAB-326\Desktop\dc exp 3>
```

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:

For Faculty Use

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

Experiment 4–Group Communication

Aim: Student should be able to develop a program for Group communication

Tools : Java

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.

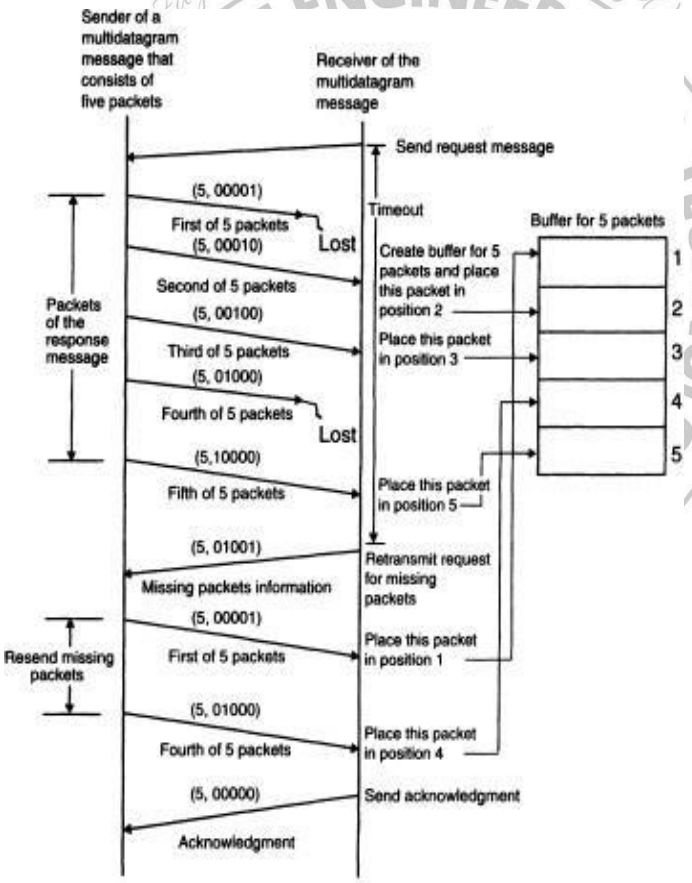


Fig. 3.13 An example of the use of a bitmap to keep track of lost and out of sequence packets in a multidatagram message transmission.

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.

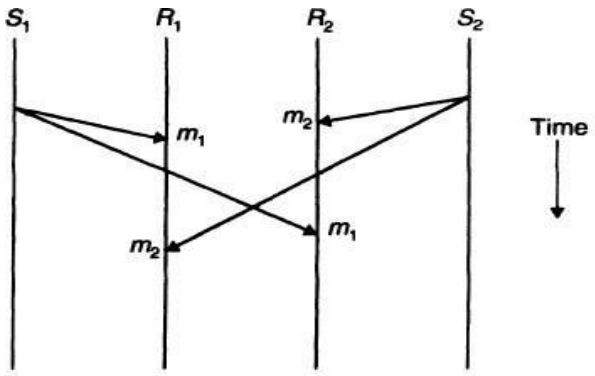


Fig. 3.14 No ordering constraint for message delivery.

Result and Discussion:

Server.java:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.io.PrintWriter;

public class Server {
   public static void main(String[] args) {
      try {
         ServerSocket serverSocket = new ServerSocket(12345);
         System.out.println("Server started. Waiting for clients...");

         while (true) {
            Socket clientSocket = serverSocket.accept();
            System.out.println("Client connected from: " +
clientSocket.getInetAddress().getHostName());
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            out.println("Hi this comp c 41");
            clientSocket.close();
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Client.java:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
```

```
public class Client {
   public static void main(String[] args) {
      try {
         Socket socket = new Socket("localhost", 12345);
         BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
         String message = in.readLine();
         System.out.println("Message from server: " + message);
         socket.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Client1.java:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

public class Client1 {
   public static void main(String[] args) {
      try {
         Socket socket = new Socket("localhost", 12345);
         BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
         String message = in.readLine();
         System.out.println("Message from server: " + message);
         socket.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Output:
Server.java

```
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.

C:\Users\comp lab-325>javac --version
javac 21.0.2

C:\Users\comp lab-325>javac Server.java

C:\Users\comp lab-325>java Server.java
Server started. Waiting for clients...
Client connected from: 127.0.0.1
Client connected from: 127.0.0.1
```

Client.java

```
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.

C:\Users\comp lab-325>javac Client.java

C:\Users\comp lab-325>java Client.java
Message from server: This is Experiment 4
```

Client.java

```
Microsoft Windows [Version 10.0.19045.4046]
(c) Microsoft Corporation. All rights reserved.

C:\Users\comp lab-325>javac Client1.java

C:\Users\comp lab-325>java Client1.java
Message from server: This is Experiment 4
```

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: After performing the experiment I was able to describe the protocol for group communication and also compare the different protocol techniques used in group communication.

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

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).

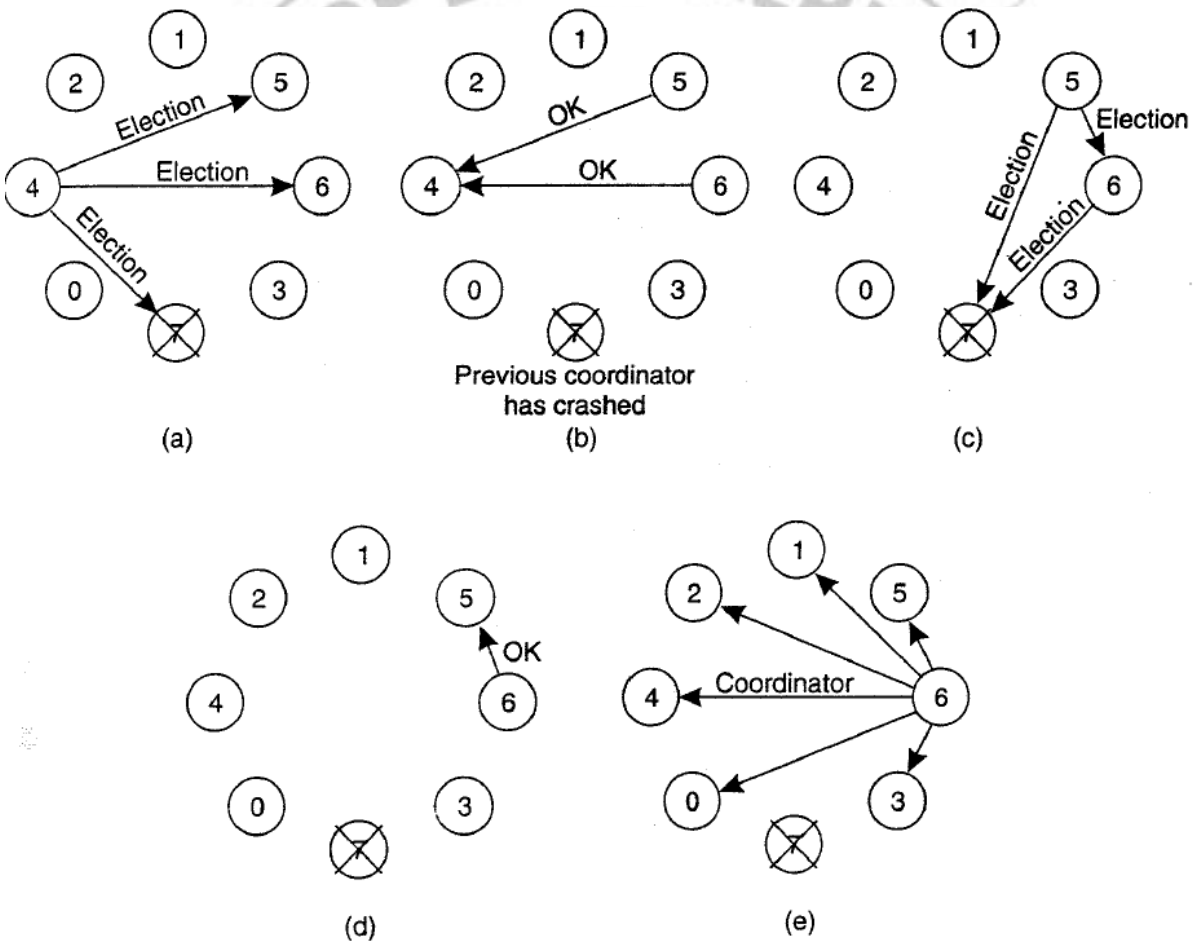


Figure 6-20. The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

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.

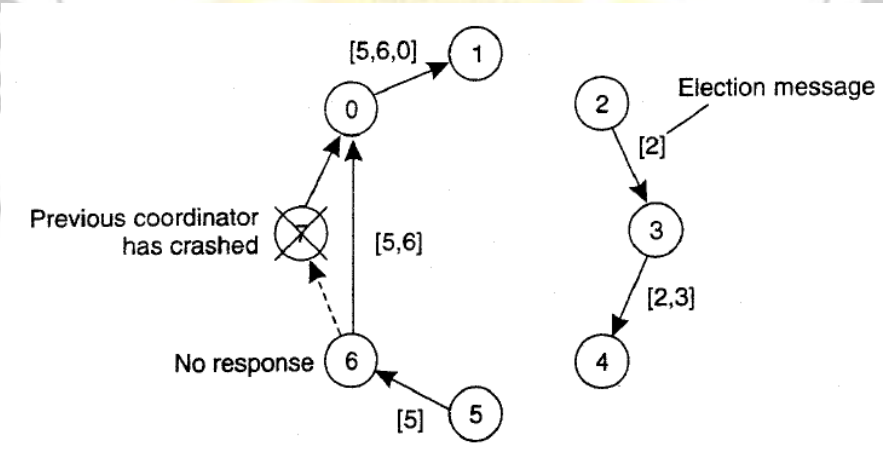


Figure 6-21. Election algorithm using a ring.

Result and Discussion: (Bully Algorithm)

Code:

```
import java.util.ArrayList;
import java.util.List;

class Process {
    private int id;
    private int priority;
    private boolean coordinator;

    public Process(int id, int priority) {
        this.id = id;
        this.priority = priority;
        this.coordinator = false;
    }

    public int getId() {
        return id;
    }

    public int getPriority() {
        return priority;
    }

    public boolean isCoordinator() {
        return coordinator;
    }

    public void setCoordinator(boolean coordinator) {
        this.coordinator = coordinator;
    }

    public void startElection(List<Process> processes) {
        System.out.println("Process " + id + " is starting election.");

        for (Process p : processes) {
            if (p.getPriority() > priority) {
                System.out.println("Process " + id + " sends election message to Process " +
p.getId());
                p.receiveElection();
            }
        }
    }

    public void receiveElection() {
```

```
        System.out.println("Process " + id + " receives election message.");
    }

    public void declareVictory(List<Process> processes) {
        Process newCoordinator = this;
        for (Process p : processes) {
            if (p.getPriority() > newCoordinator.getPriority()) {
                newCoordinator = p;
            }
        }
        for(Process p : processes){
            if(p!= newCoordinator){
                p.setCoordinator(false);
            }
        }
        newCoordinator.setCoordinator(true);
        System.out.println("Process " + newCoordinator.getId() + " is the new coordinator.");
    }

}

public class BullyAlgorithm {
    public static void main(String[] args) {

        List<Process> processes = new ArrayList<>();
        processes.add(new Process(1, 3));
        processes.add(new Process(2, 5));
        processes.add(new Process(3, 4));
        processes.add(new Process(4, 2));
        processes.add(new Process(5, 1));

        processes.get(0).setCoordinator(false);

        for (Process process : processes) {
            if (process.isCoordinator()) {
                System.out.println("Process " + process.getId() + " is already the coordinator.");
            } else {
                process.startElection(processes);
                break;
            }
        }

        processes.get(4).declareVictory(processes);

    }
}
```

Output:

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   SEARCH ERROR

PS C:\Users\LAB-326\Desktop>  c:; cd 'c:\Users\LAB-326\Desktop'; & 'C:\Program Files\Java\jdk1.8.0_3
'BullyAlgorithm'
Process 1 is starting election.
Process 1 sends election message to Process 2
Process 2 receives election message.
Process 1 sends election message to Process 3
Process 3 receives election message.
Process 2 is the new coordinator.
PS C:\Users\LAB-326\Desktop>
```

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				

Experiment-06

Aim: Develop a program for clock synchronization algorithms.

Tools: Python, Google Colab

Theory: Berkeley's Algorithm is a distributed algorithm for computing the correct time in a network of computers. The algorithm is designed to work in a network where clocks may be running at slightly different rates, and some computers may experience intermittent communication failures.

The basic idea behind Berkeley's Algorithm is that each computer in the network periodically sends its local time to a designated "master" computer, which then computes the correct time for the network based on the received timestamps. The master computer then sends the correct time back to all the computers in the network, and each computer sets its clock to the received time.

There are several variations of Berkeley's Algorithm that have been proposed, but a common version of the algorithm is as follows −

- Each computer starts with its own local time, and periodically sends its time to the master computer.
- The master computer receives timestamps from all the computers in the network.
- The master computer computes the average time of all the timestamps it has received and sends the average time back to all the computers in the network.
- Each computer sets its clock to the time it receives from the master computer.
- The process is repeated periodically, so that over time, the clocks of all the computers in the network will converge to the correct time.

One benefit of Berkeley's Algorithm is that it is relatively simple to implement and understand. However, It has a limitation that the time computed by the algorithm is based on the network conditions and time of sending and receiving timestamps which can't be very accurate and also it has a requirement of a master computer which if failed can cause the algorithm to stop working.

There are other more advanced algorithms such as the Network Time Protocol (NTP) which use a more complex algorithm and also consider the network delay and clock drift to get a more accurate time.

Result and Discussion:

Input:

```
from datetime import datetime, timedelta

def berkeley_algorithm(nodes):
    # Calculate the average time (converted to timestamps)
    average_time = sum(node['time'].timestamp() for node in nodes) /
len(nodes)

    # Calculate the time difference for each node
    for node in nodes:
        node['offset'] = average_time - node['time'].timestamp()
        node['synchronized_time'] = node['time'] +
timedelta(seconds=node['offset'])

def synchronize_clocks(nodes):
    for node in nodes:
        # Synchronize the clock for each node
        node['synchronized_time'] = node['time'] +
timedelta(seconds=node['offset'])

def print_node_times(nodes):
    for node in nodes:
        print(f"Node {node['id']} - Local Time: {node['time']},
Synchronized Time: {node.get('synchronized_time', 'Not synchronized')}")

if __name__ == "__main__":
    # Example with three nodes
    nodes = [
        {'id': 1, 'time': datetime.now()},
        {'id': 2, 'time': datetime.now() + timedelta(seconds=5)},
        {'id': 3, 'time': datetime.now() - timedelta(seconds=3)}
    ]

    print("Original Node Times:")
    print_node_times(nodes)

    berkeley_algorithm(nodes)
    synchronize_clocks(nodes)

    print("\nAfter Berkeley Algorithm:")
    print_node_times(nodes)
```

Output:

```
Original Node Times:
Node 1 - Local Time: 2024-02-26 09:34:37.229477, Synchronized Time: Not synchronized
Node 2 - Local Time: 2024-02-26 09:34:42.229481, Synchronized Time: Not synchronized
Node 3 - Local Time: 2024-02-26 09:34:34.229487, Synchronized Time: Not synchronized

After Berkeley Algorithm:
Node 1 - Local Time: 2024-02-26 09:34:37.229477, Synchronized Time: 2024-02-26 09:34:37.896148
Node 2 - Local Time: 2024-02-26 09:34:42.229481, Synchronized Time: 2024-02-26 09:34:37.896148
Node 3 - Local Time: 2024-02-26 09:34:34.229487, Synchronized Time: 2024-02-26 09:34:37.896148
```

Learning Outcomes: The student should have the ability to

LO1: Describe Berkeley's algorithm for clock synchronization
LO2: Write a program for clock synchronization algorithms.

Course Outcomes: Upon completion of the course students will be able to describe and implement clock synchronization algorithms

Conclusion: The basic implementation of the Berkeley algorithm for clock synchronization in a distributed system. It calculates the average time, determines time differences for each node, and synchronizes their clocks accordingly. The example involves three nodes with different local times, and the program prints the original and synchronized times after applying the Berkeley algorithm.

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 algorithm.

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 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.*;

class Site {
    int siteId;
    int[] RN;
    int[] LN;

    Site(int siteId, int numSites) {
        this.siteId = siteId;
        RN = new int[numSites];
        LN = new int[numSites];
    }
}

public class SuzukiKasami {
    static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.print("Enter the number of sites: ");
        int numSites = scanner.nextInt();
```

```
        System.out.print("Enter the site ID which initially has the token (1-"
+ numSites + "): ");
        int initialTokenSiteId = scanner.nextInt();
        if (initialTokenSiteId < 1 || initialTokenSiteId > numSites) {
            System.out.println("Invalid site ID for initial token holder.");
            return;
        }
        SuzukiKasami suzukiKasami = new SuzukiKasami(numSites,
initialTokenSiteId);
        suzukiKasami.simulate();
    }

    int numSites;
    Site[] sites;
    Queue<Integer> tokenQueue;
    int currentTokenHolder;

    SuzukiKasami(int numSites, int initialTokenSiteId) {
        this.numSites = numSites;
        sites = new Site[numSites];
        for (int i = 0; i < numSites; i++) {
            sites[i] = new Site(i + 1, numSites);
        }
        currentTokenHolder = initialTokenSiteId;
        tokenQueue = new LinkedList<>();
    }

    void simulate() {
        while (true) {
            System.out.println("Site " + currentTokenHolder + " has the
token.");
            // Execute critical section
            System.out.println("Site " + currentTokenHolder + " executes
critical section.");
            // Release critical section
            int[] currentLNs = sites[currentTokenHolder - 1].LN;
            int[] currentRNs = sites[currentTokenHolder - 1].RN;
            currentLNs[currentTokenHolder - 1] = currentRNs[currentTokenHolder
- 1];
            for (int j = 0; j < numSites; j++) {
                if (!tokenQueue.contains(j + 1) && currentRNs[j] ==
currentLNs[j] + 1) {
                    tokenQueue.offer(j + 1);
                }
            }
            if (!tokenQueue.isEmpty()) {
```

```
                int nextTokenHolder = tokenQueue.poll();
                System.out.println("Token sent from Site " +
currentTokenHolder + " to Site " + nextTokenHolder);
                currentTokenHolder = nextTokenHolder;
            } else {
                System.out.println("Site " + currentTokenHolder + " retains
the token.");
            }
            // Simulate next site requesting the token
            System.out.print("Enter the site ID that wants to enter the
critical section (1-" + numSites + "): ");
            int requestingSiteId = scanner.nextInt();
            if (requestingSiteId < 1 || requestingSiteId > numSites) {
                System.out.println("Invalid site ID.");
                return;
            }
            requestCriticalSection(requestingSiteId);
        }
    }

    void requestCriticalSection(int requestingSiteId) {
        System.out.println("Site " + requestingSiteId + " requests the
critical section.");
        sites[requestingSiteId - 1].RN[requestingSiteId - 1]++;
        for (int i = 0; i < numSites; i++) {
            if (i != requestingSiteId - 1) {
                sites[i].RN[requestingSiteId - 1] =
Math.max(sites[i].RN[requestingSiteId - 1],
                        sites[requestingSiteId - 1].RN[requestingSiteId - 1]);
                if (currentTokenHolder == i + 1 &&
sites[i].RN[requestingSiteId - 1] == sites[i].LN[requestingSiteId - 1] + 1) {
                    System.out.println("Token sent from Site " +
currentTokenHolder + " to Site " + requestingSiteId);
                    currentTokenHolder = requestingSiteId;
                }
            }
        }
    }
}
```

Output:

```
Enter the number of sites: 3
Enter the site ID which initially has the token (1-3): 1
Site 1 has the token.
Site 1 executes critical section.
Site 1 retains the token.
Enter the site ID that wants to enter the critical section (1-3): 2
Site 2 requests the critical section.
Token sent from Site 1 to Site 2
Site 2 has the token.
Site 2 executes critical section.
Site 2 retains the token.
Enter the site ID that wants to enter the critical section (1-3): 3
Site 3 requests the critical section.
Token sent from Site 2 to Site 3
Site 3 has the token.
Site 3 executes critical section.
Token sent from Site 3 to Site 2
```

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:

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.
- When ever a site make request for critical section, it gets a timestamp. Timestamp is also used to resolve any conflict between critical section requests.
- All algorithm which follows non-token based approach maintains a logical clock. Logical clocks get updated according to Lamport's scheme

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.

Result and Discussion:

Code:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

class Message {
    String messageType;
    int timestamp;
    int siteId;

    Message(String messageType, int timestamp, int siteId) {
        this.messageType = messageType;
        this.timestamp = timestamp;
        this.siteId = siteId;
```

```
    }
}

class Site {
    int siteId;
    boolean requesting;
    boolean executing;
    int timestamp;
    List<Message> deferredQueue;

    Site(int siteId) {
        this.siteId = siteId;
        this.requesting = false;
        this.executing = false;
        this.timestamp = 0;
        this.deferredQueue = new ArrayList<>();
    }

    void requestCriticalSection(List<Site> sites) {
        this.requesting = true;
        this.timestamp++;
        for (Site site : sites) {
            if (site.siteId != this.siteId) {
                Message requestMessage = new Message("REQUEST", this.timestamp, this.siteId);
                sendMessage(requestMessage, site);
            }
        }
        waitForReplies(sites);
    }

    void sendMessage(Message message, Site destination) {
        System.out.println("Site " + this.siteId + " sends " + message.messageType + " message
to Site " + destination.siteId);
        destination.receiveMessage(message, this);
    }

    void receiveMessage(Message message, Site sender) {
        System.out.println("Site " + this.siteId + " receives " + message.messageType + " message
from Site " + sender.siteId);
        if (message.messageType.equals("REQUEST")) {
            if (!this.requesting && !this.executing) {
                this.sendMessage(new Message("REPLY", 0, this.siteId), sender);
            } else if (this.requesting && message.timestamp < this.timestamp) {
                this.deferredQueue.add(message);
            }
        } else if (message.messageType.equals("REPLY")) {
            if (this.requesting) {
```

```
            this.deferredQueue.removeIf(m -> m.siteId == sender.siteId);
            if (this.deferredQueue.isEmpty()) {
                this.executing = true;
                System.out.println("Site " + this.siteId + " enters critical section.");
            }
        }
    }
}

void waitForReplies(List<Site> sites) {
    int repliesExpected = sites.size() - 1;
    int repliesReceived = 0;
    while (repliesReceived < repliesExpected) {
        // Wait for replies
    }
}

void releaseCriticalSection(List<Site> sites) {
    this.requesting = false;
    this.executing = false;
    for (Site site : sites) {
        if (site.siteId != this.siteId) {
            for (Message message : this.deferredQueue) {
                this.sendMessage(new Message("REPLY", 0, this.siteId), site);
            }
        }
    }
    this.deferredQueue.clear();
    System.out.println("Site " + this.siteId + " releases critical section.");
}
}

public class RicartAgrawalaAlgorithm {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the number of sites: ");
        int numberOfSites = scanner.nextInt();

        List<Site> sites = new ArrayList<>();
        for (int i = 0; i < numberOfSites; i++) {
            sites.add(new Site(i+1));
        }

        for (Site site : sites) {
            site.requestCriticalSection(sites);
            site.releaseCriticalSection(sites);
        }
```

```
  }
}
```

Output:

```
Enter the number of sites:
3
Site 1 sends REQUEST message to Site 2
Site 2 receives REQUEST message from Site 1
Site 2 sends REPLY message to Site 1
Site 1 receives REPLY message from Site 2
Site 1 enters critical section.
Site 1 sends REQUEST message to Site 3
Site 3 receives REQUEST message from Site 1
Site 3 sends REPLY message to Site 1
Site 1 receives REPLY message from Site 3
Site 1 enters critical section.
PS C:\Users\tiwar\OneDrive\Desktop>
```

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:

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.

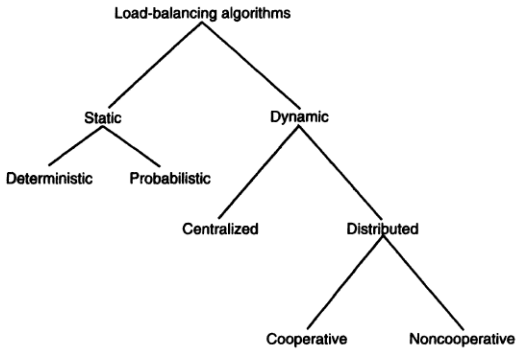


Fig. 7.3 A taxonomy of load-balancing algorithms.

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:

For Faculty Use

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

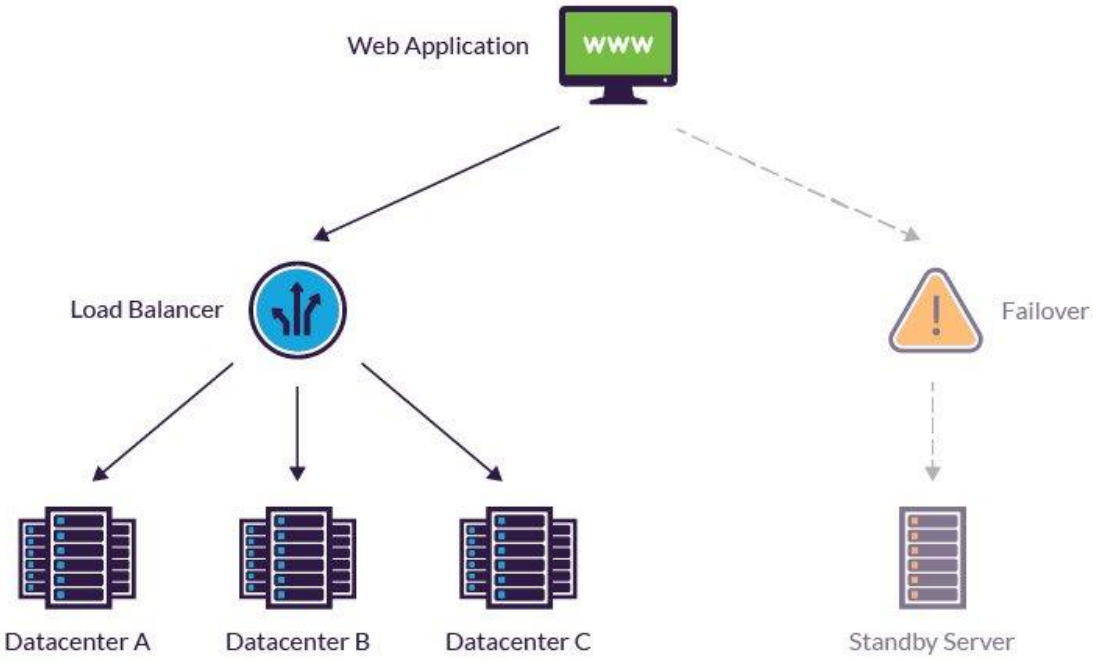
Experiment 10: Case Study on Fault Tolerance in Distributed Systems.

Learning Objective : Student should be able to examine how systems like Apache Hadoop and Apache Spark handle node failures and ensures the reliability of Computations **.**

Theory: Distributed systems are defined as a collection of multiple independent systems connected together as a single system. Every independent system has its own memory and resources and some common resources and peripheral devices that are common to devices connected together. The design of Distributed systems is a complex process where all the nodes or devices need to be connected together even if they are located at long distances. Challenges faced by distributed systems are Fault Tolerance, transparency, and communication primitives. Fault Tolerance is one of the major challenges faced by distributed systems.

Introduction: Distributed systems have revolutionized the way we process vast amounts of data, enabling parallelized computations across clusters of machines. However, as the scale of these systems increases, so does the likelihood of node failures. This case study delves into the intricate strategies employed by two leading distributed computing frameworks, Apache Hadoop and Apache Spark, to handle node failures and ensure the reliability of computations in large-scale environments.

System Architecture: Load balancing solutions enable applications to operate across multiple network nodes, mitigating the risk of a single point of failure. They optimize workload distribution, enhancing individual resilience to activity spikes and preventing slowdowns. In contrast, failover solutions are crucial during severe scenarios, like complete network failures. They automatically activate a secondary platform to keep applications running while the primary network is restored. For fault tolerance with zero downtime, "hot" failover instantly transfers workloads to a backup system. Alternatively, "warm" or "cold" failover involves a backup system with a delayed workload start-up.

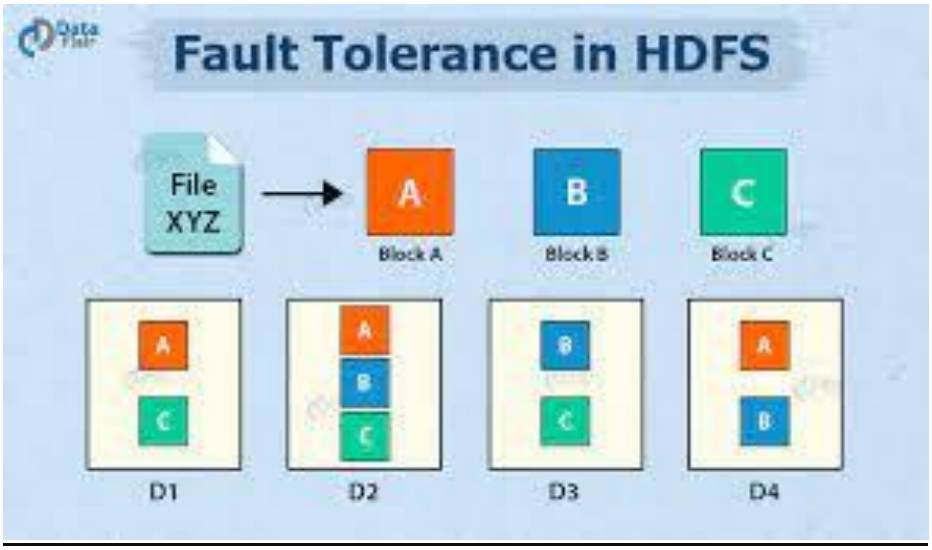


Background of Hadoop and Spark:

- Apache Hadoop: Known for its MapReduce programming model and Hadoop Distributed File System (HDFS), Hadoop divides large datasets into smaller chunks, processes them in parallel across a cluster, and stores them redundantly for fault tolerance.
- Apache Spark: Built on top of Hadoop, Spark introduces an in-memory data processing engine. It provides fault tolerance through lineage information, enabling the reconstruction of lost data partitions in case of node failures.

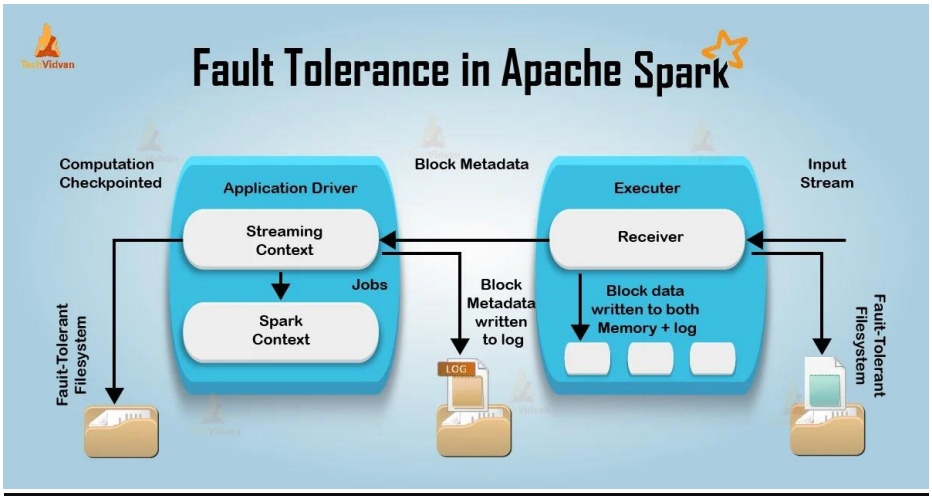
Fault Tolerance in Apache Hadoop:

- Data Replication: Hadoop achieves fault tolerance by replicating data across multiple nodes in the HDFS. Typically, data is replicated three times, and tasks are rerouted to replicas if a node fails during computation.
- Task Redundancy: MapReduce jobs in Hadoop consist of map and reduce tasks. Hadoop monitors task execution and reruns failed tasks on available nodes, ensuring the completion of the job.



Fault Tolerance in Apache Spark:

- Resilient Distributed Datasets (RDDs): Spark introduces RDDs, a fault-tolerant abstraction representing a distributed collection of data. RDDs maintain lineage information, allowing lost partitions to be recomputed from the original data.
- Directed Acyclic Graph (DAG): Spark jobs are represented as a directed acyclic graph (DAG) of stages. If a node fails, Spark can recompute only the affected partitions by referencing the lineage information in the DAG, minimizing the computational overhead.



Handling Node Failures:

- Hadoop: Upon node failure, Hadoop's ResourceManager reallocates tasks to healthy nodes. However, if the failed node contains data blocks, HDFS replicates the data to other nodes, ensuring availability.
- Spark: Spark's Driver and Cluster Manager monitor task execution. In case of node

failure, Spark recomputes lost partitions based on lineage information, ensuring the fault tolerance of computations.

Resource Management:

- Hadoop: Hadoop relies on a master-slave architecture, with the ResourceManager managing resources and the NodeManagers overseeing individual nodes.
- Spark: Spark utilizes a similar master-slave architecture with a Driver coordinating tasks and Executors managing node-level computations.

Comparative Analysis:

- Data Processing Speed: Spark generally outperforms Hadoop due to its in-memory processing capabilities, reducing the need for extensive disk I/O.
- Fault Tolerance Overheads: While both systems provide fault tolerance, Spark's lineage-based approach minimizes recomputation overhead compared to Hadoop's task reexecution.

Real world applications:

- Hadoop: Widely used for batch processing tasks, such as log analysis and data warehousing. Log Analysis: Suppose a large e-commerce platform wants to analyze its server logs to gain insights into user behavior. Hadoop can be employed to process and analyze these logs in batches, extracting valuable information such as popular products, user traffic patterns, and potential issues in the system.
- Spark: Suitable for iterative algorithms, machine learning, and interactive analytics, where in-memory processing is advantageous. Interactive Analytics: Consider a business intelligence scenario where an analyst needs to interactively explore and analyze data to make real-time decisions. Spark can be used for interactive analytics, allowing users to run queries, visualize data, and obtain insights on-demand. This is particularly beneficial in scenarios like marketing campaign performance analysis or real-time dashboards for monitoring.

Future considerations and Hybrid Approaches:

- The evolution of distributed computing is a dynamic process, and as technology advances, future considerations become pivotal for enhancing the efficiency and robustness of distributed systems.
- Two key areas of exploration involve the potential integration of hybrid approaches, combining the strengths of Apache Hadoop and Apache Spark, and the impact of advancements in hardware and network technologies on fault tolerance mechanisms.

The current landscape presents Apache Hadoop and Apache Spark as powerful yet distinct frameworks, each excelling in specific use cases.

- Future exploration may revolve around developing hybrid approaches that amalgamate the strengths of both frameworks. This could entail leveraging Hadoop's mature and reliable storage infrastructure while harnessing Spark's in-memory processing capabilities for enhanced computational speed. Hybrid architectures could be designed to dynamically allocate tasks based on the nature of computations. For instance, batch processing tasks might leverage Hadoop's efficiency, while more iterative and interactive workloads could benefit from Spark's faster in-memory processing.
- The seamless integration of these frameworks may require innovations in job scheduling, resource management, and data orchestration to optimize the overall performance of distributed systems.

Conclusion:

FOR FACULTY USE

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