Assignment 5

1. Problem Statement

1.1. Chandy-Lamport's Algorithm for Global State Recording

2. Source Code

2.1.>> Main. java

```
/* ABOUT PROJECT
 * This project demonstrates the Chandy-Lamport's global state recording algorithm
 * using threads to record the global state of distributed systems.
 */
/* ABOUT MAIN CLASS
 * The Main class serves as the core orchestrator for a simulation of distributed systems
 * using the Chandy-Lamport's global state recording algorithm in Java.
 * It begins by importing necessary libraries for file handling, UI interactions, and data
structures.
 * The main method initializes essential components such as lists for managing threads and
nodes,
 * and a Channel object for communication. Through a file dialog, users select a file containing
 * the graph structure of the distributed system, which is then parsed by the Graph class.
 * After prompting the user to specify an initiator node, the program validates this choice
against
 * the graph and proceeds to create and manage threads for each node. Threads are started
for both
 * the initiator and other nodes, with synchronization ensuring proper message exchange and
state recording.
 * Upon completion of thread execution, the program outputs a global snapshot, detailing the
 * messages exchanged and current states of each node and the communication channels.
 * Finally, the program gracefully exits, providing a comprehensive demonstration of the
 * Chandy-Lamport's algorithm application in distributed system simulation.
 */
/* ABOUT AUTHOR
    Rahul Biswas, Arijit Mondal
 */
import java.awt.FileDialog;
import java.awt.Frame;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Queue;

import javax.swing.JOptionPane;
```

```
public class Main {
    public static void main(String[] args) {
        // Lists to keep track of threads and nodes
        List<Thread> threads = new LinkedList<>();
        List<Node> nodes = new LinkedList<>();
        Channel channel = new Channel(); // Shared communication channel

        // Open file dialog to select a file
        String filePath = openFileDialog();
        if (filePath == null) {
            System.out.println("No file selected");
            System.exit(0);
        }

        // Create graph from the file
        Graph graph = new Graph(filePath);

        // Get initiator name from the user
        String initiator = JOptionPane.showInputDialog("Enter initiator:");

        // Validate initiator and get node connections
        Map<String, List<String>> nodeNames = graph.checkInitiator(initiator);
        if (nodeNames != null) {
            System.out.println();
            System.out.println("System Log ::::\n");

            // Create the initiator node and start its thread
            Node node = new Node(initiator, channel, nodeNames.get(initiator));
            Thread thread = new Thread(node);
            nodes.add(node);
            threads.add(thread);
            thread.start(); // Start the initiator thread
            node.setInitiatorStatus(); // Set the node as the initiator
            System.out.println("Information Message :: " + initiator + " is now initiator");

            // Pause for a short duration to ensure initiator starts
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            // Create and start threads for all other nodes
            for (String name : nodeNames.keySet()) {
                if (!name.equals(initiator)) {
                    Node n = new Node(name, channel, nodeNames.get(name));
                    Thread t = new Thread(n);
                    nodes.add(n);
                    threads.add(t);
```

```
            t.start(); // Start thread for each node
        }
    }

    // Wait for all threads to finish
    for (Thread td : threads) {
        try {
            td.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // Print global snapshot of the system
    System.out.println();
    System.out.println("Global Snapshot ::::");
    for (Node n : nodes) {
        System.out.println("\nLocal State Recording of " + n.getName() + " ::");
        System.out.println("-->Messages delivered: ");
        HashMap<String, List<String>> sendMessage = n.getSendMessages();
        if (!sendMessage.isEmpty()) {
            for (Map.Entry<String, List<String>> sendMessages : sendMessage.entrySet()) {
                String key = sendMessages.getKey();
                List<String> values = sendMessages.getValue();
                System.out.print("To:  " + key + "  { ");
                for (int i = 0; i < values.size(); i++) {
                    System.out.print(values.get(i));
                    if (i < values.size() - 1) {
                        System.out.print(" , ");
                    }
                }
                System.out.println(" }");
            }
        } else {
            System.out.println("NONE");
        }
        System.out.println("-->Messages received: ");
        HashMap<String, List<String>> receivedMessage = n.getReceivedMessages();
        if (!receivedMessage.isEmpty()) {
            for (Map.Entry<String, List<String>> receivedMessages :
receivedMessage.entrySet()) {
                String key = receivedMessages.getKey();
                List<String> values = receivedMessages.getValue();
                System.out.print("From:  " + key + "  { ");
                for (int i = 0; i < values.size(); i++) {
                    System.out.print(values.get(i));
                    if (i < values.size() - 1) {
                        System.out.print(" , ");
                    }
                }
```

```
                    System.out.println(" }");
                }
            } else {
                System.out.println("NONE");
            }
        }

        // Print communication channel state
        System.out.println("\nCommunication Channel State ::");
        HashMap<String, Queue<String>> communicationChannel =
channel.getCommunicationChannel();
        for (Map.Entry<String, Queue<String>> specificChannel :
communicationChannel.entrySet()) {
            String receiverName = specificChannel.getKey();
            HashMap<String, List<String>> tempHashMap = new HashMap<>();
            Queue<String> tempQueue = specificChannel.getValue();
            for (String message : tempQueue) {
                String[] splitMessage = message.split(":");
                String senderName = splitMessage[0];
                String messageContentType = splitMessage[1];
                if (messageContentType.equalsIgnoreCase("TEXT")) {
                    if (tempHashMap.containsKey(senderName)) {
                        tempHashMap.get(senderName).add(splitMessage[2]);
                    } else {
                        List<String> tempList = new ArrayList<>();
                        tempList.add(splitMessage[2]);
                        tempHashMap.put(senderName, tempList);
                    }
                }
            }
            for (Map.Entry<String, List<String>> tempNode : tempHashMap.entrySet()) {
                String senderName = tempNode.getKey();
                List<String> values = tempNode.getValue();
                System.out.print("\n Channel :: " + senderName + " -- " + receiverName + " :  { ");
                for (int i = 0; i < values.size(); i++) {
                    System.out.print(values.get(i));
                    if (i < values.size() - 1) {
                        System.out.print(" , ");
                    }
                }
                System.out.println(" }");
            }
        }

        System.exit(0); // Exit program after printing snapshot
    } else {
        // If initiator is not valid
        System.out.println("Please choose another initiator!!");
        System.exit(0);
    }
```

```
    }

    // Method to open file dialog and get the selected file path
    public static String openFileDialog() {
        FileDialog fd = new FileDialog((Frame) null, "Open", FileDialog.LOAD);
        fd.setVisible(true); // Display file dialog
        String directory = fd.getDirectory();
        String file = fd.getFile();
        if (directory != null && file != null) {
            return directory + file; // Return full path of the selected file
        }
        return null; // Return null if no file was selected
    }
}
```

2.2.>> Node. java

```
/*ABOUT NODE CLASS
 * The Node class in Java models a system or process within a distributed environment,
 *  designed to communicate with neighboring nodes via a shared Channel.
 *  Each Node instance runs in its own thread (Runnable), implementing
 *  logic to send and receive messages, manage message types (TEXT or MARKER),
 *  and maintain message histories (sendMessages and receivedMessages).
 * Key attributes include its name, neighbor nodes list, initiator status (isInitiator),
 *  and a random number generator for message scheduling. The class supports dynamic
 *  behavior where nodes can act as initiators, exchange messages, and react to incoming
 *  marker messages to record global states in distributed systems scenarios.
 */
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;

public class Node implements Runnable {
    private String name; // Name of the node
    private Channel channel; // Communication channel for the node
```

```
    private List<String> neighbor; // List of neighboring nodes
    private Boolean isInitiator = false; // Flag to check if the node is the initiator
    private Random random = new Random(); // Random number generator for message sending
    private MessageType messageType = MessageType.TEXT; // Type of message (TEXT or MARKER)
    private HashMap<String, List<String>> sendMessages = new HashMap<>(); // Messages sent by this node
    private HashMap<String, List<String>> receivedMessages = new HashMap<>(); // Messages received by this node
    private int markerSendCounter = 3; // Counter to control marker message sending

    // Constructor to initialize the node
    public Node(String name, Channel channel, List<String> neighbor) {
        this.name = name;
        this.channel = channel;
        this.neighbor = neighbor;
    }

    // Main logic of the node, to be executed in a separate thread
    @Override
    public synchronized void run() {
        while (!Thread.currentThread().isInterrupted()) {
            try {
                // If the node is the initiator, choose the type of message to send
                if (isInitiator)
                    messageTypeChooser();

                // Decide whether to send a message based on random chance
                int willSend = random.nextInt(10 + 1);
                if (messageType == MessageType.MARKER || willSend < 8)
```

```
                sendMessage();

            // If the thread is not interrupted, wait for a random period and then receive a
            // message
            if (!Thread.currentThread().isInterrupted()) {
                delay(random.nextInt(500) + 1);
                receivedMessage();
                delay(random.nextInt(500) + 1);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

// Set the initiator status for the node
public void setInitiatorStatus() {
    isInitiator = true;
}

// Get the messages sent by this node
public HashMap<String, List<String>> getSendMessages() {
    return sendMessages;
}

// Get the messages received by this node
public HashMap<String, List<String>> getReceivedMessages() {
    return receivedMessages;
}
```

```
    // Get the name of the node
    public String getName() {
        return name;
    }

    // Method to send a message to a random neighbor
    private void sendMessage() {
        if (messageType == MessageType.MARKER) {
            // If the message type is MARKER, send it to all neighbors
            for (String receiverName : neighbor) {
                String message = messageGenerator(null, receiverName);
                System.out.println("Send Marker :: " + name + " is sending " + " marker " + " to " +
receiverName);
                System.out.println("------------------------------------------------------------");
                channel.deliver(message);
            }
            // Interrupt the thread after sending marker messages
            Thread.currentThread().interrupt();
        } else if (!neighbor.isEmpty()) {
            // If the message type is TEXT, send it to a random neighbor
            int randomIndex = random.nextInt(neighbor.size());
            String receiverName = neighbor.get(randomIndex);
            String messageContent = String.valueOf(random.nextInt(100));
            String message = messageGenerator(messageContent, receiverName);
            // Update the sent messages map
            if (sendMessages != null && sendMessages.keySet() != null) {
                boolean exists = sendMessages.keySet().stream()
                        .anyMatch(k -> k.equalsIgnoreCase(receiverName));
                if (exists) {
```

```
                List<String> temp = sendMessages.get(receiverName);
                if (temp != null) {
                    temp.add(messageContent);
                } else {
                    temp = new LinkedList<>();
                    temp.add(messageContent);
                }
                sendMessages.put(receiverName, temp);
            } else {
                List<String> temp = new LinkedList<>();
                temp.add(messageContent);
                sendMessages.put(receiverName, temp);
            }
        } else {
            List<String> temp = new LinkedList<>();
            temp.add(messageContent);
            sendMessages.put(receiverName, temp);
        }
        System.out.println("Send Message :: " + name + " is sending " + messageContent + " to
" + receiverName);
        channel.deliver(message);
    }
  }

  // Method to receive a message from the channel
  private void receivedMessage() {
    String message = channel.receive(name);
    if (message != null) {
        String[] messageContent = message.trim().split(":");
```

```
            // If the received message is a MARKER, update the message type and send marker
            // messages
            if (messageContent[1].equals("MARKER")) {
                System.out.println(
                        "Received Marker :: " + name + " is received a marker from " + " from " +
messageContent[0]);
                messageType = MessageType.MARKER;
                sendMessage();
            } else {
                // If the received message is a TEXT, update the received messages map
                if (receivedMessages != null && receivedMessages.keySet() != null) {
                    boolean exists = receivedMessages.keySet().stream()
                            .anyMatch(k -> k.equalsIgnoreCase(messageContent[0]));
                    System.out.println("Received Message :: " + name + " received a message " +
messageContent[2]
                            + " from " + messageContent[0]);
                    if (exists) {
                        List<String> temp = receivedMessages.get(messageContent[0]);
                        if (temp != null) {
                            temp.add(messageContent[2]);
                        } else {
                            temp = new LinkedList<>();
                            temp.add(messageContent[2]);
                        }
                        receivedMessages.put(messageContent[0], temp);
                    } else {
                        List<String> temp = new LinkedList<>();
                        temp.add(messageContent[2]);
                        receivedMessages.put(messageContent[0], temp);
                    }
```

```
            } else {
                List<String> temp = new LinkedList<>();
                temp.add(messageContent[2]);
                receivedMessages.put(messageContent[0], temp);
            }
        }
    }
}

// Method to choose the type of message to send (TEXT or MARKER)
private void messageTypeChooser() {
    if (markerSendCounter <= 0) {
        int chooser = random.nextInt(10);
        if (chooser < 8)
            messageType = MessageType.TEXT;
        else
            messageType = MessageType.MARKER;
    } else {
        markerSendCounter--;
    }
}

// Method to generate a message string with the given content and receiver name
private String messageGenerator(String messageContent, String receiverName) {
    String message;
    message = name + ":" + messageType + ":" + messageContent + ":" + receiverName;
    return message;
}
```

```
    // Method to introduce a delay (in milliseconds) in the thread execution
    private void delay(int miliSec) throws InterruptedException {
        if (!Thread.currentThread().isInterrupted())
            Thread.sleep(miliSec);
    }
}
```

2.3.>> Graph. java

```
/*ABOUT GRAPH CLASS
The Graph class in Java encapsulates functionality for managing a directed graph
using an adjacency list data structure. It initializes with a specified file path
to populate its adjacency list, reading node connections and handling cases
where nodes have no neighbors or are connected to a null node. Methods include
adding edges between nodes, printing the graph's adjacency list representation,
and performing a Breadth-First Search (BFS) to check graph connectivity from a
specified initiator node. It offers utility for applications needing to analyze
and manipulate graph structures, particularly in scenarios involving distributed
systems or network topologies where connectivity and node relationships are crucial
for system behavior and analysis.
 */
import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Scanner;

public class Graph {
    private String filePath; // Path to the file containing the graph data
    private Map<String, List<String>> adjacencyList = new HashMap<>(); // Adjacency list to
store the graph

    // Constructor to initialize the graph with the file path
    public Graph(String filePath) {
        this.filePath = filePath;
        populateAdjacencyList(); // Populate the adjacency list with data from the file
    }

    // Method to read the file and populate the adjacency list
    private void populateAdjacencyList() {
        try {
            File file = new File(filePath); // Open the file
            Scanner scanner = new Scanner(file); // Scanner to read the file
```

```
            // Read the file line by line
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine().trim(); // Read and trim each line
                if (line.isEmpty()) {
                    continue; // Skip empty lines
                }

                String[] parts = line.split("->"); // Split the line into source and destination nodes
                String src = parts[0].trim(); // Source node

                // If there are destination nodes, add them to the adjacency list
                if (parts.length > 1) {
                    for (int i = 1; i < parts.length; i++) {
                        String dest = parts[i].trim(); // Destination node
                        if (dest.equalsIgnoreCase("null"))
                            addEdge(src, null); // Add edge with null destination
                        else
                            addEdge(src, dest); // Add edge with valid destination
                    }
                } else {
                    addEdge(src, null); // Handle case with only source node
                }
            }

            scanner.close(); // Close the scanner
            printGraph(); // Print the adjacency list representation of the graph
        } catch (Exception e) {
            System.out.println("Error reading file: " + e.getMessage()); // Handle file reading errors
        }
    }

    // Method to add an edge to the adjacency list
    private void addEdge(String src, String dest) {
        adjacencyList.putIfAbsent(src, new ArrayList<>()); // Add source node if not already
present
        if (dest != null) {
            adjacencyList.get(src).add(dest); // Add destination node to the adjacency list of the
source node
        }
    }

    // Method to print the adjacency list representation of the graph
    private void printGraph() {
        System.out.println();
        System.out.println("Graph Information :::: ");
        System.out.println();
        System.out.println("Inserted Graph (Adjacency List Representation) :");
        for (Map.Entry<String, List<String>> entry : adjacencyList.entrySet()) {
            System.out.print(entry.getKey() + " -> "); // Print source node
            List<String> neighbors = entry.getValue(); // Get list of neighbors
```

```
            if (!neighbors.isEmpty()) {
                for (int i = 0; i < neighbors.size(); i++) {
                    System.out.print(
                            neighbors.get(i) != null ? (neighbors.get(i).equals("null") ? "null" :
neighbors.get(i)): "NULL"); // Print each neighbor
                    if (i != neighbors.size() - 1) {
                        System.out.print(", "); // Print comma if not the last neighbor
                    }
                }
            } else {
                System.out.print("null"); // Print null if no neighbors
            }
            System.out.println();
        }
    }

    // Method to perform BFS and check if the graph is connected from the initiator
    // node
    private Boolean bfsForAdjacencyList(String initiator) {
        int noOfNodes = adjacencyList.size(); // Number of nodes in the graph
        boolean[] visited = new boolean[noOfNodes]; // Array to keep track of visited nodes
        Arrays.fill(visited, false); // Initialize all nodes as not visited

        List<String> q = new ArrayList<>();
        q.add(initiator); // Add initiator to the queue
        visited[getIndex(initiator)] = true; // Mark initiator as visited

        // Perform BFS
        while (!q.isEmpty()) {
            String current = q.remove(0); // Dequeue a node
            List<String> neighbors = adjacencyList.get(current); // Get neighbors of the current
node

            if (neighbors != null) {
                for (String neighbor : neighbors) {
                    if (!visited[getIndex(neighbor)]) {
                        q.add(neighbor); // Add unvisited neighbors to the queue
                        visited[getIndex(neighbor)] = true; // Mark neighbor as visited
                    }
                }
            }
        }

        // Check if all nodes were visited
        for (int i = 0; i < noOfNodes; i++) {
            if (!visited[i]) {
                return false; // If any node is not visited, return false
            }
        }
        return true; // Return true if all nodes were visited
```

```
    }

    // Method to get the index of a vertex in the adjacency list
    private int getIndex(String vertex) {
        int index = 0;
        for (String key : adjacencyList.keySet()) {
            if (key.equals(vertex)) {
                return index; // Return the index if vertex matches
            }
            index++;
        }
        return index; // Return the index
    }

    // Method to check if the initiator can be a valid starting node for BFS
    public Map<String, List<String>> checkInitiator(String initiator) {
        boolean exists = adjacencyList.keySet().stream().anyMatch(k ->
k.equalsIgnoreCase(initiator)); // Check if initiator exists in the graph
        if (exists) {
            if (bfsForAdjacencyList(initiator)) {
                System.out.println(initiator + " Can be an initiator."); // Print if initiator is valid
                return adjacencyList; // Return adjacency list
            } else {
                System.out.println(initiator + " Cannot be an initiator."); // Print if initiator is not valid
                return null;
            }
        } else {
            System.out.println("Node does not exist in the adjacency list. Please check"); // Print if
initiator does not exist
            return null;
        }
    }
}
```

2.4.>> Channel. java

```
/*ABOUT CHANNEL CLASS
 * The Channel class in Java represents a communication channel between
 *  nodes in a distributed system. It uses a HashMap to manage communication
 *  queues (Queue<String>) for each receiver node. Messages can be delivered
 *  (deliver) to a specific receiver's queue based on the message content,
 *  and messages can be received (receive) from a particular receiver's queue.
 *  The class ensures that messages are queued appropriately and retrieved in
 *  a FIFO manner. Additionally, it provides a method (getCommunicationChannel)
 *  to access the entire communication channel map, useful for debugging or logging
 *  purposes in distributed system simulations or implementations. This class
 *  facilitates asynchronous communication between nodes, essential for exchanging
 *  messages and maintaining synchronization in distributed environments.
 */
import java.util.HashMap;
```

```
import java.util.LinkedList;
import java.util.Queue;

public class Channel {
   // HashMap to store communication channels for each receiver
   private HashMap<String, Queue<String>> communicationChannel = new HashMap<>();

   // Method to deliver a message to the appropriate receiver's queue
   public void deliver(String message) {
      int lastIndex = message.lastIndexOf(":"); // Find the last occurrence of ':' in the message
      String receiverName = message.substring(lastIndex + 1); // Extract receiver's name from
the message

      // Check if the communication channel and its keys are not null
      if (communicationChannel != null && communicationChannel.keySet() != null) {
         // Check if the receiver's name exists in the communication channel
         boolean exists = communicationChannel.keySet().stream()
               .anyMatch(k -> k.equalsIgnoreCase(receiverName));
         if (exists) {
            // If the receiver's queue exists, add the message to it
            Queue<String> temp = communicationChannel.get(receiverName);
            if (temp != null) {
               temp.offer(message); // Add the message to the queue
            } else {
               // If the queue is null, create a new queue and add the message to it
               temp = new LinkedList<>();
               temp.offer(message);
            }
            // Update the communication channel with the receiver's queue
            communicationChannel.put(receiverName, temp);
         } else {
            // If the receiver's queue does not exist, create a new queue and add the
            // message to it
            Queue<String> temp = new LinkedList<>();
            temp.offer(message);
            // Add the new queue to the communication channel for the receiver
            communicationChannel.put(receiverName, temp);
         }
      } else {
         // If the communication channel is null, create a new queue and add the message
         // to it
         Queue<String> temp = new LinkedList<>();
         temp.offer(message);
         // Add the new queue to the communication channel for the receiver
         communicationChannel.put(receiverName, temp);
      }
   }

   // Method to receive a message from the appropriate receiver's queue
   public String receive(String receiverName) {
```

```
        // Check if the receiver's name exists in the communication channel
        boolean exists = communicationChannel.keySet().stream()
                .anyMatch(k -> k.equalsIgnoreCase(receiverName));
        if (exists) {
            // Get the queue for the receiver
            Queue<String> temp = communicationChannel.get(receiverName);
            // If the queue is not null and not empty, poll a message from it
            if (temp != null && !temp.isEmpty()) {
                String message = temp.poll(); // Retrieve and remove the head of the queue
                return message; // Return the message
            }
        }
        return null; // Return null if no message is available
    }

    // Method to get the communication channel (for debugging or logging purposes)
    public HashMap<String, Queue<String>> getCommunicationChannel() {
        return communicationChannel;
    }
}
```

2.5.>> MessageType. java

```
// Enum representing different types of messages
public enum MessageType {
    // Represents a text message
    TEXT,
    // Represents a marker message used in distributed algorithms
    MARKER
}
```

3. Pre-requisites

3.1. Java Development Kit (JDK): Ensure JDK is installed and properly configured on your system. This code appears to be written in Java, so JDK is essential for compiling and running Java programs.

3.2. Input File Format: The input file format represents an adjacency list of a directed graph, where each line describes a node and its outgoing edges using the syntax `source->destination1->destination2....` For example, in the lines `A->C->E`, `B->A`, `C->B`, `D->A`, and `E->D`, node `A` has directed edges to nodes `C` and `E`, node `B` has a directed edge to node `A`, and so on. This format allows for easy parsing and construction of a graph's adjacency list, which can be used for various graph

operations like BFS for connectivity checks. In the provided `Graph` class, the `populateAdjacencyList()` method reads the file, parses each line, and constructs the adjacency list, facilitating graph algorithms and operations.

Examples of this provided in Sample Input file folder.

3.3. **Code run**: Use your IDE or command-line tools (like javac for compilation and java for execution) to compile and run the Main class. Ensure all dependencies are resolved and paths (for input files) are correctly specified.

Steps to run without any IDE:

Open terminal in Code folder. Then run these commands sequentially,

```
Javac -d bin src/*.java

java -cp bin Main
```

The program uses System.out.println() extensively for logging and displaying messages related to node processing, thread actions, and system status. Ensure the console output is clear and visible during program execution.

3.4. **Abstract Window Toolkit**: The openFileDialog() method uses FileDialog to prompt the user to select an input file. Ensure that the AWT components (FileDialog and Frame) work correctly on your system.

3.5. **Node Behavior**: Each Node in the program simulates a process in a distributed system. Understand how each Node transitions through different states (Status) based on random events (setStatus() method) and interactions with other nodes (requestToken() and sendToken() methods).

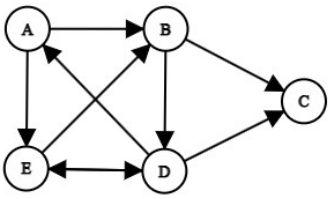
3.6. **Inserting Initiator Name:** To insert the initiator name in the program, ensure you have a correctly formatted input file representing the graph's adjacency list, where each line is in the format `source->destination1->destination2....` At the start

of the program, use the file dialog to select this graph file, which will be read and parsed to create the graph's adjacency list. When prompted, enter the name of the initiator node, ensuring it matches one of the nodes defined in the graph file. The program will validate the initiator node by checking if it exists in the graph and whether it can reach all other nodes via BFS, ensuring the initiator is a valid starting point for the Chandy-Lamport global state recording algorithm. If the initiator is valid, the program will proceed to simulate the distributed system, starting with the initiator node and performing the global state recording.

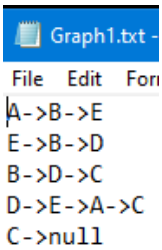
4. Results

4.1. Result 1

4.1.1. Input Graph



4.1.2. Input File

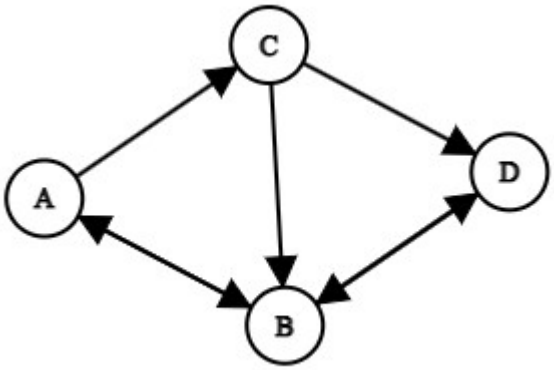


4.1.3. Output

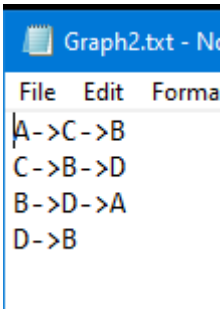
```
Information Message :: B is now initiator
Send Message :: B is sending 37 to C
Send Message :: A is sending 57 to B
Send Message :: D is sending 71 to A
Send Message :: D is sending 1 to C
Send Message :: D is sending 77 to A
Send Message :: B is sending 16 to C
Received Message :: A received a message 71 from D
Received Message :: C received a message 37 from B
Received Message :: B received a message 57 from A
Send Message :: B is sending 40 to C
Send Message :: D is sending 86 to C
Send Message :: A is sending 27 to B
Received Message :: B received a message 27 from A
Received Message :: C received a message 1 from D
Send Message :: E is sending 68 to B
Received Message :: A received a message 77 from D
Received Message :: C received a message 16 from B
Send Message :: A is sending 63 to B
Received Message :: C received a message 40 from B
Received Message :: B received a message 68 from E
Send Message :: D is sending 27 to E
Send Message :: A is sending 42 to E
Send Marker :: B is sending  marker  to D
------------------------------------------------------------
Send Marker :: B is sending  marker  to C
------------------------------------------------------------
```

4.2. Result 2

4.2.1. Input Graph



4.2.2. Input File

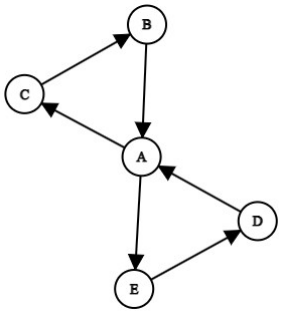


4.2.3. Output

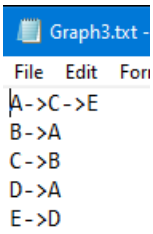
```
System Log ::::

Information Message :: D is now initiator
Send Message :: D is sending 17 to B
Send Message :: A is sending 97 to B
Send Message :: B is sending 96 to D
Received Message :: B received a message 17 from D
Send Message :: B is sending 69 to A
Send Message :: C is sending 2 to D
Received Message :: D received a message 96 from B
Received Message :: A received a message 69 from B
Send Message :: C is sending 93 to B
Received Message :: D received a message 2 from C
Received Message :: B received a message 97 from A
Send Message :: A is sending 94 to B
Send Message :: D is sending 21 to B
Send Message :: C is sending 9 to B
Send Message :: A is sending 24 to C
Received Message :: C received a message 24 from A
Send Message :: B is sending 72 to D
Received Message :: D received a message 72 from B
Send Message :: C is sending 24 to B
Received Message :: B received a message 93 from C
Send Message :: D is sending 5 to B
Send Message :: A is sending 47 to B
Received Message :: B received a message 94 from A
Send Message :: C is sending 16 to D
Received Message :: B received a message 21 from D
```

4.3.Result 3

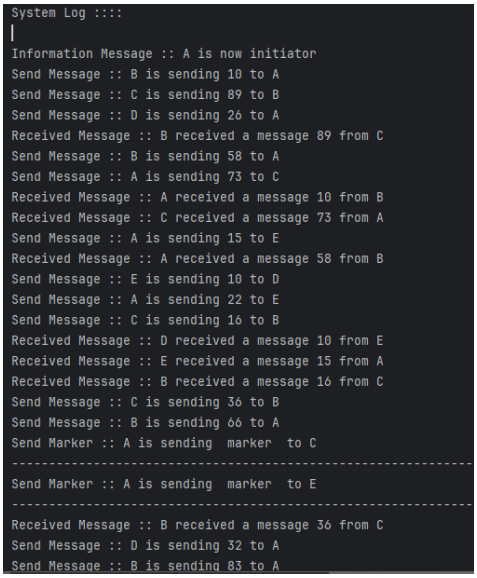
4.3.1. Input Graph



4.3.2. Input File



4.3.3. Output



** These screenshots are part of the output log files; the complete logs are available in the output folder.

5. Remarks

5.1. Simulation of Distributed System: The program simulates a distributed system using threads where each node is represented as a separate thread.

5.2. Communication Management: A shared Channel class manages communication between nodes.

5.3. Graph Structure: The network of nodes is defined by an adjacency list in an input file, which is crucial for the simulation.

5.4. Input File Formatting: Correct formatting of the input file is essential for the program's execution.

5.5. Initiator Selection: Selecting a valid initiator node is necessary to ensure the algorithm functions correctly, as the initiator must be capable of reaching all other nodes in the graph.

5.6. Java AWT Usage: The program uses Java AWT for file selection and input dialogs.

5.7. Message Passing and State Recording: The program handles both message passing and global state recording effectively.

5.8. Adherence to Pre-requisites: Ensuring proper input and adhering to defined pre-requisites are critical for demonstrating an effective simulation of the Chandy-Lamport algorithm.