Assignment - 4

1. Problem Statement

1.1. Raymond's Algorithm -- Diffusion-Computation approach

2. Source Code

2.1. >> Main.java

2.2. >> Node.java

2.3. >> Status.java

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:** This modified indentation presentation of a tree structure provides a clear and intuitive way to represent hierarchical relationships in a textual format. It's versatile enough for various applications where tree-like structures need to be managed, parsed, or displayed. Understanding and correctly interpreting this format is essential for building robust applications that work with hierarchical data effectively. The structure uses indentation to denote parent-child relationships, where each line represents a node in the tree. The indentation level indicates the depth of the node in the tree hierarchy:

- Nodes are aligned to the left margin and represent the leaf nodes of the tree.
- Subsequent Depths: Nodes are indented to the right by a consistent number (4) of spaces relative to their child nodes.

  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:

1. Open terminal in `Code` folder. Then run these commands sequentially,
2. `Javac -d bin src/*.java`
3. `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. System Resources: Ensure your system has sufficient resources (CPU, memory) to handle potentially intensive thread operations, especially if dealing with a large number of nodes or complex tree structures.

3.7. Understanding of Java I/O and Multithreading: Familiarity with Java I/O operations (`FileReader`, `BufferedReader`) and multithreading (`Thread class`, `Runnable interface`) is crucial, as the code involves file reading, concurrent processing using threads, and synchronization.

3.8. Java Enum Usage: Understanding how Java enums (`Status` enum) work and how they are utilized for defining states (`PHOLD`, `REQUESTING`, `NONE`, `ABORT`) in the Node class.

4. Results

4.1. Result 1

4.1.1. Input Tree Diagram

4.1.2. Input File

4.1.3. Output File

4.1.4. Output Tree Diagram

4.2. Result 2

4.2.1. Input Tree Diagram

4.2.2. Input File

4.2.3. Output File

4.2.4. Output Tree Diagram

4.3. Result 3

4.3.1. Input Tree Diagram

4.3.2. Input File

4.3.3. Output File

4.3.4. Output Tree Diagram

5. Remarks

5.1. Structure and Functionality: The code implements a hierarchical tree structure using Node objects to represent nodes with parent-child relationships. It

reads input from a file, builds the tree structure using indentation levels, and manages node relationships using stacks and maps (`HashMap` and `TreeMap`).

5.2. Multithreading and Concurrency: Multithreading is utilized to simulate concurrent processes (Node instances) in a distributed system. Each Node runs as a separate thread, interacting with its parent and potentially other nodes through token requests (`requestToken()` and `sendToken()` methods).

5.3. Error Handling and Logging: The code includes basic error handling for file operations (`IOException`) and thread interruptions (`InterruptedException`). Extensive logging (`System.out.println()`) is used for debugging and providing information about node states, thread actions, and system messages.

5.4. User Interaction and Input Handling: AWT (FileDialog) is used for user interaction to select an input file, ensuring flexibility in choosing the hierarchical structure to simulate.

5.5. State Management with Enums: The Node class utilizes a Status enum (`PHOLD, REQUESTING, NONE, ABORT`) to manage node states, influencing node behavior during thread execution.

5.6. Optimization and Performance Considerations: The code could benefit from optimizations in terms of memory usage and thread management, especially for large tree structures or high numbers of concurrent nodes. Performance considerations include ensuring efficient traversal and synchronization among threads to avoid potential race conditions or deadlocks.

5.7. User Instructions: Before running the code, ensure the JDK is installed, and an appropriate IDE or text editor is set up for Java development. Prepare an input file formatted with nodes and indentation levels to simulate different hierarchical

structures. Monitor the console output for detailed system messages, including node creation, thread actions, and final tree structure.

5.8. Abort Functionality in Node.java: In the Node.java class, the ABORT functionality manages node termination based on specific criteria, utilizing the Status `enum` for state management. Nodes transition to the ABORT state when conditions such as prolonged inactivity (tracked by `abortCounter`) or system-wide failures (monitored via `rootAbortCounter`) occur. This feature is crucial for simulating realistic distributed system behaviors, where nodes may fail due to deadlock, resource exhaustion, or other critical issues. Upon entering the ABORT state, nodes cease operations, potentially terminating their threads and logging relevant termination reasons. This mechanism not only enhances the simulation's accuracy but also enables the exploration of recovery strategies and the impact of node failures on the overall distributed system dynamics.