Distributed Deadlock Detection and Recovery System

1. Introduction

In distributed computing systems, resources such as files, databases, or devices are often shared among multiple processes located on different nodes. When multiple processes request resources in a conflicting order, a situation known as deadlock can occur. Deadlock prevents some or all processes from proceeding, thereby halting the system's progress.

To ensure system reliability and consistency, Distributed Deadlock Detection and Recovery (DDDR) mechanisms are used. This project demonstrates a simple simulation of a distributed deadlock detection and recovery algorithm using Python.

2. Objective

The primary goal of this project is to:

Simulate a distributed environment where processes on different nodes compete for resources.

Detect the occurrence of a deadlock using a wait-for graph (WFG).

Recover from the deadlock by aborting one or more processes involved in the cycle.

3. Methodology

a) System Model

Each node in the distributed system hosts multiple processes. A wait-for graph represents which process is waiting for another process's resource.

Node: Represents a machine or site in the distributed system.

Process: Represents a task or program running on a node.

Edge (A → B): Process A is waiting for process B to release a resource.

b) Deadlock Detection

The system checks for cycles in the wait-for graph using Depth-First Search (DFS).
If a cycle exists, it indicates a deadlock condition.
This approach is simple, efficient, and suitable for small systems or educational demonstrations.

c) Recovery Mechanism

Once a deadlock is detected:

1. One process (called the victim) is selected for termination.

2. The victim process releases all resources it holds.

3. The wait-for graph is updated, and the detection algorithm is run again to confirm that the system is deadlock-free.
In this project, the lowest process name is chosen as the victim for simplicity.

4. Implementation

The project is implemented in Python using basic data structures.
A dictionary (wait_for_graph) represents the system's state. Each key (process) maps to a list of processes it is waiting for.

Algorithm Steps:

1. Input the wait-for graph.

2. Perform DFS to detect any cycle.

3. If a cycle is found, print a deadlock detection message.

4. Abort the selected victim process.

5. Remove all edges related to the victim from the graph.

6. Repeat the detection to verify that the system is now free from deadlock.

The output shows that the system initially had a circular wait among three processes. After aborting one process, the system became deadlock-free.

5. Advantages

- Simple and easy-to-understand algorithm.
- No special libraries required — runs on any Python installation.
- Demonstrates key distributed system concepts (deadlock, detection, recovery).

6. Applications

- Distributed databases and transaction systems.
- Multi-node resource allocation in clusters or cloud environments.
- Operating system kernel design for distributed resource management.

7. Conclusion

This project successfully simulates a Distributed Deadlock Detection and Recovery System using a simplified Python model.
It demonstrates how a deadlock occurs in a distributed wait-for graph, how detection algorithms identify cycles, and how recovery can restore system consistency by aborting a process.

The experiment reinforces the importance of deadlock handling in distributed systems to ensure reliability, responsiveness, and fault tolerance.