# PREMIER UNIVERSITY CHATTOGRAM

DEPARTMENT OF **COMPUTER SCIENCE AND ENGINEERING**

# COMPLEX ENGINEERING PROBLEM

| | |
|---|---|
| *COURSE NAME* | Operating Systems |
| *COURSE CODE* | CSE 3733 |
| *ASSIGNMENT TOPIC* | **Design and Implementation of a Deadlock Detection and Recovery System in a Distributed Operating System.** |
| *DATE OF ASSIGNMENT* | 07th February 2025 |

**SUBMITTED TO**


**SHATABDI ROY MOON**

*LECTURER*

*DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING*


| **REMARKS** | **SUBMITTED BY** | |
|---|---|---|
| | **NAME** | Rimjhim Dey |
| | **ID** | 0222220005101039 |
| | **SEMESTER** | 5th |
| | **BATCH** | 42nd |
| | **SESSION** | Fall 2024 |
| | **SECTION** | A |

## *STEP 1-*

### Introduction to the Problem:

In a distributed file system with multiple servers and clients, deadlocks occur when clients request resources (files) in conflicting orders, causing circular dependencies. A deadlock prevents any of the involved clients from completing their operations. Thus, detecting and resolving deadlocks is crucial to ensure smooth system operation and fair access to resources.

## *STEP 2-*

### Problem Analysis:

The key to solving deadlocks is analyzing the system's resource allocation patterns. Deadlocks typically occur in the following conditions:

- **Circular Wait:** A set of processes (clients) are each waiting for a resource (file) held by another client in the set.
- **Hold and Wait:** Clients hold at least one resource and are waiting to acquire additional resources.
- **No Preemption:** Resources cannot be forcibly taken away from clients holding them.
- **Mutual Exclusion:** Resources are shared and cannot be used by more than one client at a time.

To detect deadlocks, we need to monitor these conditions and identify when they form a circular dependency in the system.

## *STEP 3-*

### Deadlock Detection Design:

We can use a **Resource Allocation Graph (RAG)** to represent the system's resource allocation state. Here's how the design works:

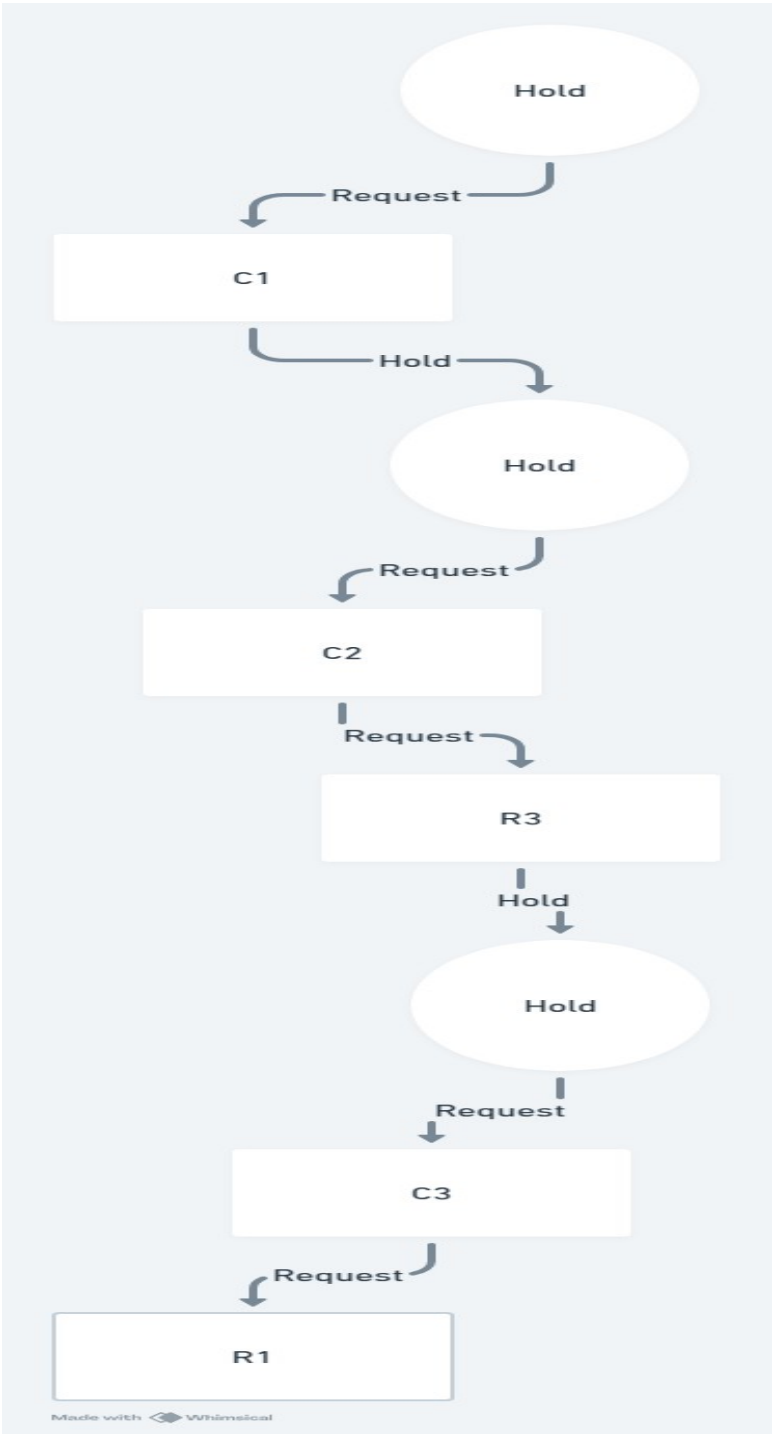
- **Nodes in RAG**:
  - **Client nodes**: Represent the clients (processes).
  - **Resource nodes**: Represent the resources (files).
- **Edges in RAG**:
  - A **request edge** from a client to a resource indicates that the client is requesting the resource.
  - A **hold edge** from a client to a resource indicates that the client is holding the resource.

## Diagram Representation

Let's take the following example for the RAG:

- **Clients**: `C1`, `C2`, `C3` (representing 3 clients)
- **Resources**: `R1`, `R2`, `R3` (representing 3 resources or files)

## RAG Diagram for Deadlock Detection Design:



### Explanation:

1. **Client C1** holds `R1` and requests `R2` (represented by the edges from `C1` to `R2`).
2. **Client C2** holds `R2` and requests `R3` (represented by the edges from `C2` to `R3`).
3. **Client C3** holds `R3` and requests `R1` (represented by the edges from `C3` to `R1`).

### Deadlock Detection:

- We can observe that there is a **cycle** in this graph:
  - `C1 → R2 → C2 → R3 → C3 → R1 → C1`.
- This cycle indicates a **deadlock** because each client is waiting for a resource that is held by another client in the cycle.

## *STEP 4-*

### Deadlock Detection Algorithm:

We will periodically check the RAG for cycles. If a cycle is detected, it indicates a deadlock.

**Cycle Detection:** The most efficient method to detect cycles is using Depth-First Search (DFS). During the DFS traversal, if we encounter a node that is already in the current path of traversal, we have found a cycle.

**Periodicity:** This process will run periodically to check if any deadlocks have occurred since the last check.

### Pseudo Code for Deadlock Detection:

```
// Define the graph (RAG)
1.Graph G = new Graph()

2.// Function to check for deadlock by detecting cycles using DFS
3.function DetectDeadlock():

4.// Step 1: Initialize visited, recursion stack arrays
5.visited = new Array(G.nodes) // Array to track visited nodes

6.inRecursionStack = new Array(G.nodes) // Array to track nodes in current DFS path

7.// Step 2: Loop over all client nodes (i.e., processes)
8.for each node in G.nodes:
```

```
9.if visited[node] is false:
10.if DFS(node, visited, inRecursionStack):
11.return "Deadlock Detected" // Cycle detected, deadlock exists
12.return "No Deadlock" // No cycle detected

13.// Function to perform DFS traversal and check for cycle
14.function DFS(node, visited, inRecursionStack):

15.// Step 3: Mark the current node as visited and add to recursion stack
16.visited[node] = true
17.inRecursionStack[node] = true

18.// Step 4: Explore all neighbors (resources and other clients)
19.for each neighbor in G.neighbors(node):

20.// If the neighbor is not visited, perform DFS on it
21.if visited[neighbor] is false:
22.if DFS(neighbor, visited, inRecursionStack):
23.return true // If a cycle is found, return true

24.// Step 5: If the neighbor is in the current recursion stack, cycle detected
25.else if inRecursionStack[neighbor] is true:
26.return true // Cycle detected

27.// Step 6: Remove the node from the recursion stack (backtracking)
28.inRecursionStack[node] = false
29.return false // No cycle found in the current path

30.// Main function to periodically detect deadlock
31.function PeriodicDeadlockDetection():
32.while true: // Loop to check periodically
33.result = DetectDeadlock()
34.if result == "Deadlock Detected":
35.print("Deadlock detected in the system")

36.// Step 7: Handle deadlock recovery (Preemption, Rollback, etc.)
37.HandleDeadlockRecovery()
38.else:
39.print("No deadlock detected")

40.wait(10 seconds) // Wait before the next check

41.// Function to handle deadlock recovery
42.function HandleDeadlockRecovery():
```

```
43.// Implement preemption or rollback strategies here
44.// Example: Preempt a resource or rollback a process
45.print("Recovery strategies applied")

46.// Start periodic deadlock detection
47.PeriodicDeadlockDetection()
```

## Explanation of the Deadlock Detection Algorithm:

1. **Graph Representation (RAG)**:
    - The **Resource Allocation Graph (RAG)** consists of **client nodes** (representing clients or processes) and **resource nodes** (representing files or other resources).
    - **Request edges** point from clients to resources (clients requesting resources).
    - **Hold edges** point from resources to clients (clients holding resources).

2. **Cycle Detection (DFS)**:
    - The DFS function explores the graph to detect cycles. In the context of deadlock detection, if DFS revisits a node that is part of the current recursion stack, it indicates that a cycle has been found, meaning a deadlock exists.
    - Two arrays are used:
        - **visited[]**: Marks whether a node has been fully explored.
        - **inRecursionStack[]**: Tracks nodes that are currently being explored in the DFS path.
    - If a cycle is detected, the system concludes that a deadlock has occurred.

3. **Periodicity**:
    - The **PeriodicDeadlockDetection** function periodically checks for deadlocks by calling `DetectDeadlock()`.
    - If a deadlock is detected, appropriate **recovery strategies** are applied, such as **preemption** (forcefully removing resources from one of the clients) or **rollback** (rolling back the actions of a client).
    - The system waits for a fixed period (e.g., 10 seconds) before checking again.

4. **Deadlock Recovery**:
    - After detecting a deadlock, the **HandleDeadlockRecovery** function can apply recovery strategies like **preemption** or **rollback** to break the deadlock and allow the system to proceed.

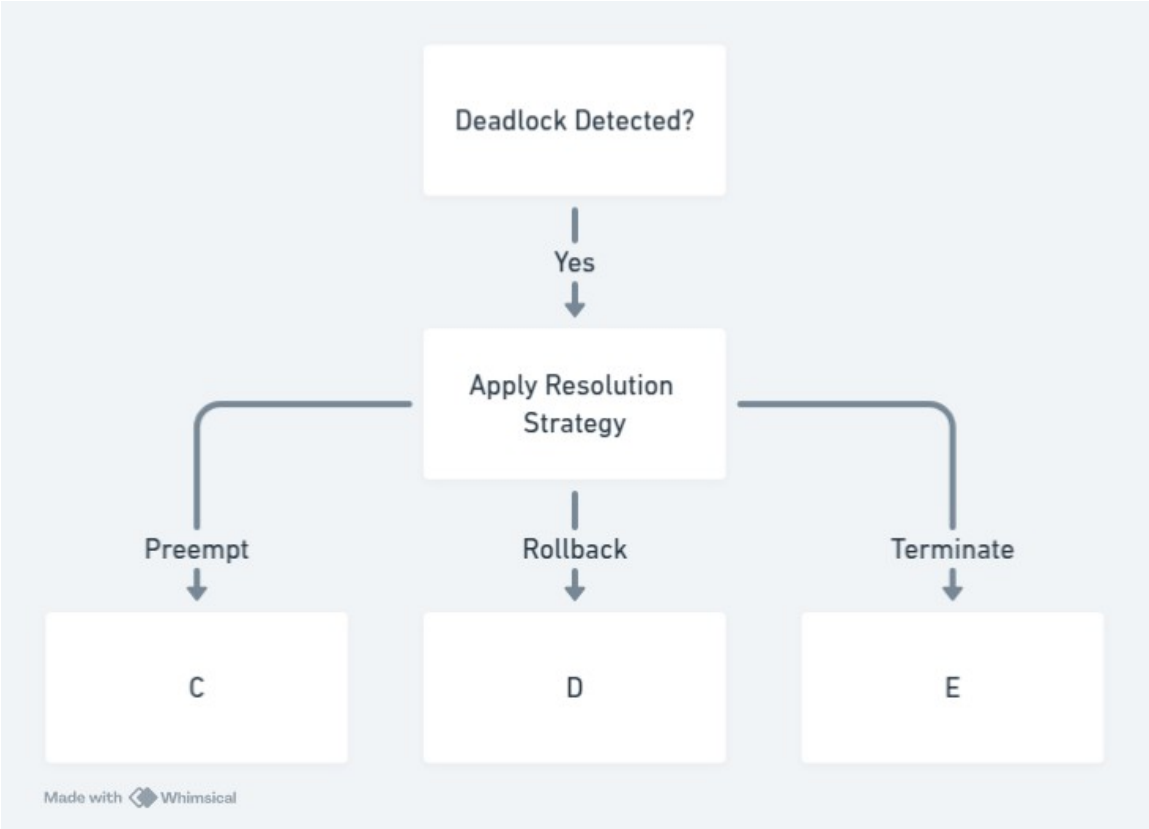
## *STEP 5-*

### Deadlock Resolution Strategy:

Once a deadlock is detected, we need to employ a strategy to break it. We will use the following strategies:

1. **Preemption**:
    - Forcefully take resources away from clients to break the cycle. For instance, if Client A is holding File 1 and waiting for File 2, we preempt File 1 and give it to Client B to resolve the cycle.
    - **Fairness**: Preemption should be fair, meaning no client is starved indefinitely.
2. **Rollback**:
    - Revert one or more clients to a previous state before they requested resources. This can be used to undo actions that led to the deadlock.
3. **Process Termination**:
    - Kill one or more clients involved in the deadlock, releasing all the resources they hold.

The system can choose between these methods based on the situation.



## Deadlock Resolution Algorithm:

**Step 1:** When a deadlock is detected, retrieve the set of clients (processes) involved in the cycle.

**Step 2:** Apply one of the following resolution strategies:

- **Preemption**: Select a client holding a resource, preempt the resource, and assign it to a waiting client.
- **Rollback**: Choose a client, roll it back to a previous state, and release the resources it holds.
- **Process Termination**: Select one or more clients and terminate them to free up resources.

**Step 3:** Update the Resource Allocation Graph (RAG) to reflect the changes.

**Step 4:** Recheck for deadlock. If still detected, repeat the resolution process until the cycle is broken.

## Pseudo Code for Deadlock Resolution:

```
function ResolveDeadlock(deadlockedClients):

print("Deadlock detected among clients:", deadlockedClients)

// Step 1: Select a resolution strategy based on system policy
strategy = ChooseResolutionStrategy()

// Step 2: Apply the selected strategy
if strategy == "Preemption":
client, resource = SelectClientAndResourceForPreemption(deadlockedClients)
PreemptResource(client, resource)
print("Resource", resource, "preempted from Client", client)
else if strategy == "Rollback":
client = SelectClientForRollback(deadlockedClients)
RollbackClient(client)
print("Client", client, "rolled back to previous state")
else if strategy == "Terminate":
client = SelectClientForTermination(deadlockedClients)
TerminateClient(client)
print("Client", client, "terminated")

// Step 3: Update Resource Allocation Graph
UpdateRAG()
```

```
// Step 4: Check if deadlock is resolved
if DetectDeadlock() == "Deadlock Detected":
print("Deadlock persists. Resolving again...")
ResolveDeadlock(deadlockedClients)
else:
print("Deadlock resolved successfully")
function ChooseResolutionStrategy():

// Policy-based selection: Could be based on priority, fairness, etc.
// For simplicity, we start with Preemption, then Rollback, then Terminate

if SystemPolicyAllows("Preemption"):
return "Preemption"
else if SystemPolicyAllows("Rollback"):
return "Rollback"
else:
return "Terminate"
function PreemptResource(client, resource):

// Forcefully reclaim resource from client
ReleaseResourceFromClient(client, resource)
function RollbackClient(client):

// Revert client to a previous safe state and release its resources
RestoreClientToPreviousState(client)
ReleaseAllResources(client)
function TerminateClient(client):

// Kill the client and release all resources it holds
KillProcess(client)
ReleaseAllResources(client)
```

## Explanation of Deadlock Resolution Algorithm:

1. **Trigger**: Once deadlock is detected, the resolution process starts.

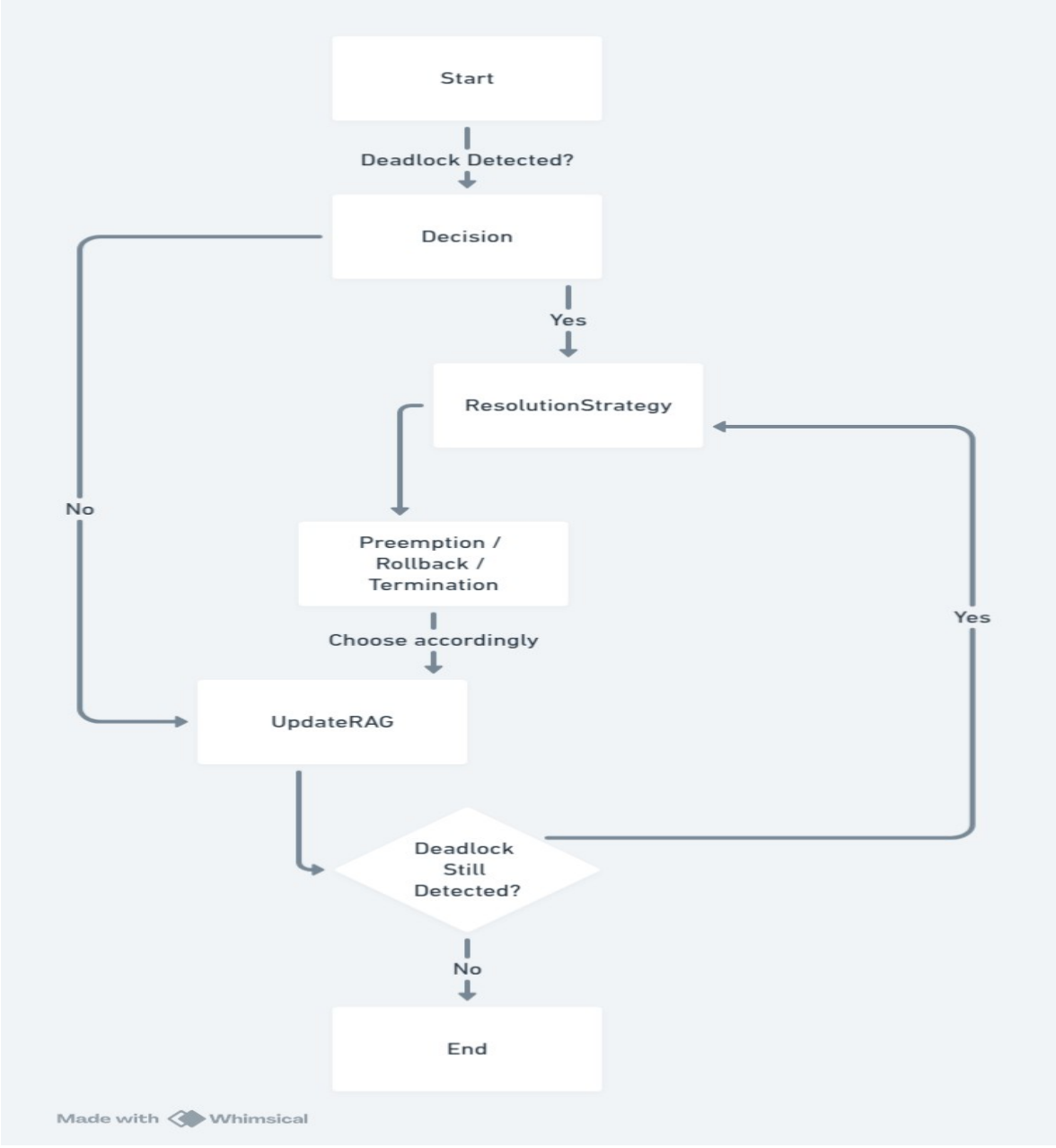
2. **Strategy Selection**: The system selects a resolution method:
   - **Preemption** if it is the least disruptive.
   - **Rollback** if the client can be safely restored to a previous state.
   - **Termination** if other strategies fail.

3. **Application**: Apply the chosen strategy:
    - **Preemption**: Reclaim a resource forcefully and assign it to another client.
    - **Rollback**: Undo the client's actions and release its resources.
    - **Termination**: Kill the client and release its resources.
    -
4. **Graph Update**: Update the **RAG** to reflect the released resources.

5. **Cycle Check**: Recheck for deadlock. If a deadlock still exists, repeat the process.

6. **Success Confirmation**: The process stops once the deadlock is resolved.

## Decision Flowchart for Resolution:



## *STEP 6-*

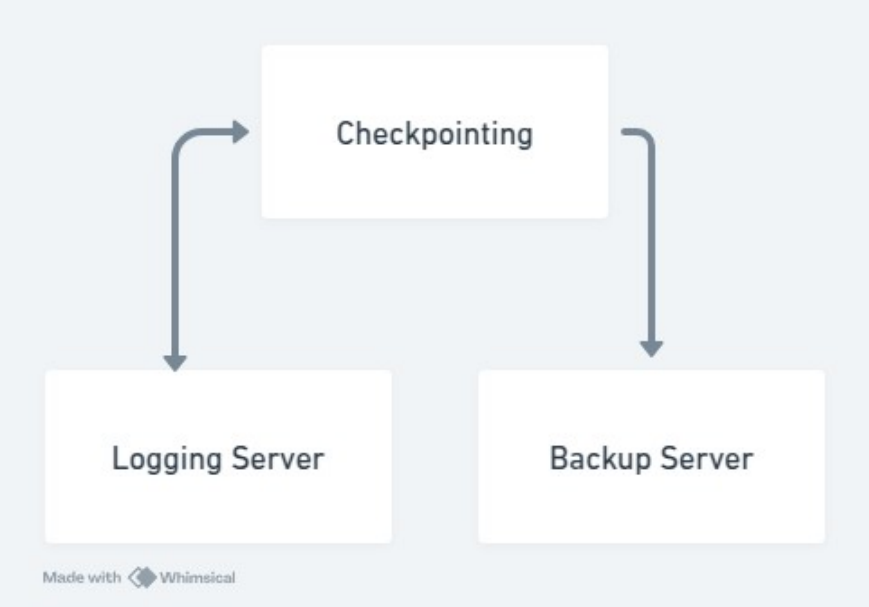
### Fault Tolerance Design:

To ensure that the deadlock detection and recovery system is fault-tolerant, we incorporate the following strategies:

**Checkpointing:** The system periodically saves the state of clients and resources to allow recovery in case of a crash. If a server crashes, it can restart from the last checkpoint.

**Logging:** All resource allocation events are logged. This log helps in identifying which resources were held by which clients before a failure and allows the system to revert to a consistent state.

**Redundancy:** Use multiple servers and duplicate logs to ensure that if one server fails, another can take over without data loss.




This system allows us to recover from server failures by using backup servers and logs.

### Fault Tolerance Algorithm:

1. Periodically **save checkpoints** of the current state of clients and resources.
2. **Log** every resource request, allocation, and release.
3. Maintain a **backup server** that can **take over** in case of primary server failure.
4. Upon **failure detection**:
    - Restore from the latest checkpoint.
    - Use logs to **replay recent resource operations** and **rebuild the Resource Allocation Graph (RAG)**.
    - Resume normal operation.

## Pseudo Code for Fault Tolerance System:

START

1.1. Set Timer for Periodic Checkpointing
2.2. WHILE System is Running:
3.a. Periodically SAVE system state as CHECKPOINT.
4.b. LOG every resource allocation, release, and client action.
5.3. IF Server Failure Detected:
6.a. SWITCH to Backup Server.
7.b. RESTORE state from the latest CHECKPOINT.
8.c. REPLAY Logs to rebuild the Resource Allocation Graph (RAG).
9.d. RESUME normal operations.

END

## Explanation of Fault Tolerance Algorithm:

**Checkpointing:** Captures the state of clients and resource allocations periodically

**.Logging:** Records all resource transactions for recovery.

**Backup Server:** Takes over when the primary server fails.

**Recovery:** On failure, state is restored from a checkpoint, and logs are replayed to ensure data consistency.

## *STEP 7-*

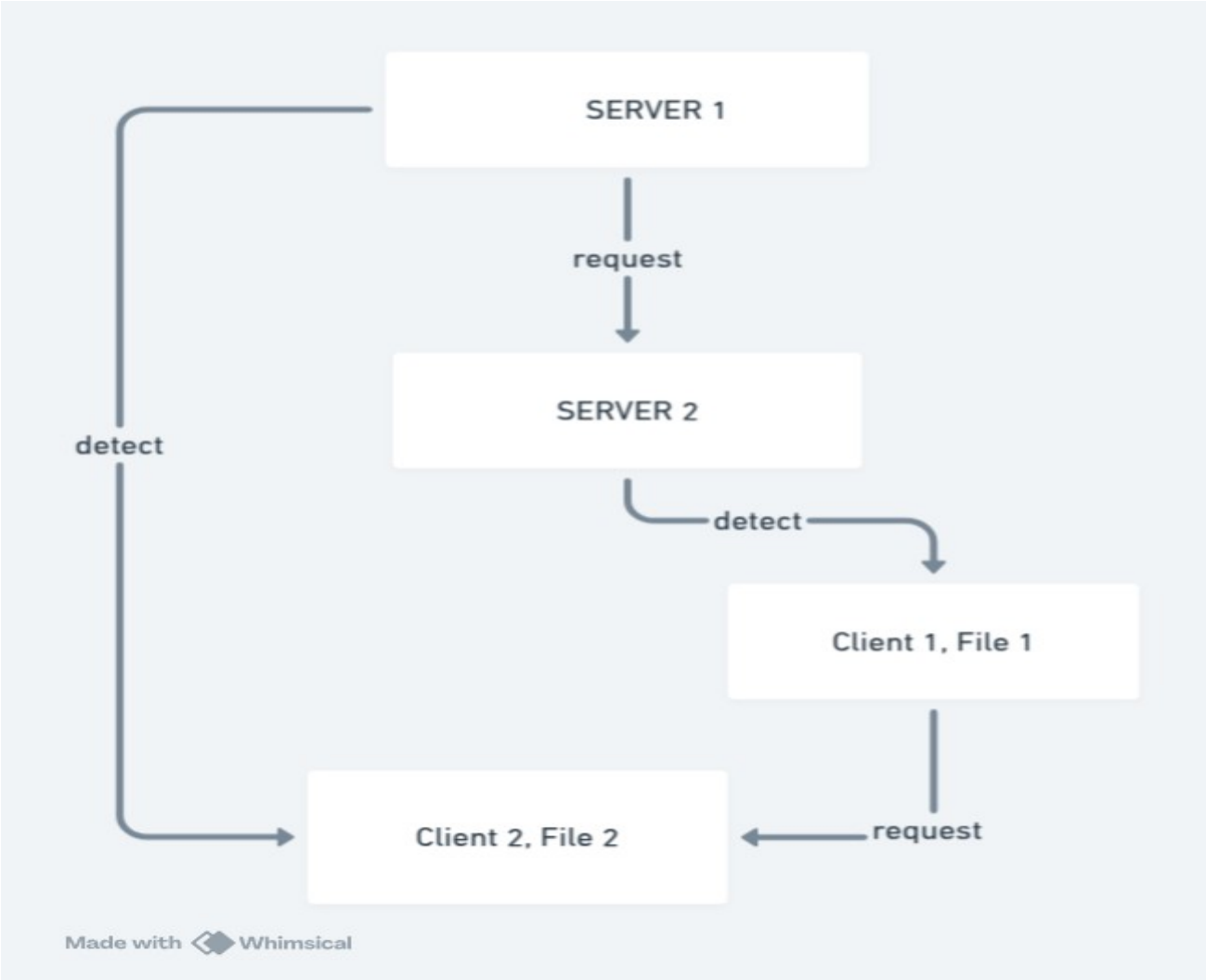
## Scalability Design

The system must scale to handle large numbers of clients and resources. To achieve this, we propose a distributed detection and recovery system:

**Distributed Detection:** Each server can detect deadlocks locally based on its clients and resources. Periodically, the servers will synchronize their findings to detect global deadlocks.

**Load Balancing:** Detection and resolution tasks will be distributed evenly across servers to avoid performance bottlenecks.

## Distributed System Design Diagram:




Servers detect local deadlocks and synchronize periodically to ensure the entire system is checked.

## Scalability Algorithm:

1. **Divide Clients and Resources** across multiple servers.
2. Each **Server Detects Deadlocks Locally** using DFS on its subset of clients and resources.
3. Servers **synchronize periodically** to **detect global deadlocks**.
4. **Distribute Load** evenly among servers by dynamically assigning clients and resources.

## Pseudo Code for Distributed Deadlock Detection:

START

1.1. DISTRIBUTE Clients and Resources among Servers.
2.2. Each SERVER:
3.a. Perform Local Deadlock Detection using DFS on local RAG.
4.b. PERIODICALLY Synchronize Deadlock Status with Other Servers.
5.3. IF Global Deadlock Detected:
6.a. COORDINATE Deadlock Resolution Strategy (Preemption, Rollback, Termination).

END

## Explanation of Scalability Algorithm:

- **Distributed Detection:** Servers handle local deadlocks and communicate to detect global cycles.
- **Load Balancing:** Balances detection and recovery tasks across servers to improve performance.
- **Synchronization:** Ensures a complete view of the system state across all servers.

## *STEP 8-*

## System Evaluation:

**Deadlock Detection Efficiency:**

- Our system employs **Depth-First Search (DFS)** to detect cycles in the **Resource Allocation Graph (RAG)**.
- This algorithm is efficient with a **time complexity of O(V + E)**, where:
    - **V** represents the number of nodes (clients + resources).
    - **E** represents the number of edges (request and hold relationships).
- Since the algorithm runs **periodically**, it ensures **timely detection of deadlocks** without excessive performance overhead.

**Resolution Strategies Analysis:**

Each deadlock resolution approach offers **specific advantages and trade-offs** depending on the situation:

| Strategy | Advantages | Disadvantages |
|---|---|---|
| **Preemption** | Quickly breaks deadlocks by forcefully reassigning resources. | Can disrupt ongoing client operations, causing dissatisfaction. |
| **Rollback** | Restores the system to a consistent state before the deadlock occurred. | May result in the loss of some work, requiring clients to redo tasks. |
| **Termination** | Ensures fast recovery by terminating processes involved in the deadlock. | Drastic action—results in complete loss of progress for terminated clients. |

---

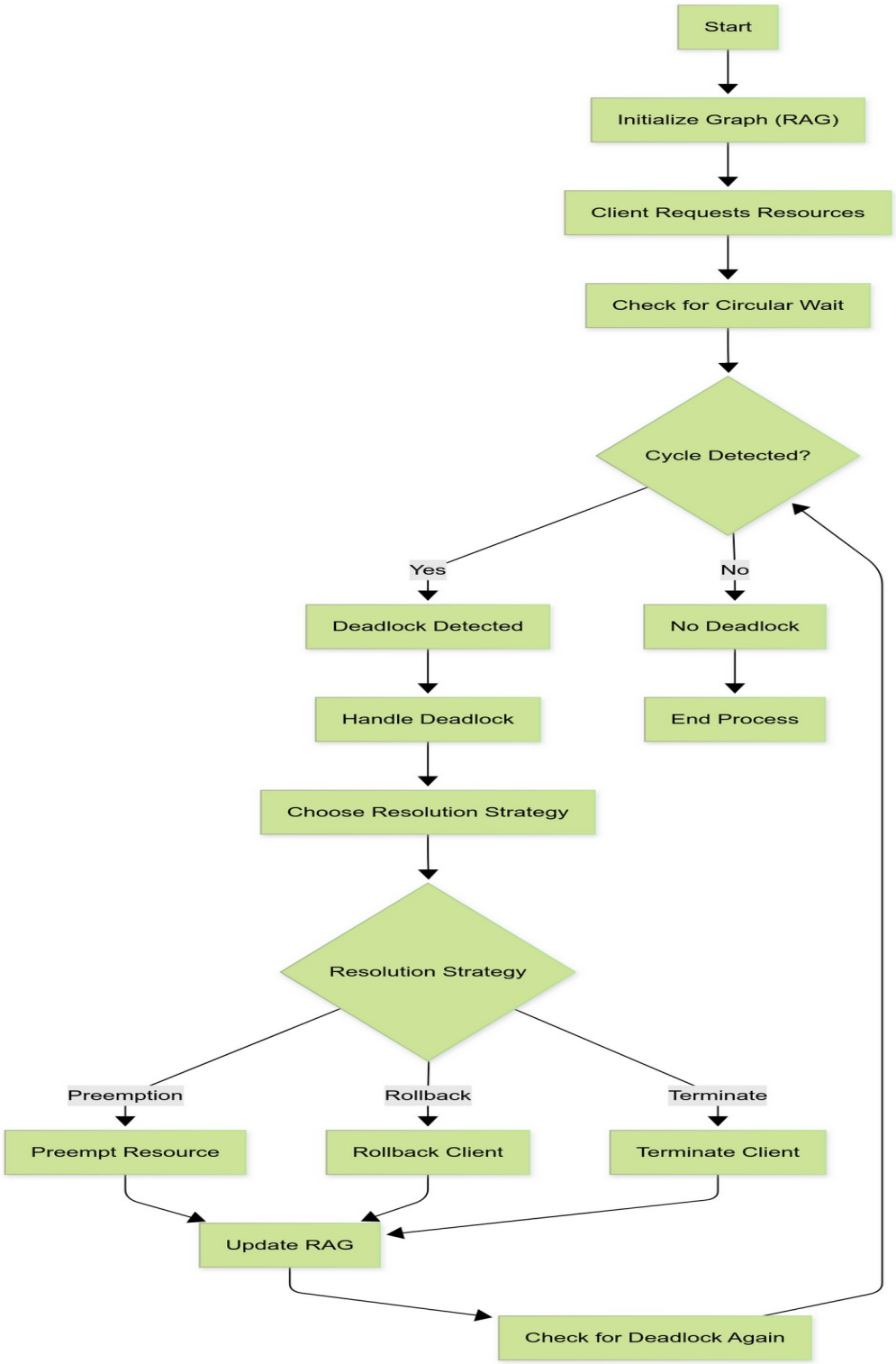
**Fault Tolerance Robustness:**

- The system incorporates **checkpointing, logging, and redundancy** to **recover from crashes** and **network failures**.
- This design ensures that **deadlock detection and resolution processes can continue without data loss** even if individual servers fail.

**Scalability Strength:**

- The system **adopts a distributed detection and recovery mechanism**, allowing **multiple servers** to:
    - **Handle deadlock detection locally** for their assigned clients and resources.
    - **Synchronize periodically** to **detect system-wide deadlocks**.
- This **distributed approach** prevents bottlenecks and ensures **smooth performance** as the **number of clients and resources grows**.

## Workflow Diagram of Deadlock Detection and Recovery System:



## *STEP 9-*

## Conclusion:

In this experiment, we successfully implemented a deadlock detection and recovery system to address the challenges posed by deadlocks in concurrent computing environments. By utilizing efficient algorithms and recovery mechanisms, we ensured that the system could detect and handle deadlocks in real-time, maintaining the smooth execution of processes. The workflow diagram provided a clear understanding of how deadlock detection is integrated into the system, showing the sequence of steps taken from identifying potential deadlocks to recovering from them. Through this experiment, we demonstrated the importance of timely detection and effective recovery strategies in enhancing system stability and performance. Ultimately, this approach significantly reduces the risk of system crashes due to deadlocks, ensuring a more reliable and robust system.