Deadlock Detection
Deadlock Detection

- Important problem in distributed databases
- General Issues
  - Reusable vs. consumable resources
  - Resource vs. communication deadlock
  - Wait-For graphs (WFG)
  - Prevention, Avoidance, Detection??
  - Resolution??
Deadlock Detection Strategies

Requirements:
- No undetected deadlocks
- No “false” or “phantom” deadlocks

Strategies:
- Centralized
- Distributed
- Hierarchical
A Simple Algorithm (Ho-Ramamoorthy)

- Each node has a status table, contains status (resources locked and resources waited on) of all processes at that node.
- A central site periodically collects the status table from all nodes, constructs the WFG and checks for cycles.
- If no cycle detected, no deadlock.
- If cycle detected, status from all nodes requested again and WFG constructed using ONLY information common both times. If the same cycle is detected again, deadlock is declared.
- Does not work!! Why??
Chandy-Misra-Haas Algorithm for AND Deadlocks

- Distributed control
- Asynchronous, reliable and FIFO communication
- An “Edge-Chasing” algorithm
  - A blocked process initiates by sending messages to all processes it is waiting for
  - Those in turn send to processes they are waiting for
  - If the initiator’s message comes back to it, we have a cycle
  - The algorithm follows the edges of the WFG, hence the name
Algorithm

- Controller: controls a set of resources and processes at a machine
- Processes belonging to a controller requests the controller for a resource
- If the resource is not available locally, controller will request other controllers in other machines for it on behalf of the process
- A process i is dependant on another process j if there exists a path from i to j in the WFG
- If i and j are in the same node, i is locally dependent on j
- Process i is idle if it is waiting for some resource (must get all resources it is waiting for to proceed)
Algorithm

- Uses a special probe message of the form \((i, j, k)\) where:
  - \(i\) : process originally initiating deadlock detection
  - \(j\) : current sender of the probe
  - \(k\) : current destination/receiver of the probe

- Each controller has
  - a boolean array \(\text{dependent}_k\) for each process \(k\) it controls
  - \(\text{dependent}_k[i] = \text{true}\) iff process \(i\) is dependent on process \(k\)
    - Initially false for all \(i\)
Initiation:

- If a process i is locally dependent on itself (local cycle on same controller), declare deadlock and exit.
- Probe (i, j, k) is sent by process i when:
  - i is locally dependent on j
  - j is waiting on k
  - j and k are on different controllers
- Probe sent for all such j, k
On receiving probe (i, j, k)

- If process k is idle, dependent\(_k[i]\) = false, and process k has not released all resources needed by j
  - dependent\(_k[i]\) = true
  - If k = i declare deadlock and exit (probe has returned to the initiator)
- Else for all a, b such that process k is locally dependent on process a, process a is waiting for process b, and a and b are on different controllers
  - Send probe (i, a, b) to process b