Lecture #9: Distributed Deadlock Detection

These topics are from Chapter 7 (Distributed Deadlock Detection) in *Advanced Concepts in OS*

Topics for Today

- Review: Deadlock
  - Definition
  - Models for Requests: Single-Unit request model, AND request model, OR request model,
  - General Resource Graph
  - Necessary and Sufficient conditions for a deadlock
- Distributed deadlock detection algorithms

System Model

- The systems have only reusable resources.
- Processes are allowed only exclusive access to resources.
- There is only one copy of each resource.

Distributed Deadlock Models

Based on WFG (not GRG)

- Nodes are processes
- Resources are located at a site, but may be held by processes at other sites
- Edge (P,Q) indicates P is blocked and waiting for Q to release some resource
- Deadlock exists iff there is a directed cycle or knot.

The text fudges a bit by forgetting that in some models the existence of a cycle is necessary but not sufficient, and the existence of a knot is sufficient but not necessary, leaving a gap.

Distributed Deadlock Handling Strategies

- Deadlock prevention
  - All resource at once.
  - Preventing a process from holding while waiting

    - inefficient, can become deadlocked at resource acquiring phase, resource requirements are unpredictable -- not an efficient, universal solution.
- Deadlock avoidance
    - A resource is granted to a process if the resulting state is safe
    - Every site has to maintain the global state
    - The checking for a safe state must be done with mutual exclusion
    - The number of processes and resources in a distributed system is large
    - **Not a practical solution**
- Deadlock detection
    - Once deadlock, always deadlock -- detection won't be outdated
    - deadlock detection can be preceed concurrently with normal activities
    - **This is the usual approach** -- the focus of this Chapter

Distributed Deadlock Detection Issues

- issues
    - maintenance of WFG
    - detection of cycles (or knots) in the WFG
- requirements
    - progress = no undetected deadlocks
    - safety = no false deadlocks

What do we do when a deadlock is detected?

Categorization of Methods

- centralized control
- distributed control
- hierarchical control

Simple Centralized Control

- one *control site* maintains the WFG and checks for deadlock
- other sites send request and release messages to control site

What are the advantages and disadvantages?

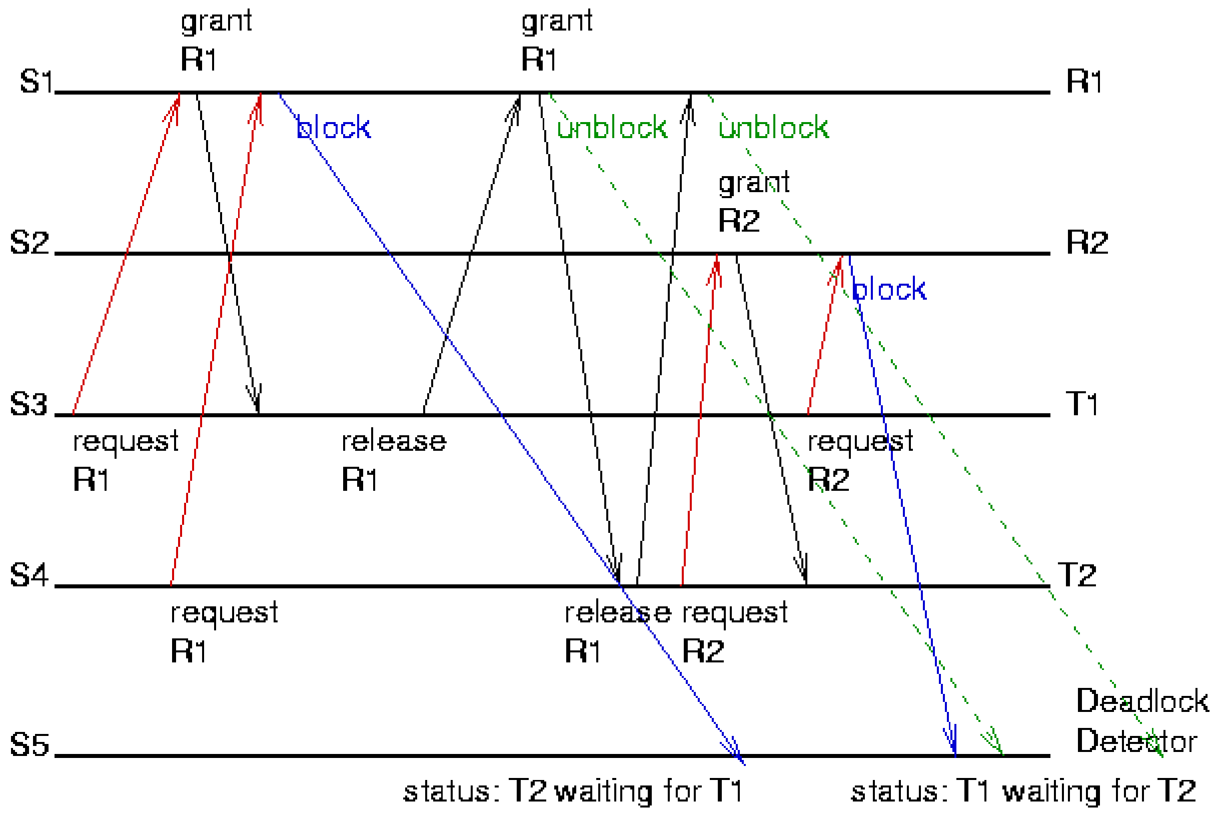
If we try to improve this algorithm by having each site maintaining its own WFG locally and sending the information to the control site periodically, we may have false deadlocks:

False Deadlock Example

Two transactions run concurrently:

$$T_1\text{: lock } R_1 \begin{array}{l} T_2\text{: lock } R_1 \\ T_1\text{: unlock } R_1 \end{array} \begin{array}{l} T_2\text{: unlock } R_1 \\ T_1\text{: lock } R_2 \end{array} \begin{array}{l} T_2\text{: lock } R_2 \\ T_1\text{: unlock } R_2 \end{array} T_2\text{: unlock } R_2$$

False Deadlock Example: Event Trace Diagram



Ho-Ramamoorthy Algorithm: Two-Phase

- each site maintains table with status of all local *processes*
- control site periodically requests status from all sites, builds WFG, and checks for deadlock
- if deadlock detected, control site repeats status requests, but throws out transactions that have changed

May still report false deadlocks.

Ho-Ramamoorthy Algorithm: One-Phase

- each site maintains two tables:
    - resource status (for all local resources)
    - process status (for all local processes)
- control site periodically requests copies of tables, builds WFG, and checks for deadlock
- transactions are not used unless the process table info agrees with the resource table info

Notice the similarity in principle here to the Chandy-Lamport global sate recording algorithm, *i.e.*, we need to capture not just the states of the processes but also the states of the messages in transit.

Classification of Distributed Detection Algorithms

- path-pushing
  - path information transmitted, accumulated
- edge-chasing
  - ``I'm waiting for you'' probes are sent along edges
  - single returned probe indicates a cycle
- diffusion
  - ``Are you blocked?'' probes are sent along all edges
  - all queries returned indicates a cycle
- global state detection
  - take and use snapshot of system state

Obermarck's Path-Pushing Algorithm

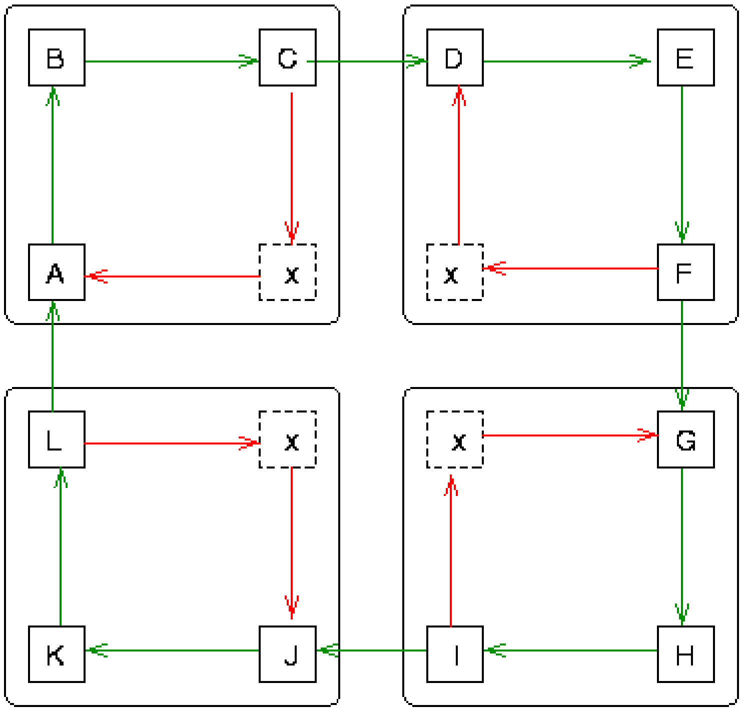
- designed for distributed database systems
- processes are called ``transactions'' T_1, T_2, ¼T_n
- there is special virtual node *Ex*
- transactions are totally ordered

How can we totally order the transactions?

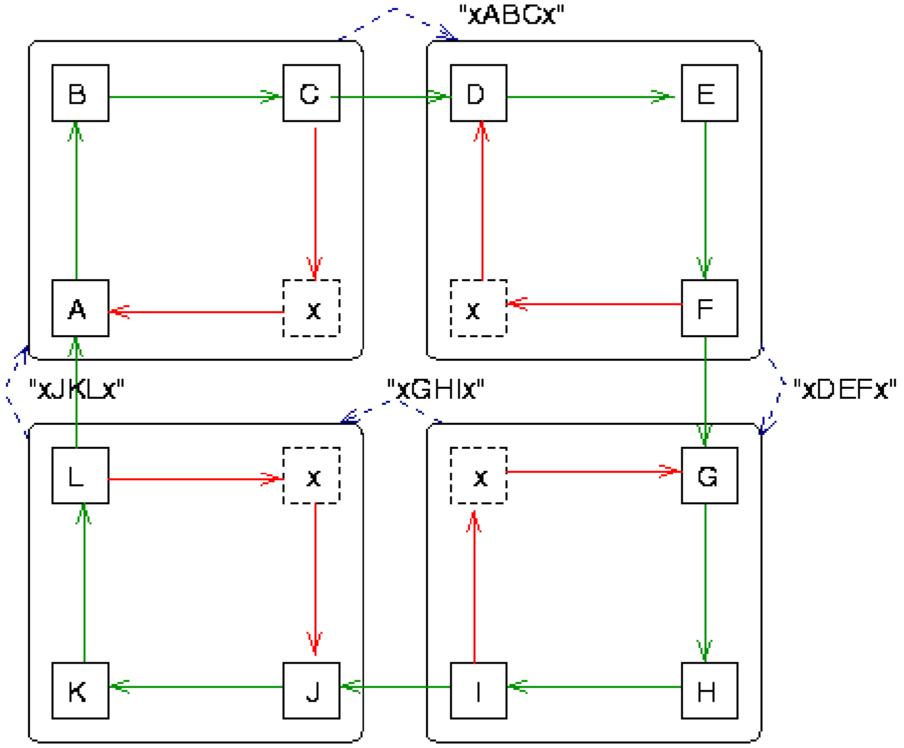
Obermarck's Path-Pushing Algorithm

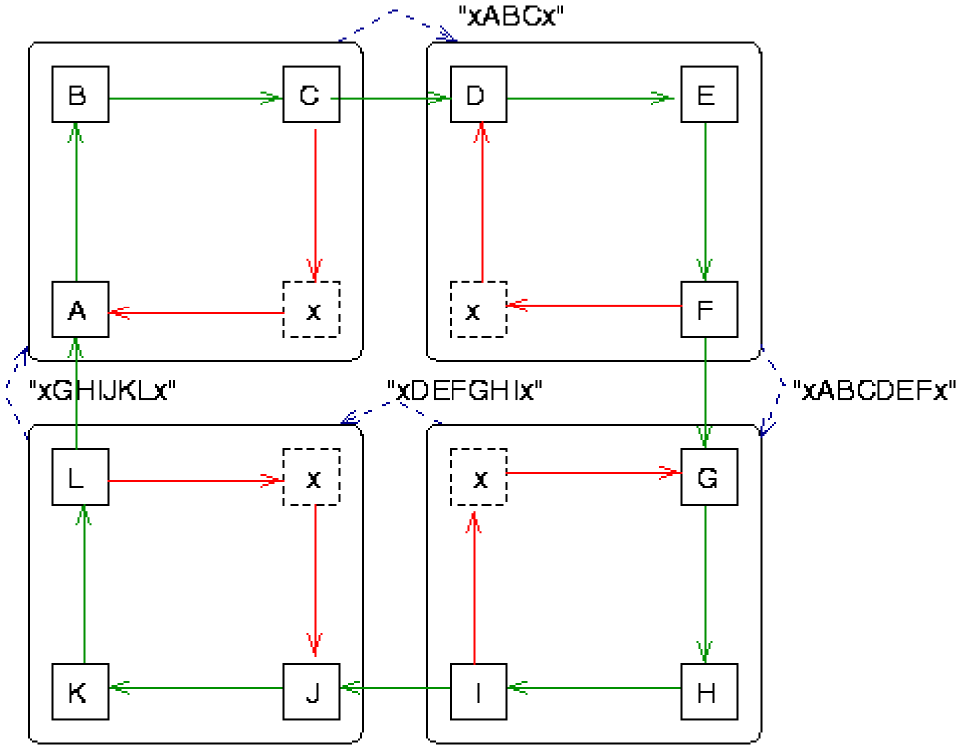
1. wait for info from previous iteration of Step 3
2. combine received info with local TWFG
   detect all cycles
   break local cycles
3. send each cycle including the node *Ex*
   to the external nodes it is waiting for
4. time-saver: only send path to other sites if last
   transaction is higher in lexical order than the first

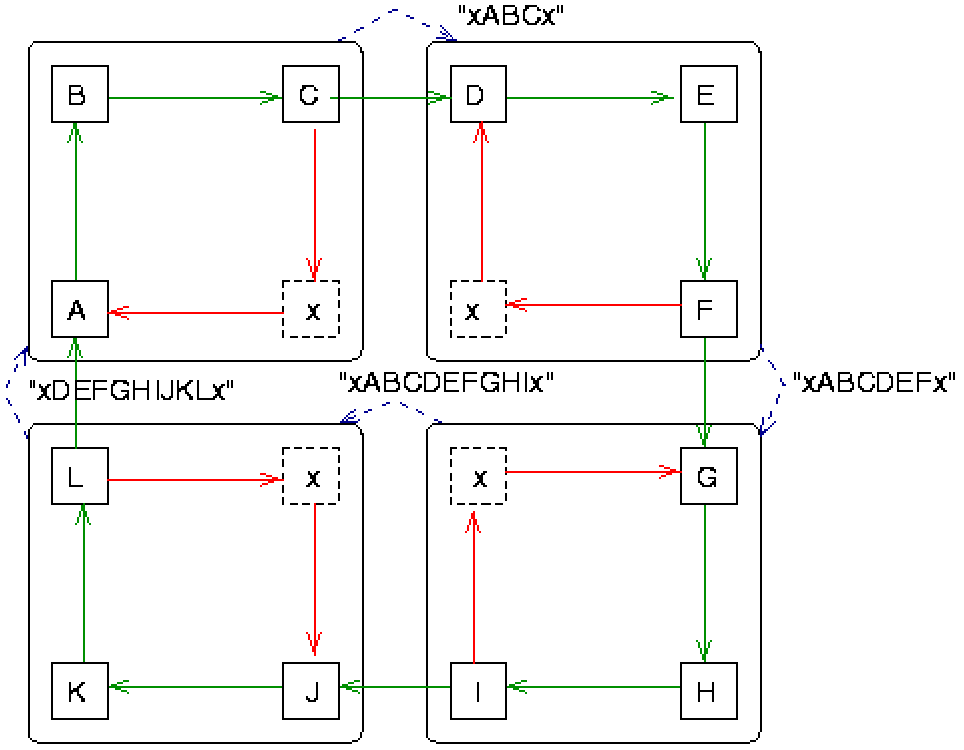
Note: In the textbook, the rule above is reversed, i.e., it says to only send the path of the first transaction is higher in lexical order than the first. Either rule will work. The essential idea is simply to have one

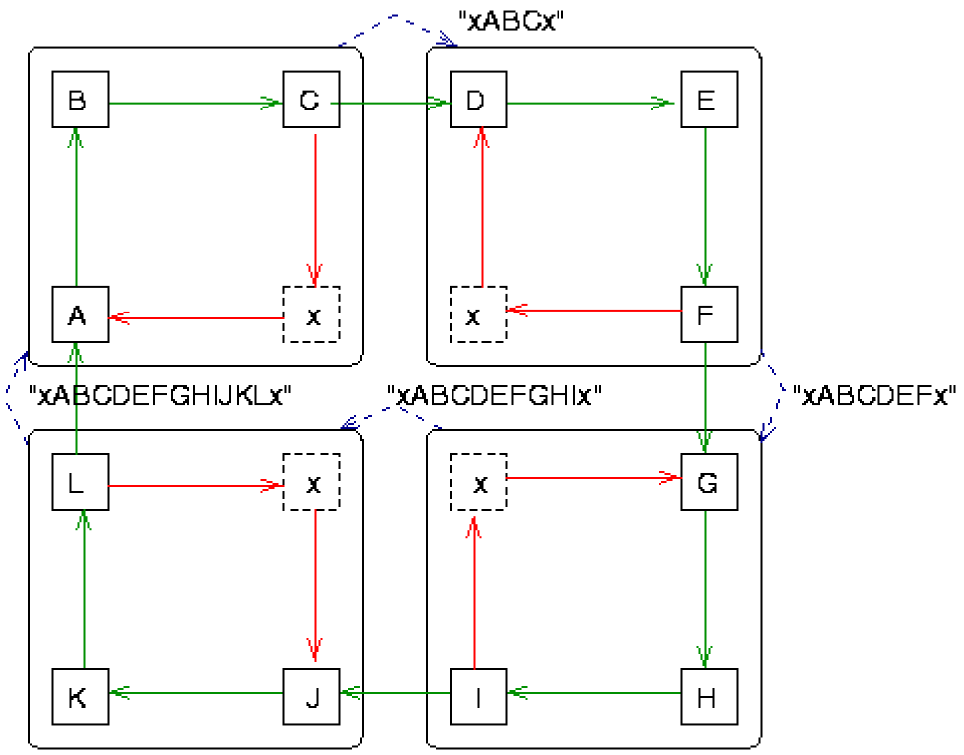


canonical representation of each path.




"xABCx"
B
C
D
E
A
x
x
F
"xGHIJKLx"
"xDEFGHIx"
"xABCDEFx"
L
x
x
G
K
J
I
H


"xABCx"
B
C
D
E
A
x
x
F
"xDEFGHIJKLx"
"xABCDEFGHIx"
"xABCDEFx"
L
x
x
G
K
J
I
H



Problems with Obermarck's Path-Pushing Algorithm

Detects false deadlocks, due to asynchronous snapshots at different sites.

Message complexity? Message size? Detection delay?

Exactly how are paths combined and checked?

Obermarck's Path-Pushing Algorithm: Performance

- $O(n(n-1)/2)$ messages
- $O(n)$ message size
- $O(n)$ delay to detect deadlock

Chandy-Misra-Haas Edge-Chasing Algorithm

- for AND request model
- *probe*= (i,j,k) is sent for

detection initiated by P_i,
by site of P_j to site of P_k

- deadlock is detected when a probe returns to its initiator

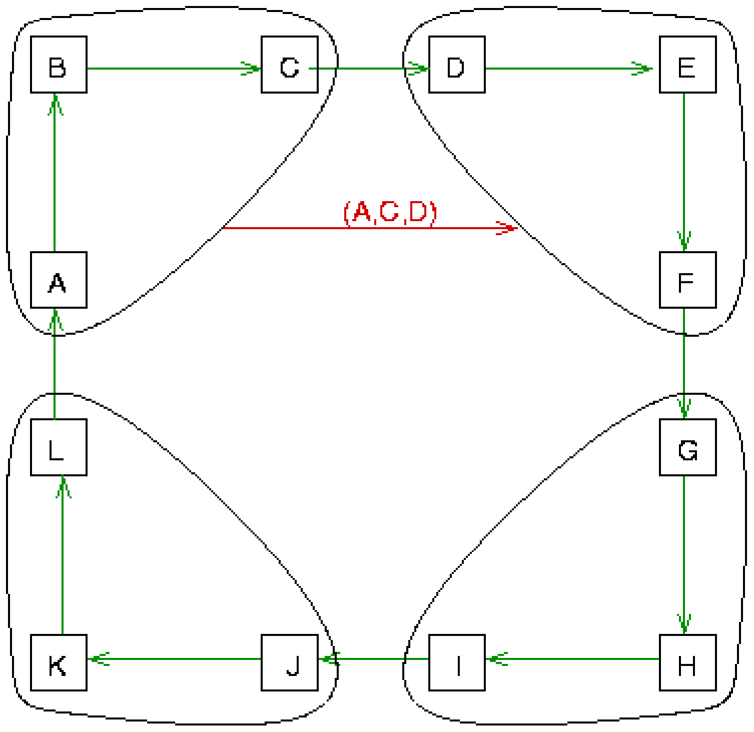
Terminology

- P_j is dependent on P_k if there is a sequence $P_j, P_{i1}, P_{i2}, ¼P_{im}, P_k$ such that each process except P_k is blocked and each process except the first holds a resource for which the previous process is waiting
- P_j is locally dependent on P_k if it is dependent and both processes are at the same site
- array $dependent_i(j) = true$ Û P_i knows that P_j is dependent on it

Algorithm Initiation by P_i

if P_i is locally dependent on itself then declare a deadlock
else send probe (i, j, k) to home site of P_k for each j, k such that all of the following hold

- P_i is locally dependent on P_j
- P_j is waiting on P_k
- P_j and P_k are on different sites



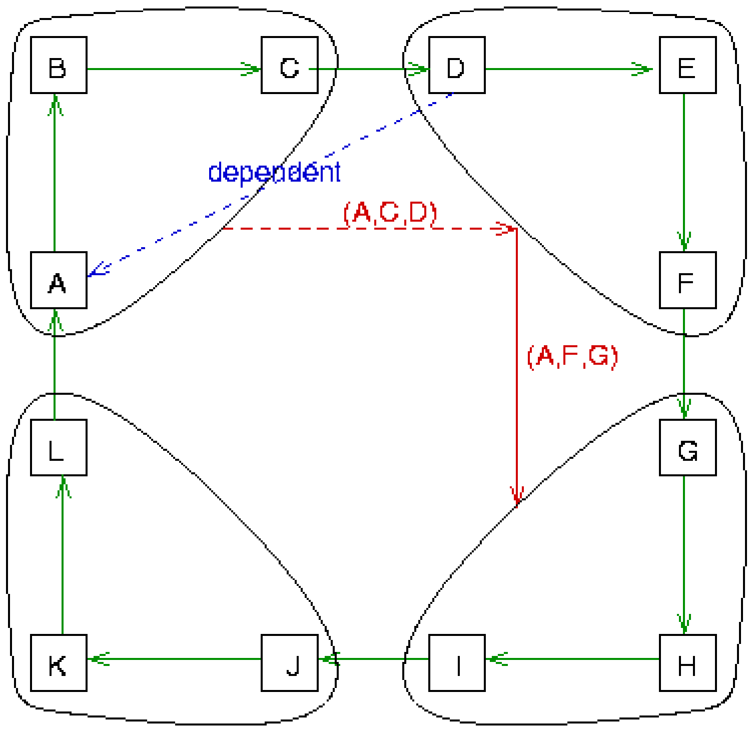
Algorithm on receipt of probe (i,j,k)

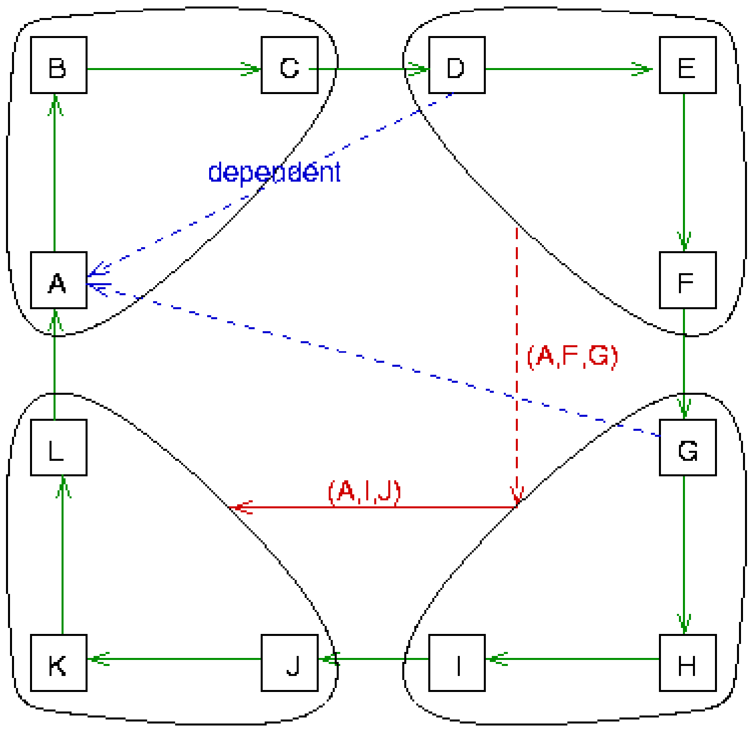
check the following conditions

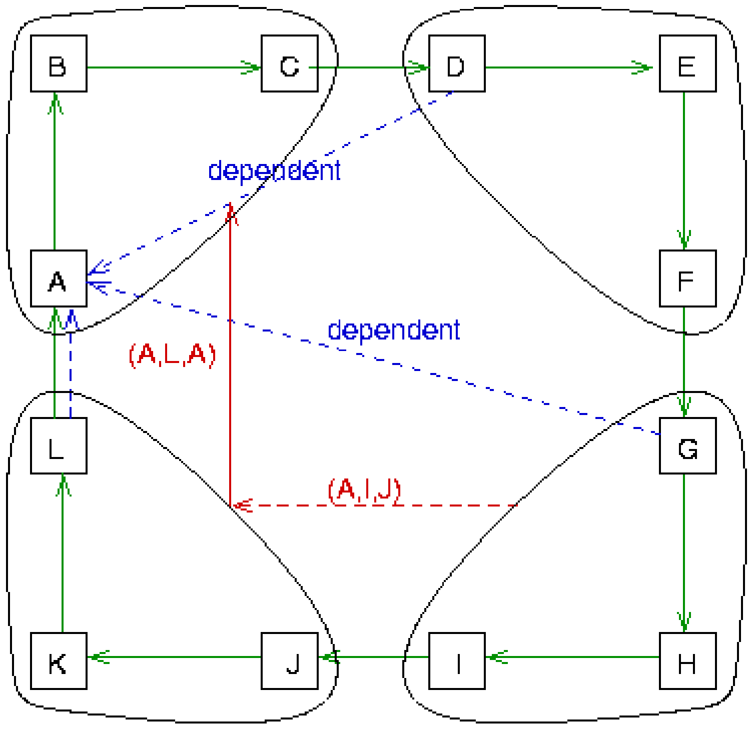
- P_k is blocked
- $dependent_k(i) = false$
- P_k has not replied to all requests of P_j

if these are all true, do the following

- set $dependent_k(i) = true$
- if $k=i$ declare that P_i is deadlocked
- else send probe (i,m,n) to the home site of P_n for every m and n such that the following all hold
  - P_k is locally dependent on P_m
  - P_m is waiting on P_n
  - P_m and P_n are on different sites







Chandy Misra Haas Complexity

- What is the message complexity?
- What is delay in detection?

Analysis

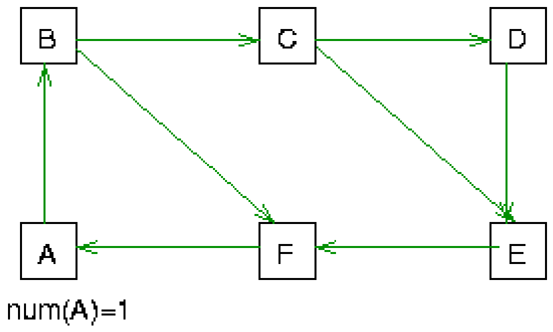
- $m(n-1)/2$ messages for m processes at n sites in the book, is this right?
- 3-word message length
- $O(n)$ delay to detect deadlock

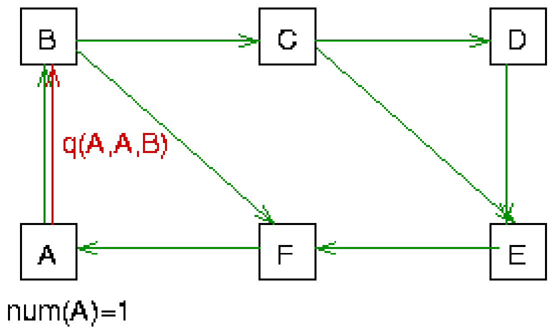
Diffusion Based Algorithms: Chandy *et al.*

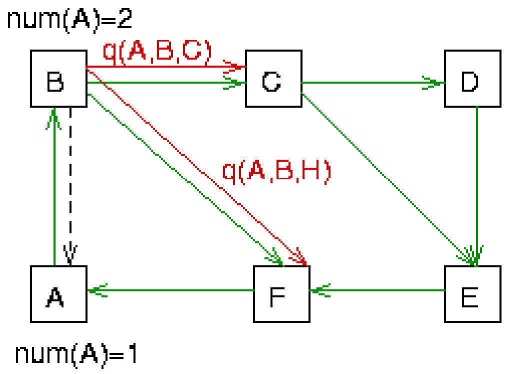
- for OR request model
- processes are active or blocked
- A blocked process may start a diffusion.
- if deadlock is not detected, the process will eventually unblock and terminate the algorithm
- message = *query* (i,j,k)
  - i = initiator of check
  - j = immediate sender
  - k = immediate recipient
- reply = *reply* (i,k,j)

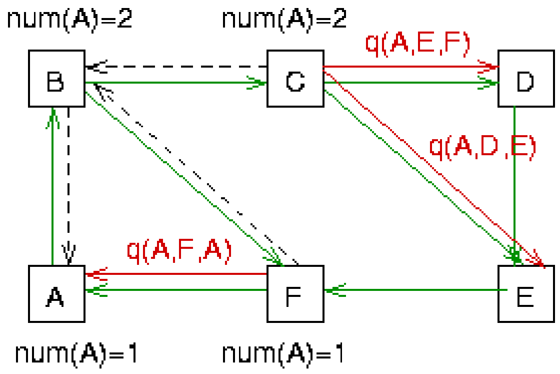
On receipt of *query*(*i*,*j*,*k*) by *m*

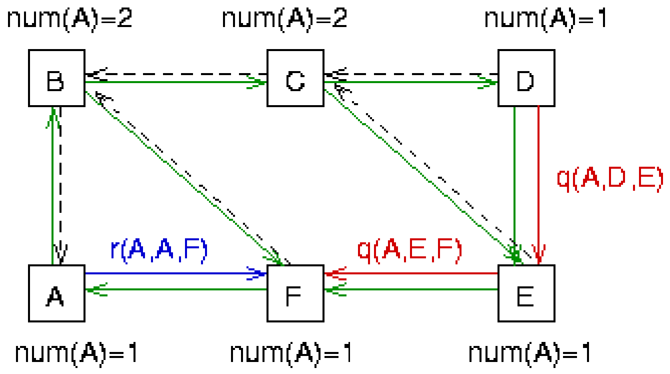
- if not blocked then discard the query
- if blocked
  - if this is an *engaging* query
    propagate *query*(i,k,m) to dependent set of m
  - else
    - if not continously blocked since engagement then discard the query
    - else send *reply*(i,k,j) to j









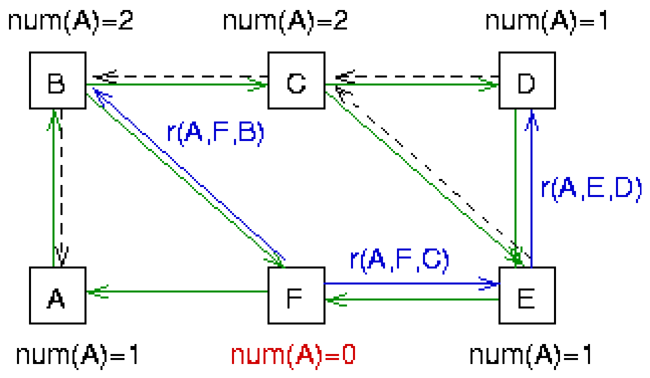


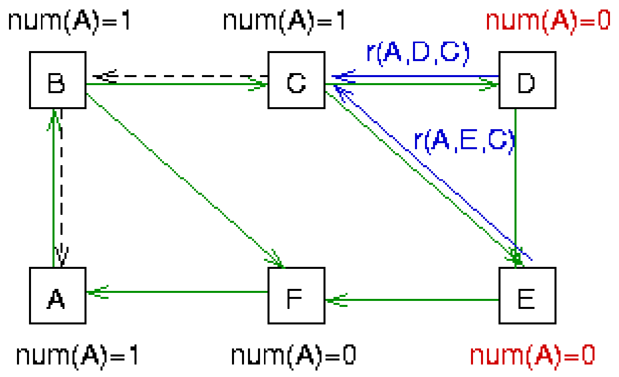
On receipt of *reply*(*i*,*j*,*k*) by *k*

- if this is not the last reply

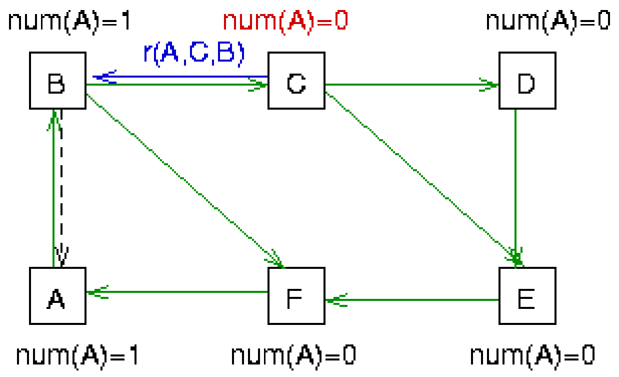
then just decrement the awaited reply count

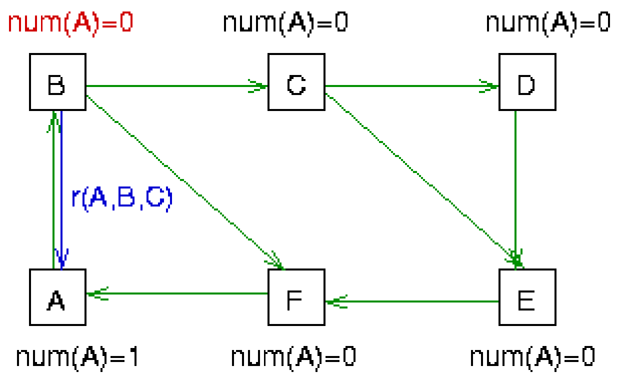
- if this is the last reply then
    - if *i*=*k* report a deadlock
    - else send *reply*(*i*,*k*,*m*)
      to the engaging process *m*



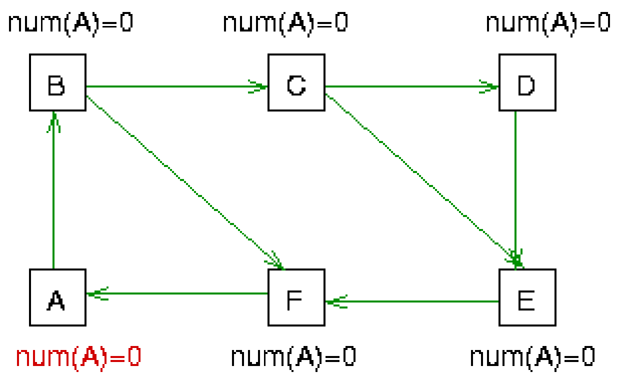


The black dashed arrows indicate the engaging process for each engaged process. This information is needed to route the reply when the number of other processes for which the engaged process is waiting goes to zero.





Observe that the engaging process arrows form a *spanning tree* of the subgraph corresponding to the set of process for which the initiating process is waiting. If every process in this subgraph is blocked, we have a *knot*.



At this point, a knot has been detected.

Global State Detection Based Algorithms

- take snapshot of distributed WFG
- use graph reduction to check for deadlock

Details differ.

Recall: What is graph reduction?

Graph Reduction

General idea: simulate the result of execution, assuming all unblocked processes complete without requesting any more resources

- while there is an unblocked process, remove the process and all (resource-holding) edges to it
- there is deadlock if the remaining graph is non-null