Distributed Deadlock Detection

Introduction

- **Deadlock:** a situation where a process or set of the process is blocked, waiting on an event that will never occur.

- Avoiding performance degradation due to deadlocks requires that a system be deadlock free or that deadlocks be quickly detected and eliminated.
- Deadlock in distributed systems are similar to deadlocks in single processor systems, only worse.
  - They are harder to avoid ,prevent or even detect.
  - They are hard to cure because all relevant information is scattered over many machines.
- In Distributed Systems, a process can request and release resources in any order
- If the sequence of allocation is not controlled, deadlocks can occur

Assumptions:

- System has only reusable resources
- Only exclusive access to resources
- Only one copy of each resource
- States of a process: running or blocked
  - Running state: process has all the resources
  - Blocked state: waiting on one or more resource

Types of Resources

- Reusable resource:
  - Description:
    - Used by one process at a time and not consumed by that use
    - Processes obtain resources that they later release for reuse by other processes
  - Examples:
    - Processor , I/O channels, main and secondary memory, files, databases
- Consumable resources
  - Description:
    - Created (produced) and destroyed (consumed) by a process
  - Examples:
    - Interrupts, signals, messages, and information in I/O buffers

Necessary Conditions for Deadlock

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource (One process holds a resource in a non-sharable mode & Other processes requesting resource must wait for resource to be released).
- **Hold and wait:** a process holding resource(s) is waiting to acquire additional resources held by other processes.
- **No preemption:** Resources not forcibly removed from a process holding it, a resource can be released only voluntarily by the process upon its task completion.

Necessary Conditions for Deadlock

- **Circular wait:** A closed chain of processes exists.
  A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that :

  P_0 is waiting for a resource that is held by P_1,

  P_1 is waiting for a resource that is held by P_2, ...,

  P_{n-1} is waiting for a resource that is held by P_n,

  and P_n is waiting for a resource that is held by P_0.

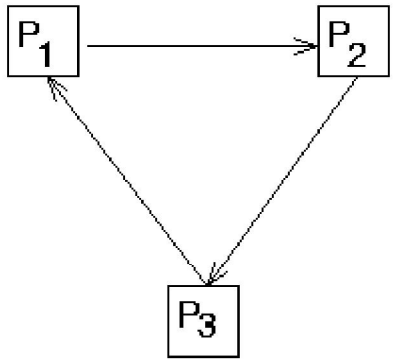
Types of Deadlocks

- Resource Deadlocks
  - Processes can simultaneously wait for several resources and can not proceed until they have acquired all those resources.
  - Deadlock occurs if each process in a set request resources held by another process in the same set, and it must receive all the requested resources to move further.

- Communication Deadlocks
  - Each process is waiting for communication (process's message) from another process, and will not communicate until it receives the communication for which it is waiting.

Wait-For Graphs (WFG)

- In DS, the system state can be modeled by a directed graph called a *wait-for-graph*(WFG).

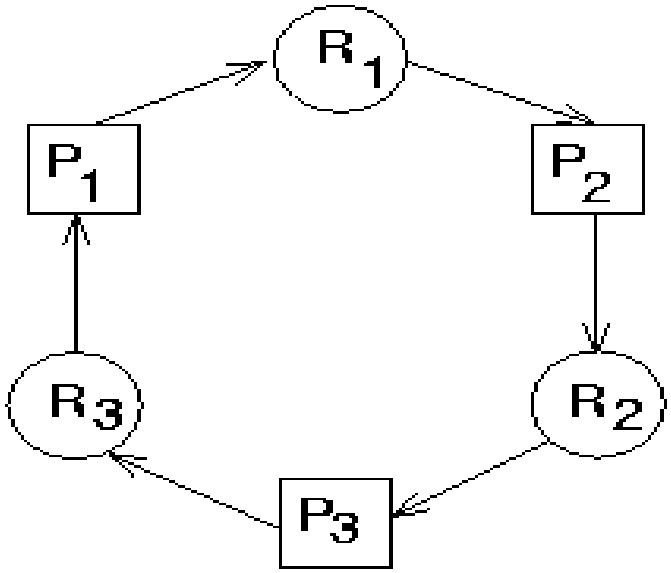


- Wait-for Graphs (WFG): P1 -> P2 implies P1 is waiting for a resource from P2.

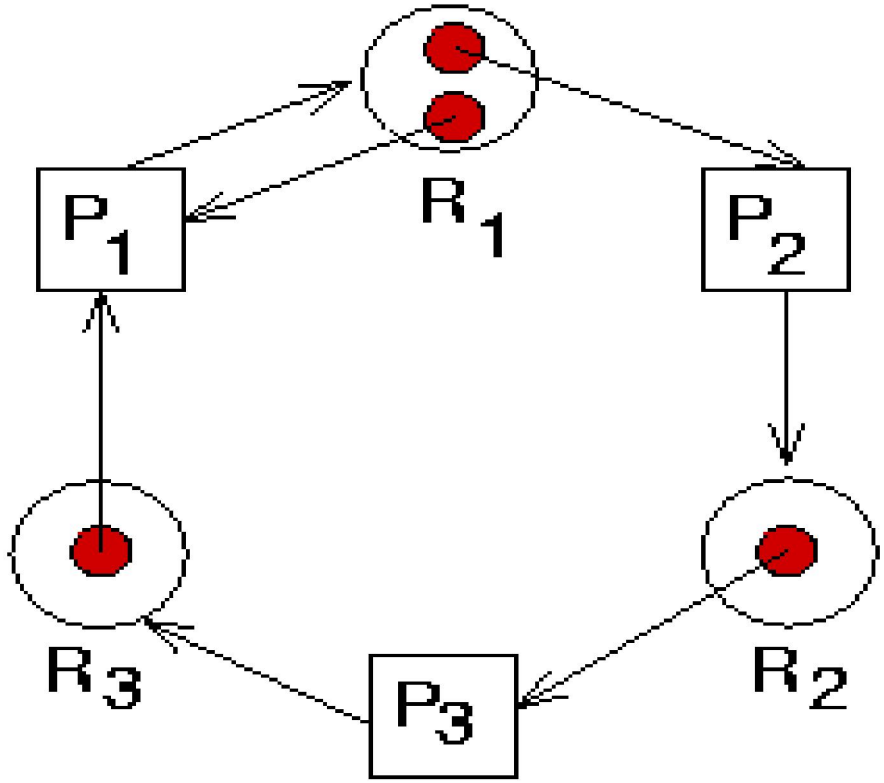
- WFG in databases
  - Transaction-wait-for Graphs (TWF)
    - Nodes are transactions and T1→T2 indicates that T1 is blocked and waiting for T2 to release some resource

Single-Unit Resource Allocation Graphs

- Nodes correspond to processes and resources.
- The simplest request model since a process is resticted to requesting only one unit of a resource at a time.
- The outdegree of nodes in the WFG is one.
- A deadlock corresponds to a cycle in the wait-for-graph, provided that there is only one unit of every resource in the system.



Multiunit Resource Allocation Graphs



Resource allocation graph

- To represent the state of process & resource interaction in DS

- Nodes of a graph → processes and resources.
- Edges of a graph → pending requests or assignment of resources.

- A pending request is represented by a *request edge* directed from the node of a requesting process to the node of the requested resource

- A resource assignment is represented by an *assignment edge* directed from the node of an assigned resource to the node of the assigned process.

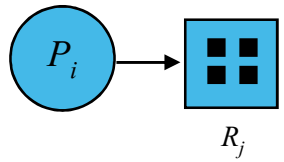
Resource Allocation Graph

- Process

- Resource Type with 4 instances

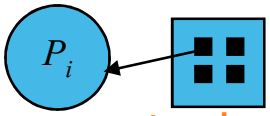
The sequence of Process's resource utilization

- P_i requests instance of R_j



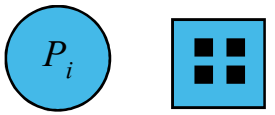
Request edge

- P_i is holding an instance of R_j



Assignment edge

- P_i releases an instance of R_j

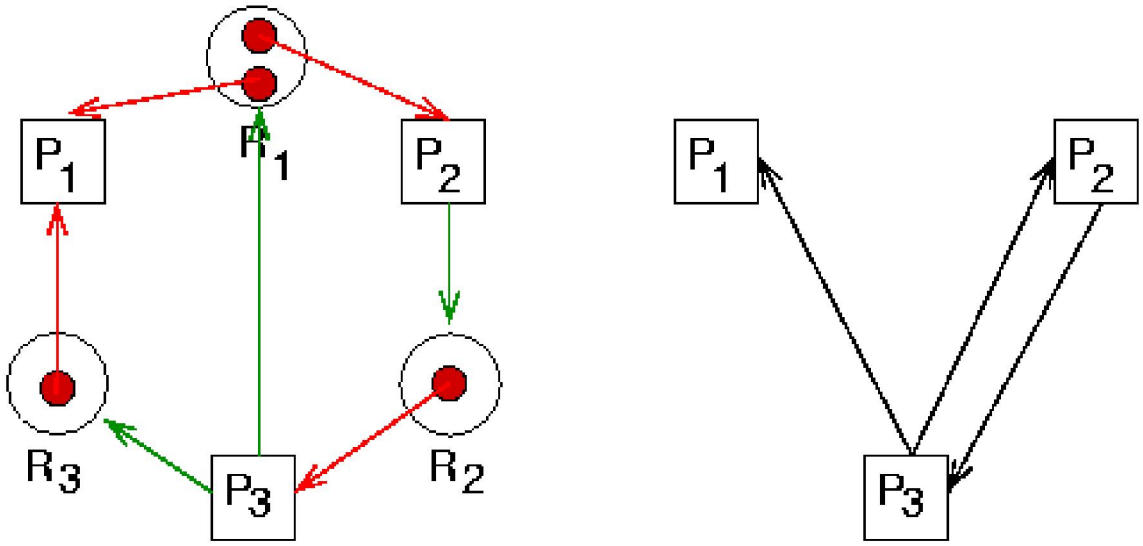


Resource allocation graph

- System is deadlocked if its resource allocation graph contains a *directed cycle* or a *knot*
  - A **knot** in a directed graph is a collection of vertices and edges with the property that every vertex in the knot has outgoing edges, and all outgoing edges from vertices in the knot terminate at other vertices in the knot.

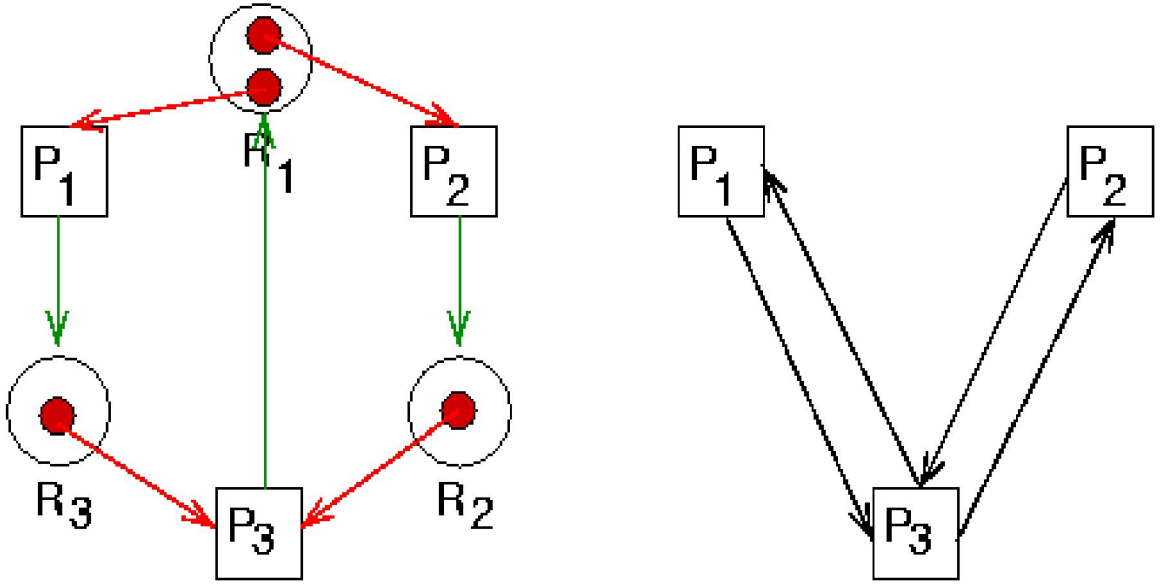
Resource allocation graph

- A cycle with no deadlock, and no knot.



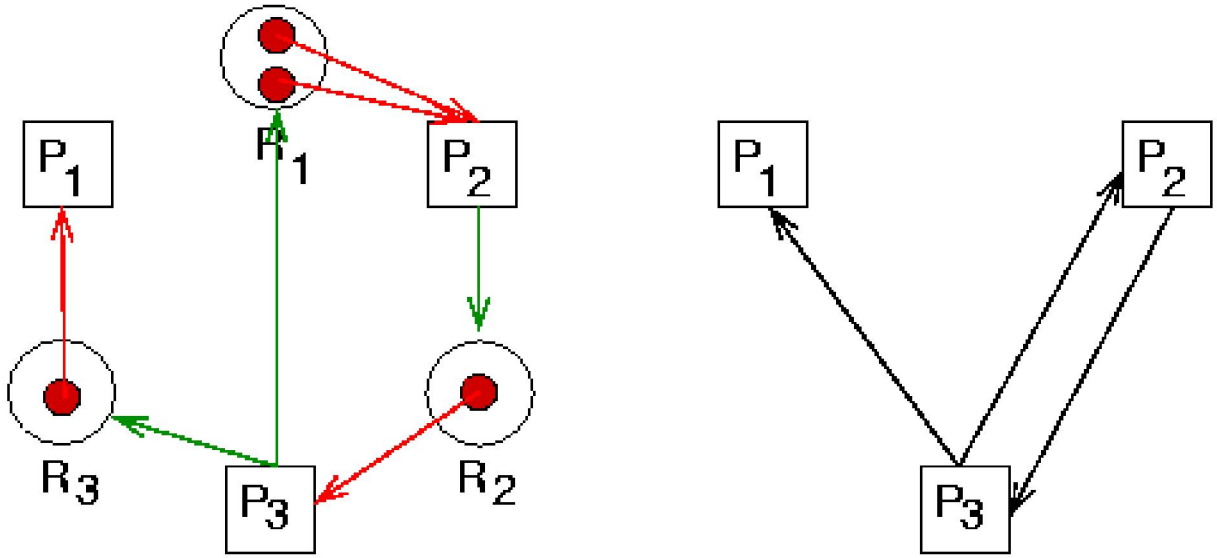
Resource allocation graph

- A knot is a sufficient condition for deadlock.

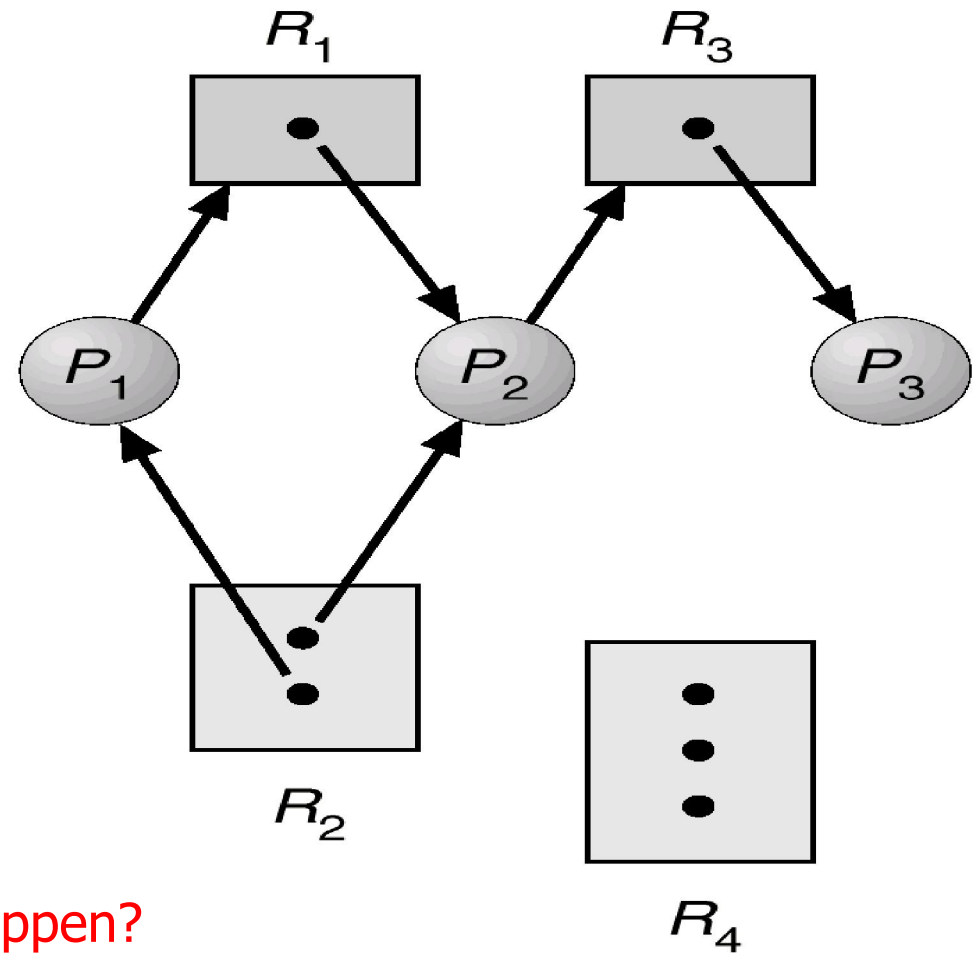


Resource allocation graph

- A knot is not a necessary condition for deadlock.

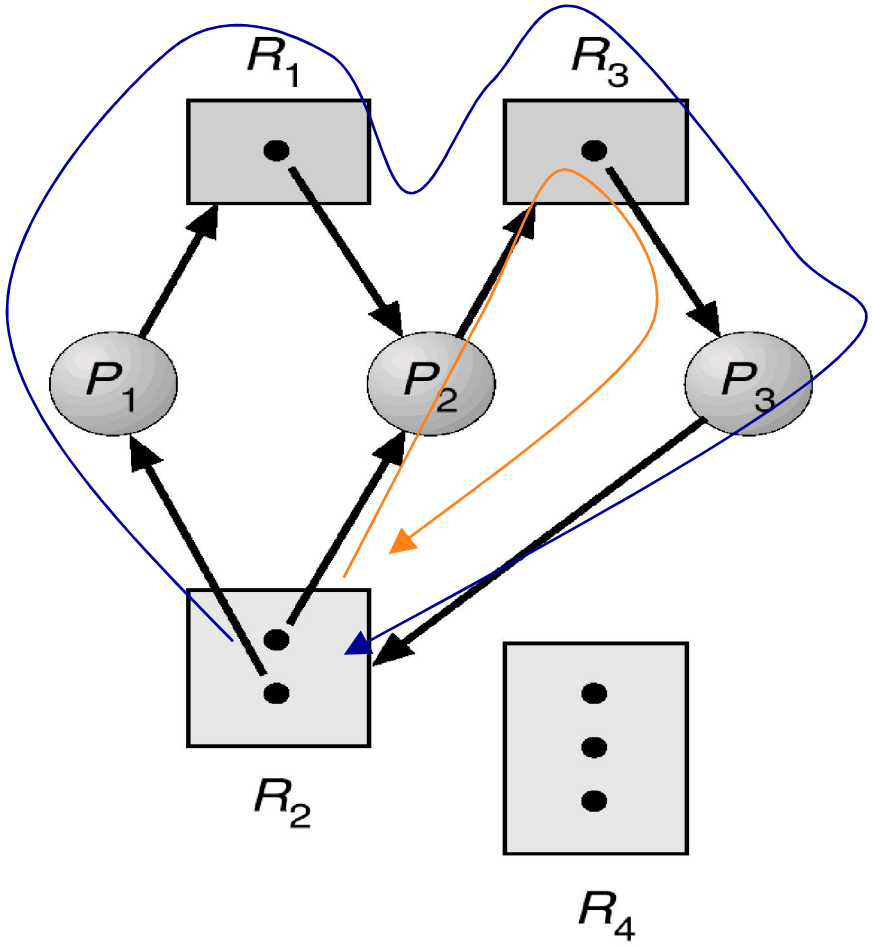


Resource-allocation graph



Can a deadlock happen?

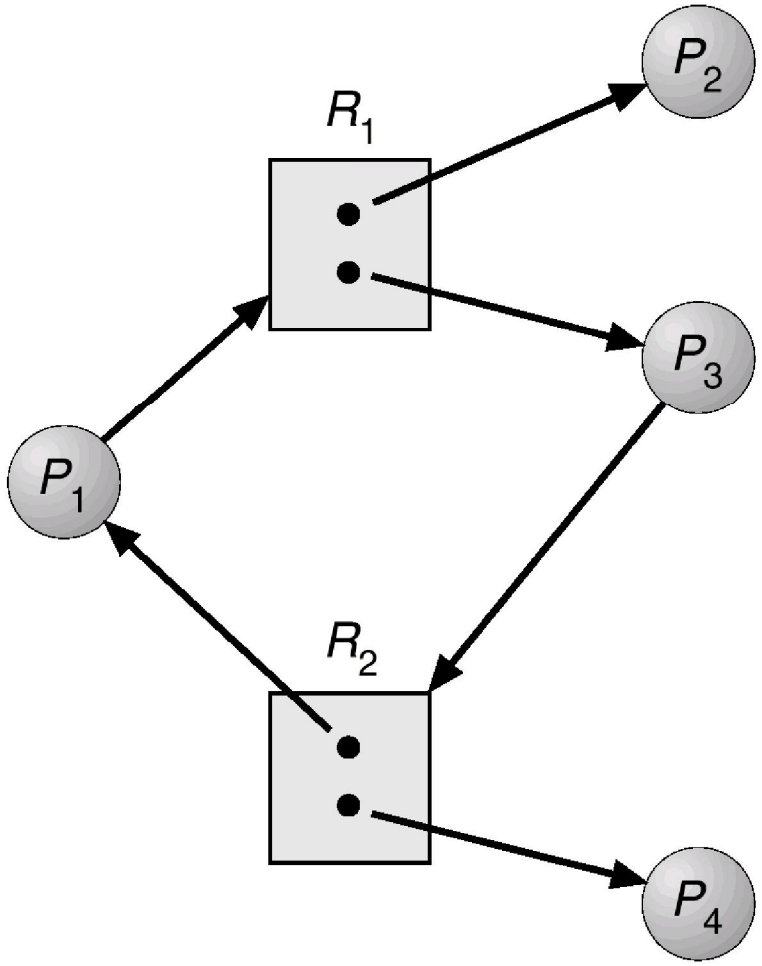
Resource-allocation graph



Can a deadlock happen?

There are two cycles found.

Resource Allocation Graph With A Cycle But No Deadlock



- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Models of Deadlocks

- Depending upon type of resource requests of processes, there are four types of deadlocks and so deadlock models

1. Single unit request model
2. AND request model
3. OR request model
4. AND-OR request model
5. P-our-of-Q request model

Models of Deadlock

- **Single unit Model**
- Outdegree of node in the WFG is 1
- Cycle in WFG represents deadlock

- **AND Model**
- A process/transaction can simultaneously request for multiple resources.
- Remains blocked until it is granted *all* of the requested resources.
- Resource deadlocked processes are modeled by AND
- A **cycle is sufficient** to declare a deadlock provided there is one copy of resources

- **OR Model**
- A process/transaction can simultaneously request for multiple resources.
- Remains blocked till *any one* of the requested resource is granted.
- Communication deadlocks are modeled by OR
- A **cycle** is a **necessary** condition
- A **knot** is a **sufficient** condition

- **AND-OR model**
- Process requests are specified by using a predicate like R1 AND (R2 OR R3)
- Knot is a sufficient condition for deadlock
- **P-out-of-Q model**
- A process simultaneously requests Q resources and remains blocked until P is granted
- When P=Q, it becomes AND model
- When P=1, it becomes OR model
- Knot is a sufficient condition for deadlock

Deadlock Handling Strategies

- There are three strategies for handling deadlocks, viz., deadlock prevention, deadlock avoidance, and deadlock detection.

1. **Deadlock Prevention:**

   - Resources are granted to requesting process in such a way that granting a request for a resource never leads a deadlock

   - Can be achieved by
     - either having process acquire all the needed resources simultaneously before it begins execution,
     - or by preempting a process that holds the needed resource

Deadlock Handling Strategies

- Many drawbacks
  - Decreases the system concurrency
  - Set of processes can become deadlocked in resource acquiring phase
  - In many systems, future resource requirements are unpredictable

Deadlock Handling Strategies

2. **Deadlock Avoidance:**
   - Before allocation, check for possible deadlocks.
   - A resource is granted to a process if the resulting global system state is safe.
   - A state is safe if there exists at least one sequence of execution for all processes such that all of them can run to completion

   - Difficult in DS because,
     - It needs global state info in each site requiring huge storage and communication requirements
     - Due to the large number of processes and resources, it will be computationally expensive to check for a safe state

Deadlock Handling Strategies

- **Deadlock Detection:**
  - Resources are granted to requesting processes without any check
  - Periodically the status of resource allocation and pending requests is examined
  - Find cycles in WFG

  - Two favorable conditions:
    - Once a cycle is formed in the WFG, it remains until it is detected and broken
    - Cycle detection can proceed concurrently with the normal activities of a system

Issues in Deadlock detection and resolution

- Two basic issues :
  - Detection of existing deadlocks
  - Resolution of detected deadlocks

- Detection of deadlocks involves two issues:
  - Maintenance of the WFG
  - Search of the WFG for the presence of cycles(or knots)

- Correct deadlock detection algorithm must satisfy the following two conditions:
  1. Progress—No undetected deadlocks
     - Algorithm must detect all existing deadlocks in finite time
  2. Safety—No false deadlocks
     - Algorithm should not report deadlocks which are non-existent

Issues in Deadlock detection and resolution

- Resolution
  - Breaking existing dependencies in the system WFG
  - Roll back one or more processes and assign their resources to other deadlocked processes

Control organizations for Distributed Deadlocks

1. Centralized Control
   - A *control site (designated site)* constructs global wait-for graphs (WFGs) and checks for directed cycles.
   - WFG can be maintained continuously (or) built on-demand by requesting WFGs from individual sites.
   - Simple and easy to implement
   - Have a single point of failure
     - Communication links near the control sites are likely to be congested

Control organizations for Distributed Deadlocks

2. Distributed Control
   - Responsibility for detecting a global deadlock is shared equally among all sites
   - The global state graph is spread over many sites and several sites participate in the detection of a global cycle.
   - Deadlock detection is initiated only when the waiting process is suspected to be a part of deadlock cycle
   - Any site can initiate the deadlock detection process

   - Not vulnerable to single point failure
   - Difficult to design due to the lack of globally shared memory
   - Several sites may initiate detection for the same deadlock

3. Hierarchical control

- Sites are arranged in hierarchical fashion
- Site detects deadlocks only in its descendant sites
- Best of both centralized and distributed control org
- No single point of failure like centralized

Centralized Algorithms

- **The completely centralized algorithm**
  - Control site maintains the WFG of the entire system and checks it for the existence of the deadlock
  - *Request resource* and *release resource* messages sent by each site to the control site
  - Simple and easy to implement

- Problem:
  - Highly inefficient
  - Large delays in user response, large communication overhead, congestion of communication links near the control site
  - Poor reliability

Centralized Algorithms

- Solution:
  - Can be mitigated partially by having each site maintain its resource status (WFG) locally and by having each site send its resource status to a designated site periodically for construction of the global WFG and the detection of deadlocks.
  - Due to inherent communication delays and the lack of synchronized clocks, the designated site may get inconsistent view of the system and detect false deadlocks

  - Example:
  - R1 and R2 are stored at sites S1 and S2
  - Transactions T1 and T2 are started at sites S3 and S4 respectively:

Centralized Algorithms

T1(S3)	T2(S4)
lock R1	lock R1
unlock R1	unlock R1
lock R2	lock R2
unlock R2	unlock R2

- False deadlock: T1->T2->T1

Centralized Algorithms

- **Ho-Ramamoorthy 2-phase Algorithm**
  - Each site maintains a status table of all local processes initiated at that site: includes all resources locked & all resources being waited on.
  - Controller requests (periodically) the status table from each site.
  - Controller then constructs WFG from these tables, searches for cycle(s).
  - If no cycles, no deadlocks.

Centralized Algorithms

- Otherwise, (cycle exists): Request for state tables again.
- The designated site construct WFG *only* those transactions which are common to both reports to get consistent view of the system(consistent means correct state of the system)
- If the same cycle is detected again, system is in deadlock.
- Later proved: cycles in 2 consecutive reports *need not* result in a deadlock. Hence, this algorithm detects false deadlocks

Centralized Algorithms

- **Ho-Ramamoorthy 1-phase Algorithm**
  - Each site maintains 2 status tables: *resource status* table (for all local resources) and *process status* table (for all local processes).
  - Controller periodically collects these tables from each site.
  - Constructs a WFG from transactions common to both the tables.
  - No cycle, no deadlocks.
  - A cycle means a deadlock.

Centralized Algorithms

- Does not detect false deadlocks:

  E.g.

  - Resource table at S1: R1 is waited upon by P2 (R1← P2)
  - Process table at S2: P2 is waiting for R1 (P2→R1)
  - then, the edge P2→R1 reflects current system state.

  - This is faster
  - Requires less messages compared to 2 phase
  - Requires more storage

Distributed Algorithms

- All sites collectively cooperate to detect a cycle in the state graph that is likely to be distributed over several sites of the system

- Initiated when,
  - A process is forced to wait and,
  - it can be initiated either by the local site of the process or by the site where the process waits

Classification of Distributed Algorithms for Deadlock detection

- E. Knapp(1987) has classified the distributed deadlock detection algorithms into following:
- **Path-pushing:** resource dependency information disseminated through designated paths (in the graph).
- **Edge-chasing:** special messages or probes circulated along edges of WFG. Deadlock exists if the probe is received back by the initiator (single returned probe indicates a cycle).
- **Diffusion computation:** queries on status sent to process in WFG (all queries returned indicates a cycle).
- **Global state detection:** get a snapshot of the distributed system.

Edge-Chasing Algorithm

- **Chandy-Misra-Haas's Algorithm for AND model (Resource Model)**
  - If a process makes a request for a resource which fails or times out, the process generates a *probe message* and sends it to each of the processes holding one or more of its requested resources.
  - Each probe message contains the following information:
    - the *id* of the process that is blocked *(the one that initiates the probe message)*;
    - the *id* of the process which is sending this particular version of the probe message; and
    - the *id* of the process that should receive this probe message.
  - A probe(i, j, k) is used by a deadlock detection process P_i. This probe is sent by the home site of P_j to home site of P_k.
  - This probe message is circulated via the edges of the graph. Probe returning to P_i implies deadlock detection.

Edge-Chasing Algorithm

- **Chandy-Misra-Haas's Algorithm for AND model (Resource Model)**
  - Terms used:
    - P_j is *dependent* on P_k, if a sequence of P_j, P_{i1},.., P_{im}, P_k exists, such that each process except P_k in the scquence is blocked and each process , except the first one (P_j), holds a resource for which the previous process in the secquence is waiting.
    - P_j is *locally dependent* on P_k, if P_j is dependent upon P_k and both the processes are at same site.
    - Each process maintains a boolean array $dependent_i$: $dependent_i(j)$ is true if Pi knows that P_j is dependent on it. (initially $dependent_i(j)$ set to false for all i & j).

Chandy-Misra-Haas's Algorithm

Sending the probe:

if P_i is locally dependent on itself then deadlock.

else for all P_j and P_k such that

(a) P_i is locally dependent upon P_j, and

(b) P_j is waiting on P_k, and

(c) P_j and P_k are on different sites, send probe(i,j,k) to the home site of P_k.

Receiving the probe:

if (d) P_k is blocked, and

(e) $dependent_k(i)$ is false, and

(f) P_k has not replied to all requests of P_j,

then begin

$dependent_k(i) := true;$

if k = i

then declare that P_i is deadlocked

…….

Chandy-Misra-Haas's Algorithm

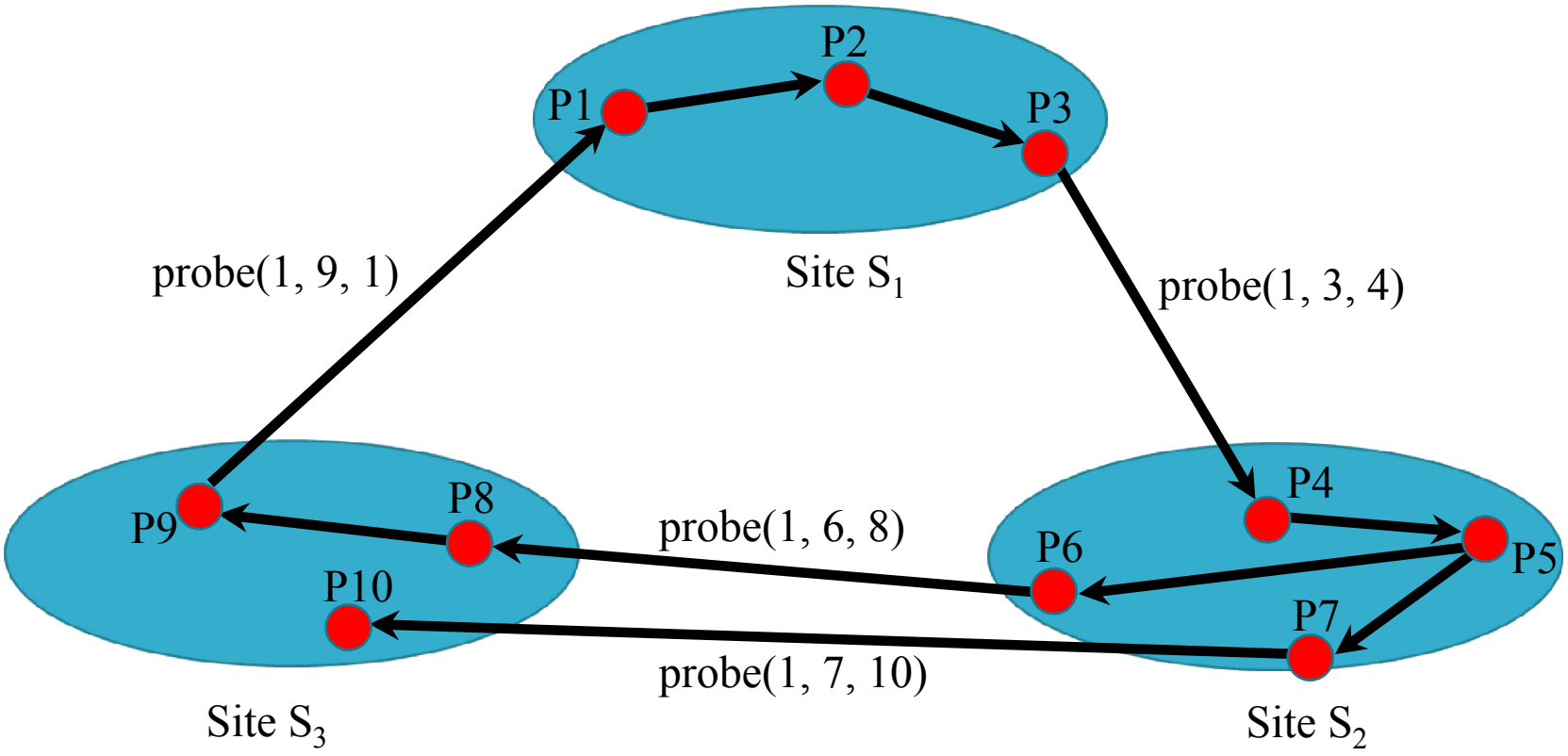
Receiving the probe (contd…):

else for all P_m and P_n such that
(a') P_k is locally dependent upon P_m, and
(b') P_m is waiting on P_n, and
(c') P_m and P_n are on different sites, send probe(i,m,n) to the home site of P_n.
end.

When a process receives a probe message, it checks to see if it is also waiting for resources. If not, it is currently using the needed resource and will eventually finish and release the resource. If it is waiting for resources, it passes on the probe message to all processes it knows to be holding resources it has itself requested. The process first modifies the probe message, changing the sender and receiver ids.

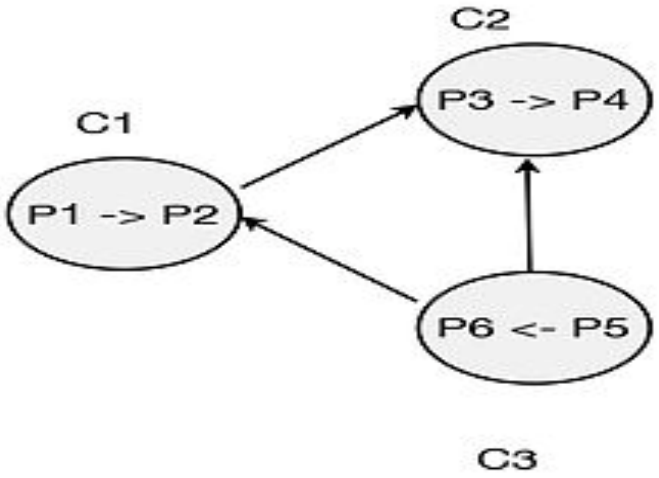
If a process receives a probe message that it recognizes as having initiated, it knows there is a cycle in the system and thus, **deadlock**.

Example


P2
P1
P3
probe(1, 9, 1)
Site S_1
probe(1, 3, 4)
P9
P8
P4
probe(1, 6, 8)
P6
P5
P10
P7
probe(1, 7, 10)
Site S_3
Site S_2

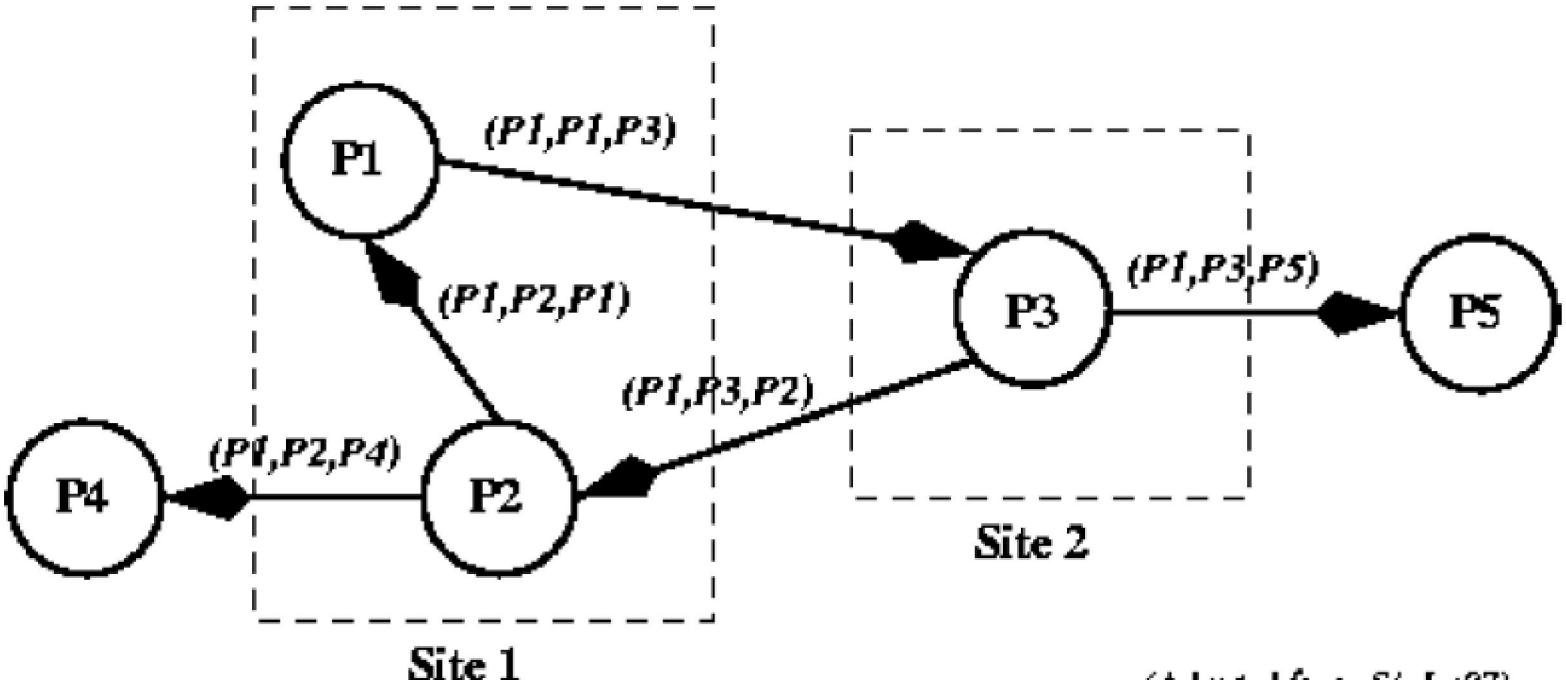
Example

- In the shown example, P_1 initiates the deadlock detection. C_1sends the probe saying P_2 depends on P_3. Once the message is received byC_2, it checks whether P_3 is idle. P_3 is idle because it is locally dependent on P_2 and updates $dependent_3(2)$ to True.
- Same as above, C_2 sends probe to C_3 and C_3 sends probe to C_1. At C_1 P_1 is idle so it update $dependent_1(1)$ to true. Therefore it can be declared a deadlock has occurred.

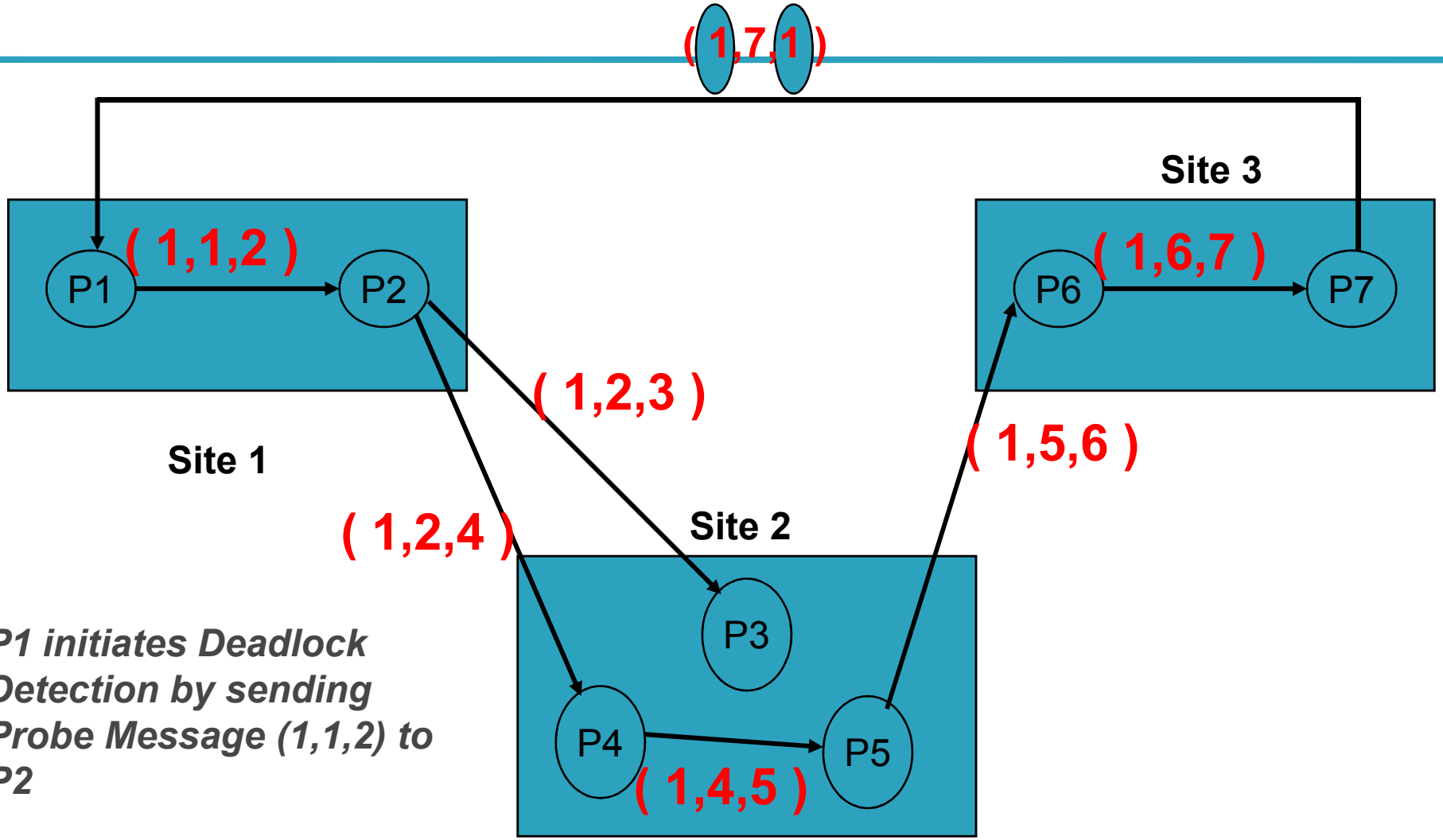


Example

- **P1** initiates the probe message, so that all the messages shown have **P1** as the initiator. When the probe message is received by process **P3**, it modifies it and sends it to two more processes. Eventually, the probe message returns to process **P1**. **Deadlock!**



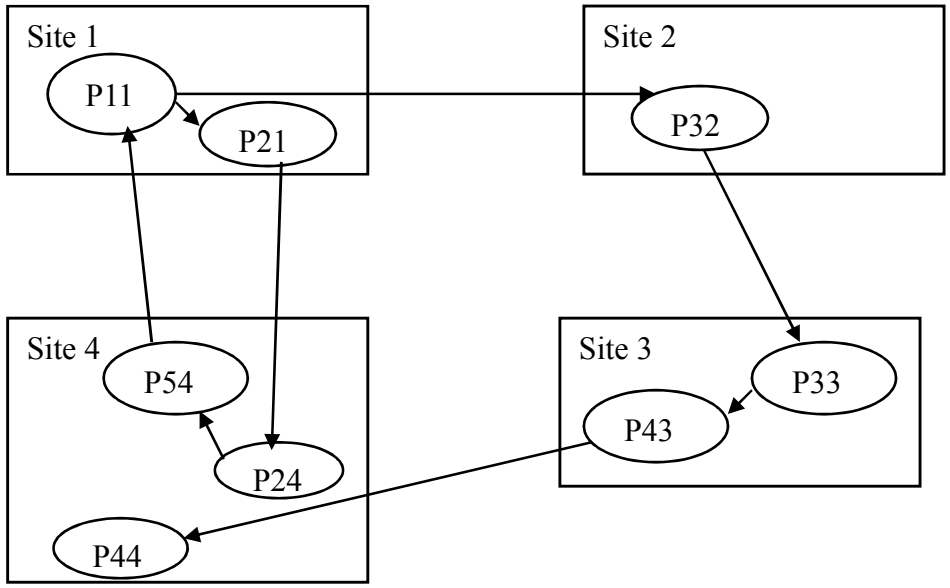
Example



P1 initiates Deadlock Detection by sending Probe Message (1,1,2) to P2

Example

- If process P11 initiates Chandy-Misra edge chasing algorithm then find out whether deadlock is occurred in the system or not? Justify your answer.



Chandy-Misra-Hass Deadlock Detection Algorithm

- The advantages of this algorithm include the following:
  1. It is easy to implement.
  2. Each probe message is of fixed length.
  3. There is very little computation.
  4. There is very little overhead.
  5. There is no need to construct a graph, nor to pass graph information to other sites.
  6. This algorithm does not find false (phantom) deadlock.
  7. There is no need for special data structures.
- Performance analysis

- At most m(n-1)/2 messages to detect a deadlock having m processes and spans over n sites
- Delay in detecting the deadlock is O(n)

Other edge-chasing algorithms: The Mitchell-Kerritt, Sinha-Niranjan Algorithms

A Diffusion Computation Based Algorithm

Algorithm for OR model (Communication Model)

- Deadlock Detection computation is diffused through WFG of the system
- Process can be in two states:
  - Active state- a process is executing.
  - Blocked state – a process is waiting to acquire a resource.
- A blocked process may start a diffusion- by sending query message to all the processes from whom it is waiting to receive a message.
- Messages takes two forms
  - Query message (i,j,k)
    - *i* = initiator of check
    - *j* = immediate sender
    - k = immediate recipient
  - Reply message (i,k,j)

A Diffusion Computation Based Algorithm

- Dependent set of process:
  - Set of processes from whom the process is waiting to receive message
- Engaging query:
  - The first query message received for the deadlock detection initiated by Pi.
- $wait_k(i)$:
  - It is a local boolean variable at process P_k indicates that it has been continuously blocked since it received the last engaging query from process Pi

A Diffusion Computation Based Algorithm

If an active process receives a query or reply message, it discards it. When a blocked process P_k receives a query (i,j,k) message, it takes the following actions:

- If this is the first query message receives by P_k for deadlock detection initiated by P_i (*called the engaging query*), then it propagates the query to all the processes in its dependent set and sets a local variable num_k *(i) to the number of query messages sent.*
- If this is not an engaging query, then P_k returns a reply message to it immediately, provided P_k has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.

An initiator detects a deadlock when it receives reply messages to all the query messages it had sent out.

Diffusion-based Algorithm

Initiation by a blocked process P_i:

send query(i,i,j) to all processes P_j in the dependent set DS_i of P_i;
num(i) := |DSi|; $wait_i(i)$:= true;

Blocked process P_k receiving query(i,j,k):

if
this is *engaging* query for process P_k /* first query from P_i */
then
send query(i,k,m) to all P_m in DS_k;
$num_k(i)$:= $|DS_k|$; $wait_k(i)$:= true;
else if
$wait_k(i)$
then
send a reply(i,k,j) to P_j.

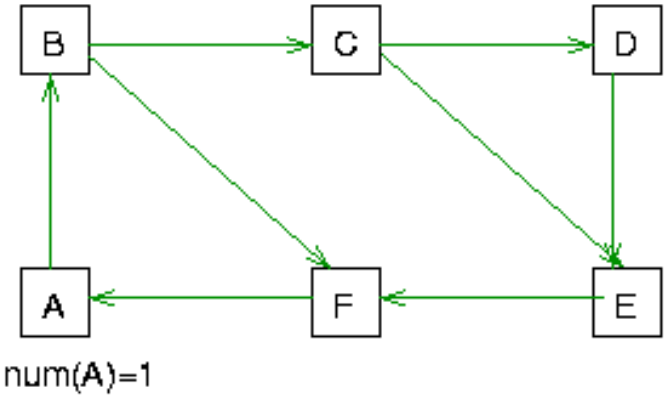
Diffusion-based Algorithm

Process P_k receiving reply(i,j,k)

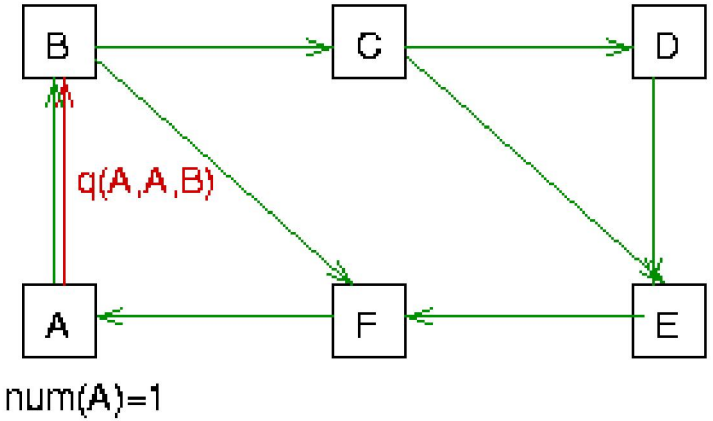
```
 if
        wait_k(i)
then
        num_k(i) := num_k(i) - 1;
        if
                num_k(i) = 0 then
               if
                        i = k then declare a deadlock.
              else
                        send reply(i, k, m) to P_m, which sent the engaging query.
```

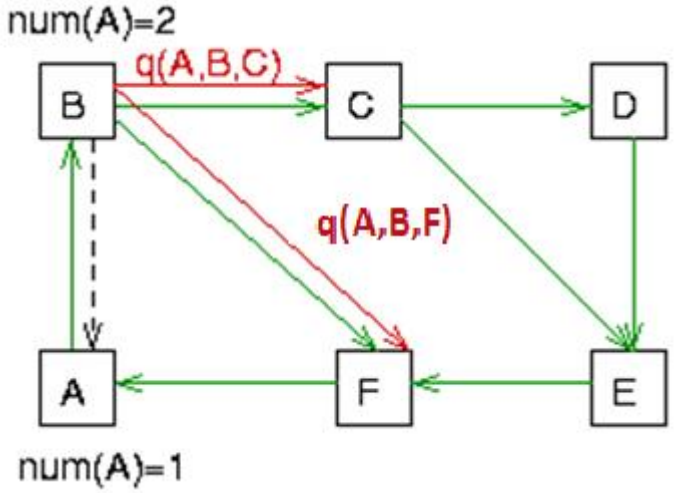
On receipt of *query*(*i*,*j*,*k*) by m

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

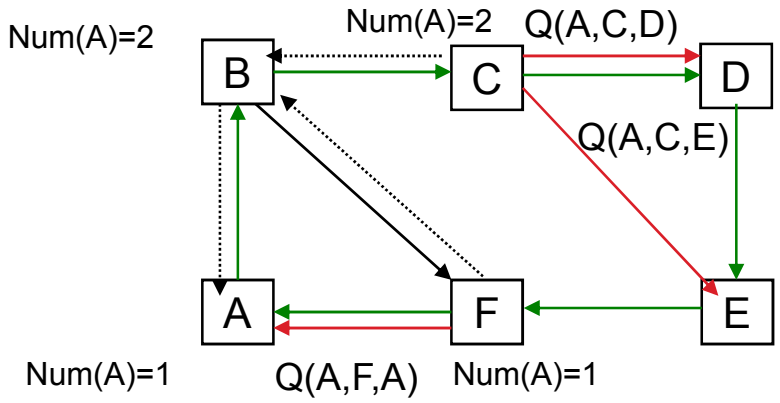


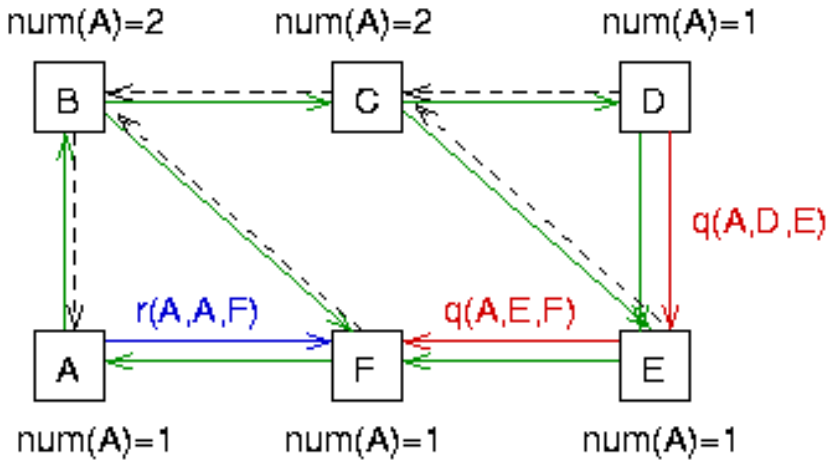
On receipt of *query*(*i*,*j*,*k*) by *m*





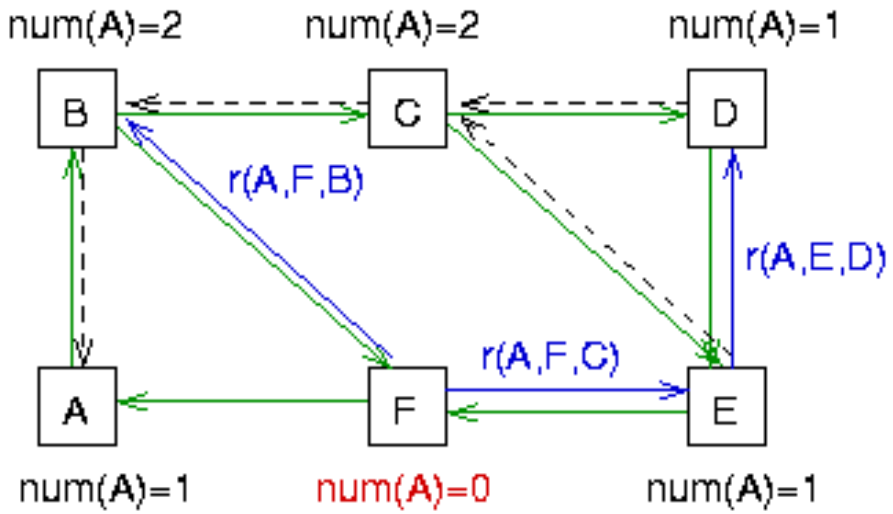
On receipt of *query*(*i*,*j*,*k*) by *m*



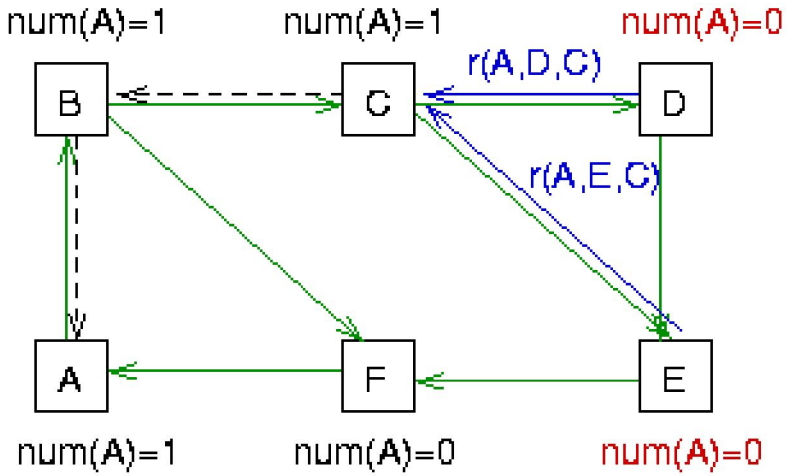


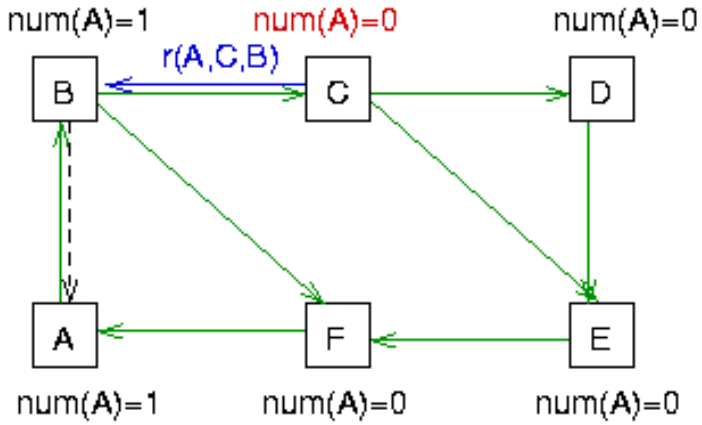
On receipt of *reply*(*i*,*j*,*k*) by *k*

- 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*

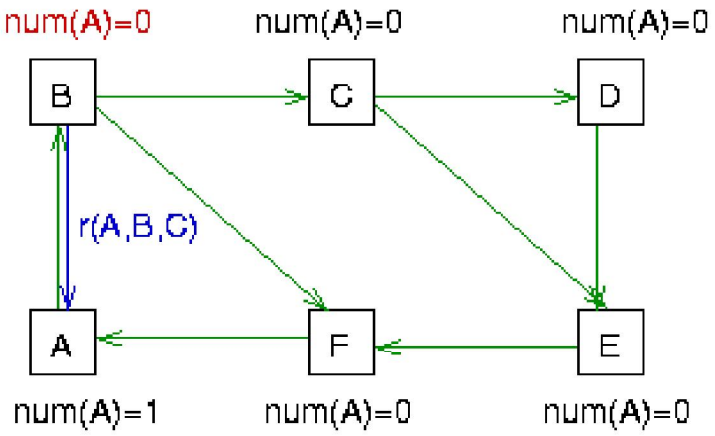


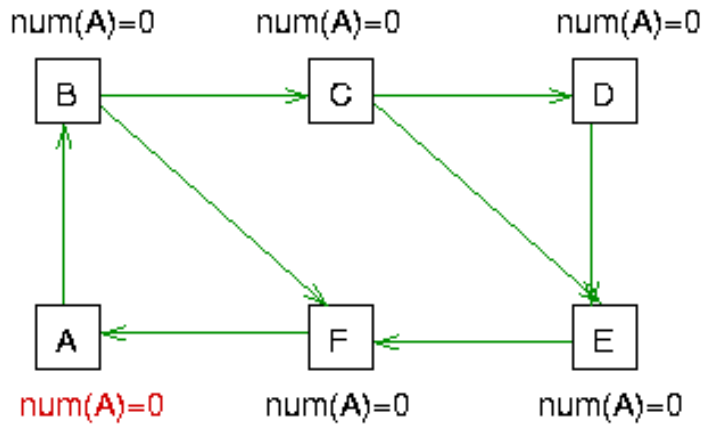
On receipt of *reply*(*i*,*j*,*k*) by *k*





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

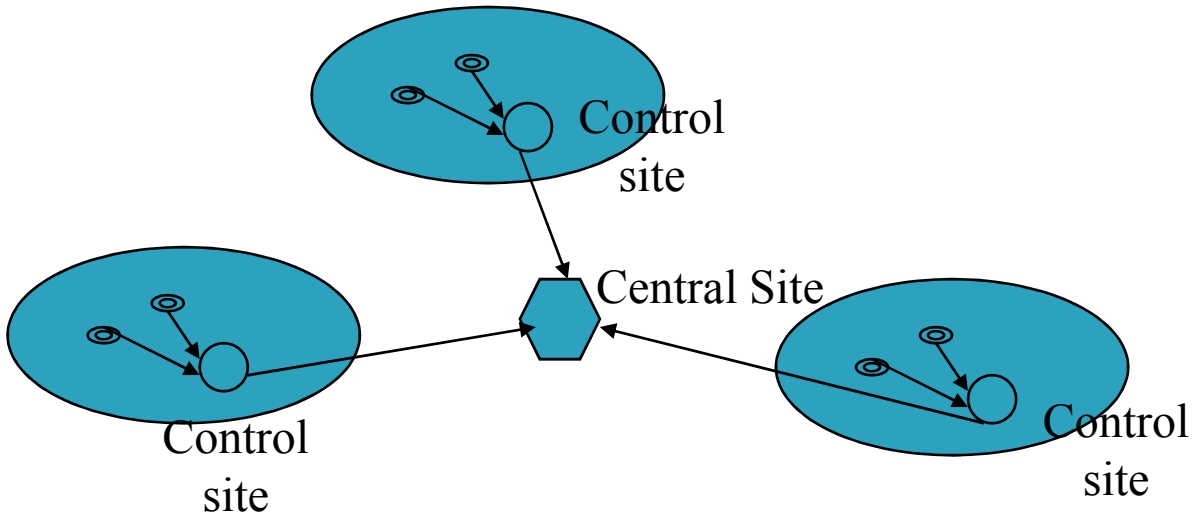




Knot detected !

Hierarchical Deadlock Detection

- Follows Ho-Ramamoorthy's 1-phase algorithm.
- More than 1 control site organized in hierarchical manner.
- Each control site applies 1-phase algorithm to detect (intracluster) deadlocks.
- Central site collects info from control sites, applies 1-phase algorithm to detect intercluster deadlocks.



Persistence & Resolution

- Deadlock persistence:
  - Average time a deadlock exists before it is resolved.
- Implication of persistence:
  - Resources unavailable for this period: affects utilization
  - Processes wait for this period unproductively: affects response time.
- Deadlock resolution:

Deadlock Resolution

- Aborting at least one process/request involved in the deadlock.
- Efficient resolution of deadlock requires knowledge of all processes and resources.
- If every process detects a deadlock and tries to resolve it independently -> highly inefficient ! Several processes might be aborted.
- Priorities for processes/transactions can be useful for resolution.

- Highest priority process initiates and detects deadlock .
- When deadlock is detected, lowest priority process(s) can be aborted to resolve the deadlock.