CES-27 Processamento Distribuído

Deadlock Detection

Prof Juliana Bezerra
Prof Vitor Curtis

Outline

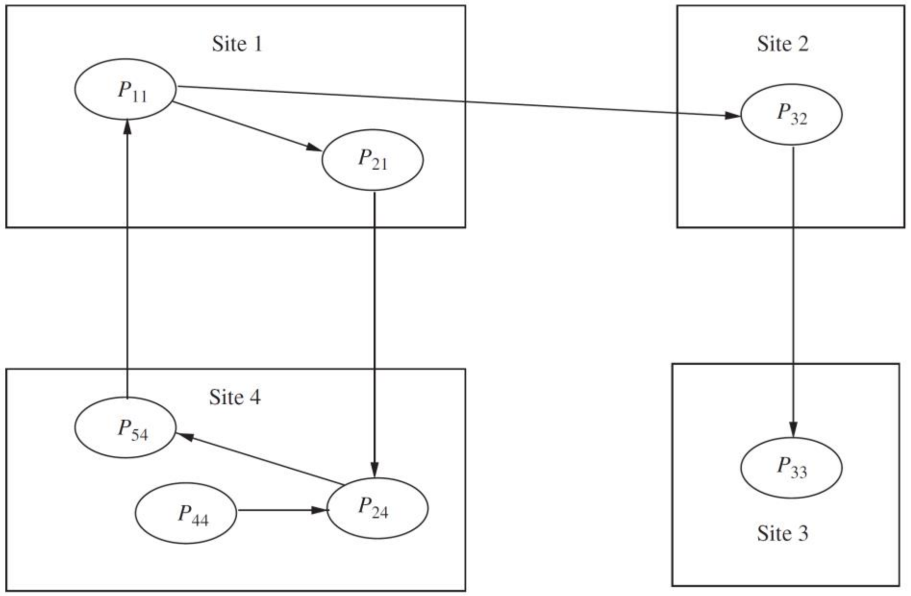
- **Deadlock**
- Prevention
- Detection
- Chandy-Misra-Haas for AND
- Chandy-Misra-Haas for OR

Definition

- **Permanent** blocking of a set of processes that compete for system resources (database records, communication lines, etc.)
  - Deadlock occurs when a set of processes are in a wait state
  - Each process is waiting for a resource held by some other waiting process
  - Therefore, all deadlocks involve conflicting resources needs by two or more processes
- 1971 – Coffman conditions: four necessary conditions for deadlock
  - Mutual Exclusion: processes require exclusive control of its resources (not sharing)
  - Hold and Wait: process holds some resources and waits for other resources, before it can finish the job
  - No Pre-emption: process will not give up a granted resource, voluntarily and by another process, until it is finished with it
    - Preemption: act of temporarily interrupting a task, without its cooperation, and with the intention of resuming it later
  - Circular Wait: each process in the chain holds a resource requested by another

Example of Deadlock

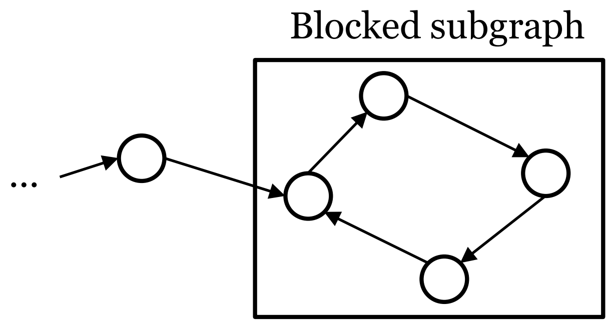
- The state of a distributed system can be modeled by a Wait-For Graph (WFG)
- WFG is a directed graph where nodes are processes and edges P1 → P2 represents that P1 is blocked because it is waiting for P2 to release some resource
  - Attention: P44 and P33 are not waiting for other processes in the graph, so they should not be blocked by definition. However, many references consider that they are blocked and it may lead to misunderstandings

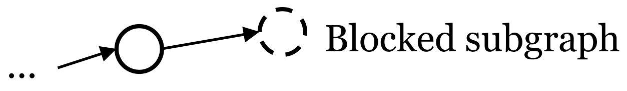


Conventions for WFG

- In this material, consider these conditions:
  - Blocked process (not filled): it is waiting for some resource, i.e., it must have at least one outgoing edge
  - Active process (filled): it has a resource that it will release in the future
  - Blocked subgraph (dashed circle and not filled): it represents a subgraph where all processes are blocked and waiting for the other processes, i.e., a deadlock

○	Blocked process, waiting for other process
●	Active process, it can release resources
◌	Blocked subgraph, i.e., the subgraph is in deadlock

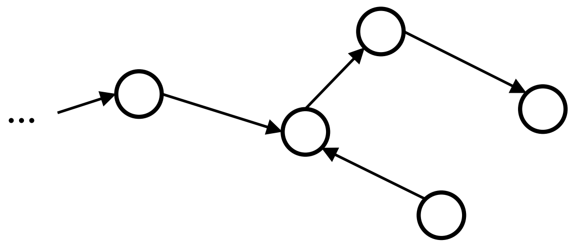




Conventions for WFG

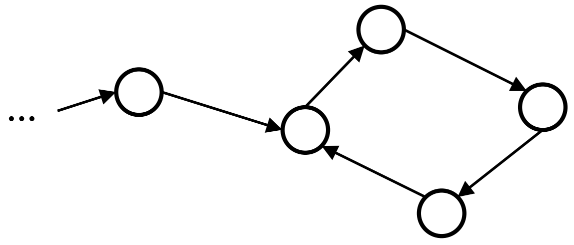
- Result: A WFG graph cannot have a blocked process without at least one outgoing edge
- In other materials, if you see a WFG like the one below (not a valid WFG) with an end (leaf) node without outgoing edge, check the semantic of this node
  - It can be an active process that will release resources, or
  - It can be a deadlocked subgraph like the two examples on the right below
- See additional material, in the end, for a discussion about it

Not a valid WFG

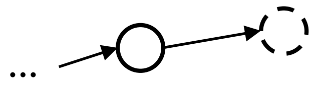


Waiting for what?
Does it have resources to release?

Valid WFG



Valid WFG



Deadlock Handling Strategies

- Dealing with deadlock may be very difficult, especially in distributed systems
- Ignoring deadlock is a surprisingly common approach
  - Yes, the system will crash
- Ostrich Algorithm
  - “stick one's head in the sand and pretend there is no problem”
- Reasonable if deadlocks occur very rarely or cost of prevention is very high
  - Trade off between convenience and correctness
- According to many sources
  - Unix, Linux and Windows take this approach for some of the more complex resource relationships to manage
  - Is it fake? I don’t know ☺ (tell me if you discover it)
  - At least for previous versions, it looks like it is true

Ostrich Approach

Ostrich Approach (São Paulo)

Proud of São Paulo for using the same engineering best practices than Unix, Linux and Windows!

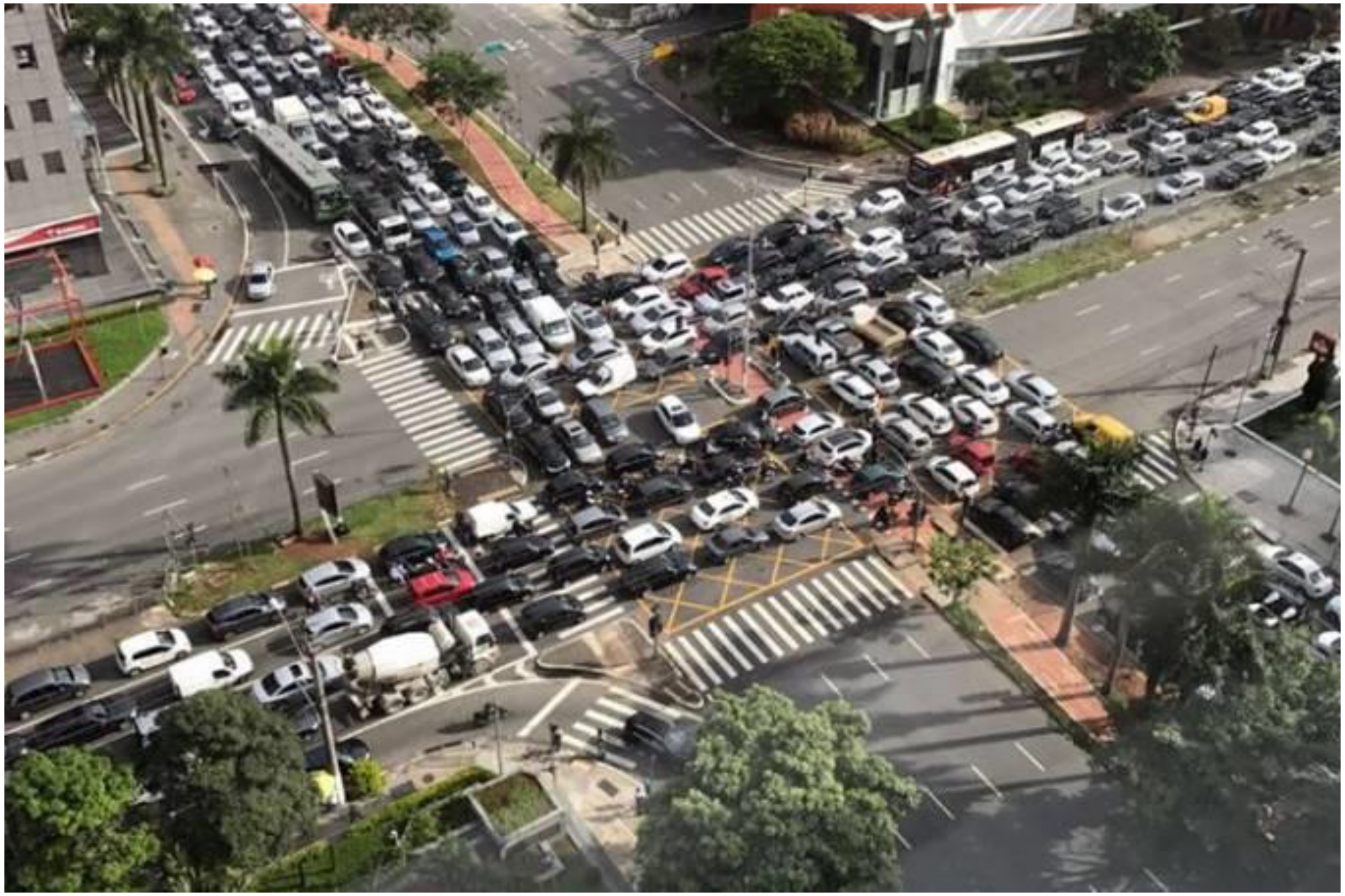
Faria Lima
x
Juscelino Kubitschek
01/15/2008

Fontes: 15/01/2008 [by Silvio Tanaka at Google building]

SP in all slides of O.S. and D.S.

Faria Lima
x
Juscelino Kubitschek
02/02/2017

Note: I can't stand waiting for January/February of 2019



Fontes: https://vejasp.abril.com.br/cidades/foto-de-no-no-transito-da-faria-lima-movimenta-redes-sociais/

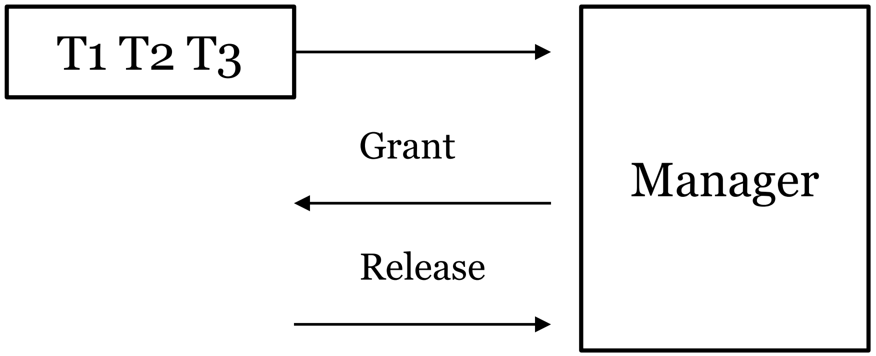
Deadlock Handling Strategies

- Prevention (pessimistic approach): design the system with restrictions so that deadlock is impossible a priori
  - *e.x. compile-time/statically, by design*
  - Conservative but it can limit access to resources and impose so much restrictions on processes (reduced throughput)
- Avoidance: resources are granted only if the global state is safe; a request may be denied
  - *run-time/dynamically, before it happens*
  - Main difference: this approach keeps all necessary conditions for deadlock, but it makes dynamically decisions that are free of deadlocks
  - It requires the global state of the system (done by a mutual exclusion for safe checking) in order to check for safe states
    - Each process has to store the knowledge of all processes, which implies on large capacity of storage and traffic over the network for D.S.
  - Not a very practical solution due to a lot of problems
- Detection & Recovery (optimistic approach): let the deadlock occur, detect it, then recover it
  - *run-time/dynamically, after it happens*
  - Usual approach for D.S.

Fontes: Deadlock [Kshemkalyani, Singhal]

Deadlock Avoidance

- Centralized deadlock avoidance
  - Central unit knows everything and control all processes
- Banker's Algorithm
  - We will not see it



"Years" in idle state or multithreading idea

Outline

- Deadlock
- **Prevention**
- Detection
- Chandy-Misra-Haas for AND
- Chandy-Misra-Haas for OR

Deadlock Prevention

- One of the four necessary conditions (Coffman) is broken
- Mutual Exclusion
  - Systems with only simultaneously shared resources (read-only) is generally not possible or very restricted
  - Not a good approach
- Hold and Wait
  - Disallow waiting while holding resources, i.e., a process can only request a resource if it does not hold any others;
  - Non-waiting requests, i.e., a request that cannot be satisfied immediately will fail;
  - Or, request all required resources at once before execution begins, i.e., it holds nothing while waiting and takes all or nothing

Deadlock Prevention

- One of the four necessary conditions (Coffman) is broken
- No Preemption
  - If a process is denied access to a resource, then it must release all resources it currently held;
  - Preempted resources (taken way) from a process are added to its waiting list;
  - Process restarts only when can regain all resources it requested
  - This approach may imply on rollbacks or return errors
- Circular Wait
  - The classic approach

First Prevention Solution

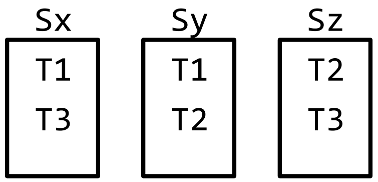
- Example of resource ordering: assign a total ordering on all the resources, then allow a process to request resources only in increasing order (Direct Acyclic Graph)
  - If a process requested the resource type A, then it may subsequently request only those following A in the ordering
- Requests
  - P1(a), P1(b), P1(f)
  - P2(b), P2(d), P2(e)
  - P3(d), P3(e)
- If P1 has just *a*, P2 cannot get *b*
  - P1 can request *b*
- If P1 has *a*, *b*, and *f*, then P3 can get *d*
  - P1 cannot request *c*, *d* and *e*
- Very restrictive solution

Prevention Solution

- Example of central prevention by avoiding circular wait
- Tasks (Processes/Transactions) declare the resources they need
    - T1: x, y
    - T2: y, z
    - T3: z, x
- In this case, we will use the relation ⇒ (potentially block)
    - If A ⇒ B, then A potentially block B
- Tasks arrive in this order: T1, T2, T3
- Possible resource scheduling
    - O(1,x), O(2,y), O(3,z), i.e., x to T1, y to T2, z to T3
    - O(1,x): T1 ⇒ T3
    - O(1,x), O(2,y): T2 ⇒ T1 ⇒ T3
    - O(1,x), O(2,y), O(3,z): T3 ⇒ T2 ⇒ T1 ⇒ T3 (cycle)
- Dynamically prevention of deadlock
    - Besides z is available, O(3,z) is blocked because it creates a cycle
- This approach usually congests the manager if the distributed system is large

Prevention Solution

- In a distributed solution, each site Si has some resources and knows just the dependences of them
  - In this example the site Sa has the resource *a*
- Requests
  - T1: x, y
  - T2: y, z
  - T3: z, x
- Resource scheduling
  - O(1,x), O(2,y), O(3,z), i.e., x to T1, y to T2, x to T3
  - O(1,x) at Sx: T1 ⇒ T3
  - O(2,y) at Sy: T2 ⇒ T1
  - O(3,z) at Sz: T3 ⇒ T2 (it cannot prevent deadlock)
- Consider a total ordering of the requests (ex. using logical clock)
  - It means that there are no equal timestamps
  - TSi is the timestamp of Ti
  - Ti can be executed before Tj iff (TSi < TSj)



Prevention Solution

- Distributed solution with total ordering by logical clocks
- Requests
  - T1: x, y
  - T2: y, z
  - T3: z, x
- Resource scheduling
  - O(1,x), O(2,y), O(3,z), i.e., x to T1, y to T2, x to T3
  - Timestamps: TS1<TS2<TS3 imposes T1 before T2 before T3
  - O(1,x) at Sx: TS1<TS3 and T1 ⇒ T3
  - O(2,y) at Sy: TS1<TS2 but T2 ⇒ T1 (potential cycle)
  - O(3,z) at Sz: TS2<TS3 but T3 ⇒ T2 (potential cycle)
- O(2,y) is not allowed because T2 ⇒ T1, and this would create a potential cycle once TS1<TS2
- Same for O(3,z)

Prevention Solution

- The previous approach is very restrictive, but we may "improve" it
- Another approach is not to block the tasks, but rollback them when a conflict occur
    - In this case, just the timestamps are sufficient
    - No resource declaration is used
- Two main approaches of prevention
    - Wait-Die(no preemption): An older task (higher priority) is allowed to wait, and a younger is aborted
    - Wound-Wait(preemption): An older task causes the preemption (abort) of a younger one, and a younger is allowed to wait
- New timestamps may cause starvation

```
Wait-Die (no preemption): Ti→Tj(resource)

if TSi < TSj
   block Ti // wait
else
   abort Ti // die
   // restart Ti (same timestamp)
```

```
Wound-Wait (preemption): Ti→Tj(resource)

if TSi < TSj
   abort Tj // wound
   // restart Tj (same timestamp)
else
   block Ti // wait
```

Outline

- Deadlock
- Prevention
- **Detection**
- Chandy-Misra-Haas for AND
- Chandy-Misra-Haas for OR

Detection Solutions

- 1987 - Knapp's Classification: distributed deadlock detection algorithms can be divided into four classes
  - Path-pushing
    - Build a global WFG for each site of the distributed system
    - During a detection, each site sends its local WFG to all the neighboring sites; After a update, the new WFG is then passed along to other sites until one site has a sufficiently complete picture of the global state to announce or not a deadlock
  - Edge-chasing
    - It detects cycles in a distributed system by propagating special messages called *probes* along the edges of the WFG
    - The formation of a cycle is detected by a site if it receives the matching *probe* sent by it previously (probe messages are usually small)

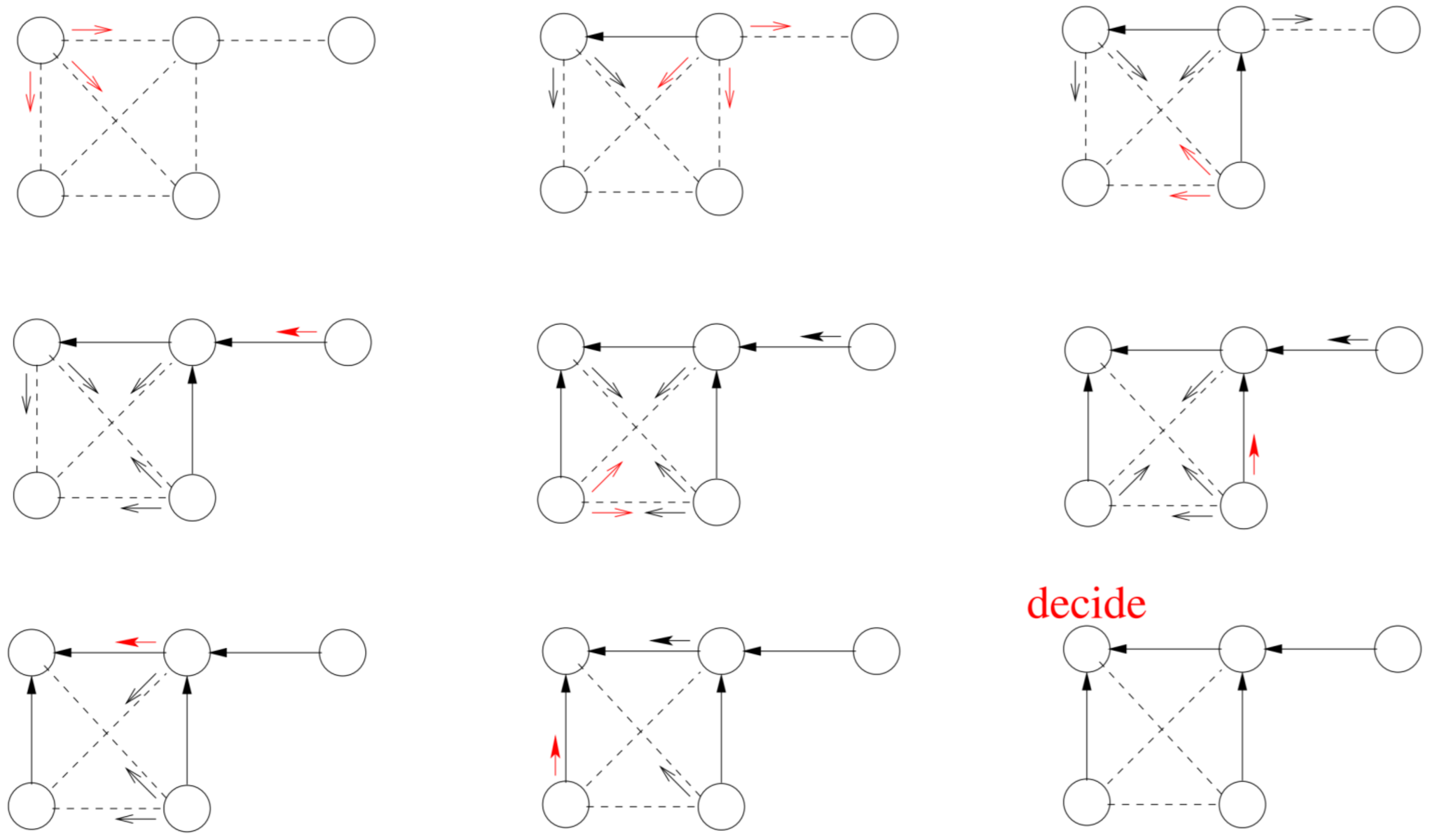
Fontes: Deadlock [Kshemkalyani, Singhal]

Detection Solutions

- 1987 - Knapp's Classification: distributed deadlock detection algorithms can be divided into four classes
  - Diffusing computation-based
    - It detects deadlock by sending echo messages along the WFG, i.e., computation is diffused through the WFG
    - Active processes ignore the diffusion (example in the next slide)
  - Global state detection
    - A deadlock in a distributed system is a stable property that holds at any moment
    - This approach takes a snapshot of the distributed system and check for the deadlock

Fontes: Deadlock [Kshemkalyani, Singhal]

Echo Messages

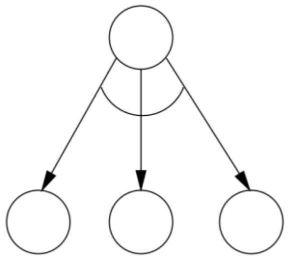


Fontes: Deadlock [Kshemkalyani, Singhal]

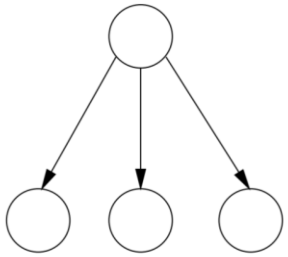
Models of Deadlock

- AND deadlock (resource)
  - Blocked until all resources requested by the process is granted, i.e., it can continue the computation just with all resources (e.g. transactions)
- OR deadlock (communication)
  - Blocked untill any of the resources requested by the process is granted, i.e., it can continue the computation with just one resource (e.g. waiting for work messages)
- P-out-of-Q deadlock
  - General case, blocked until P resources of Q requests is granted
  - It is the OR model when P=1, and AND model when P=Q

P-out-of-Q WFG representation



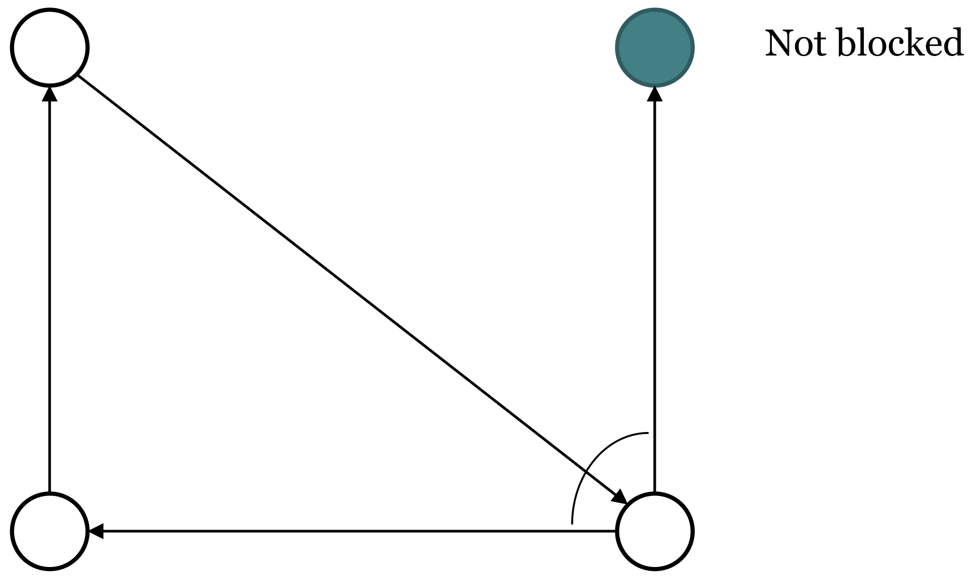
AND (3–out–of–3) request



OR (1–out–of–3) request

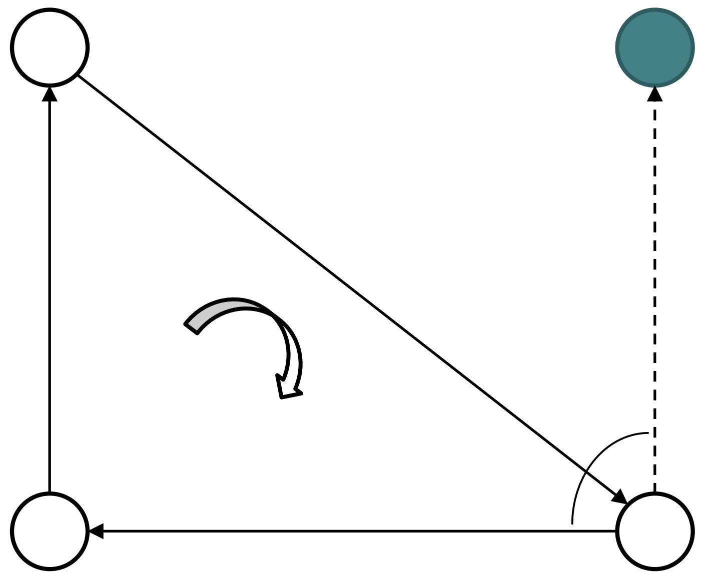
Central Detection with WFG Reduction

Static analysis of an AND-deadlock by reduction of a WFG snapshot



Central Detection with WFG Reduction

Static analysis of an AND-deadlock by reduction of a WFG snapshot



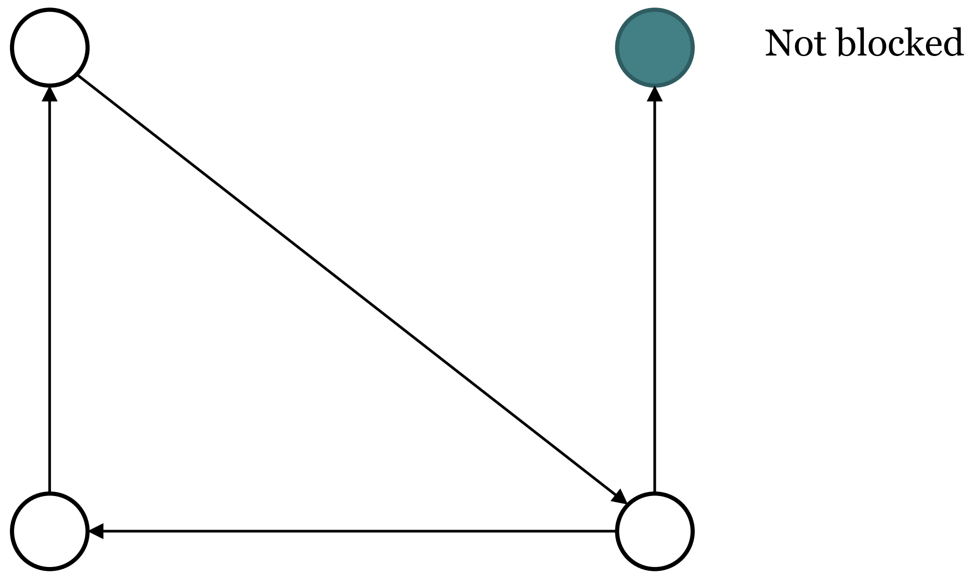
Not blocked

Still blocked because it needs all requested resources

A cycle in an AND model results in deadlock

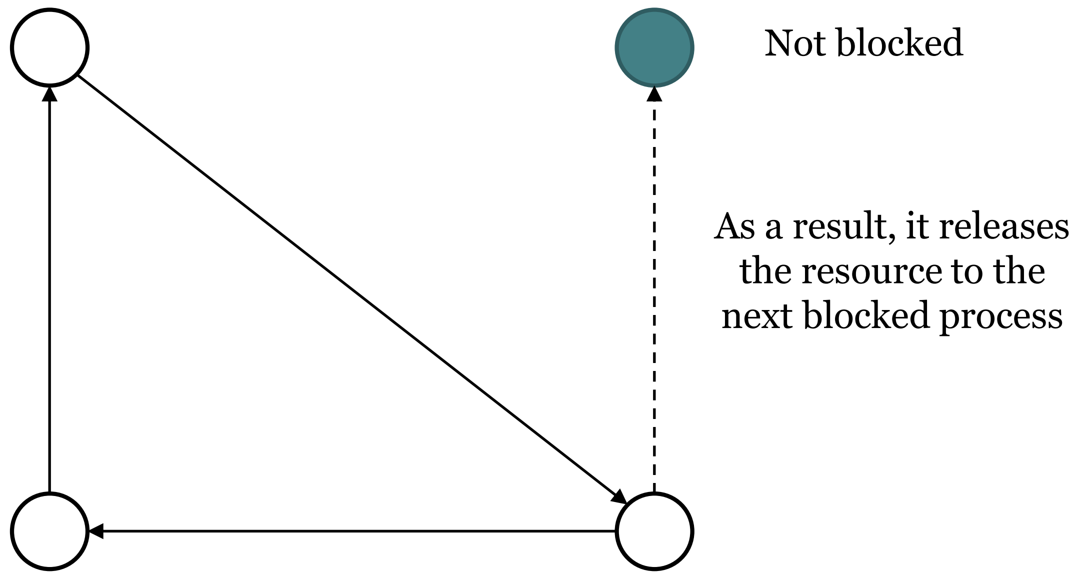
Central Detection with WFG Reduction

Static analysis of an OR-deadlock by reduction of a WFG snapshot



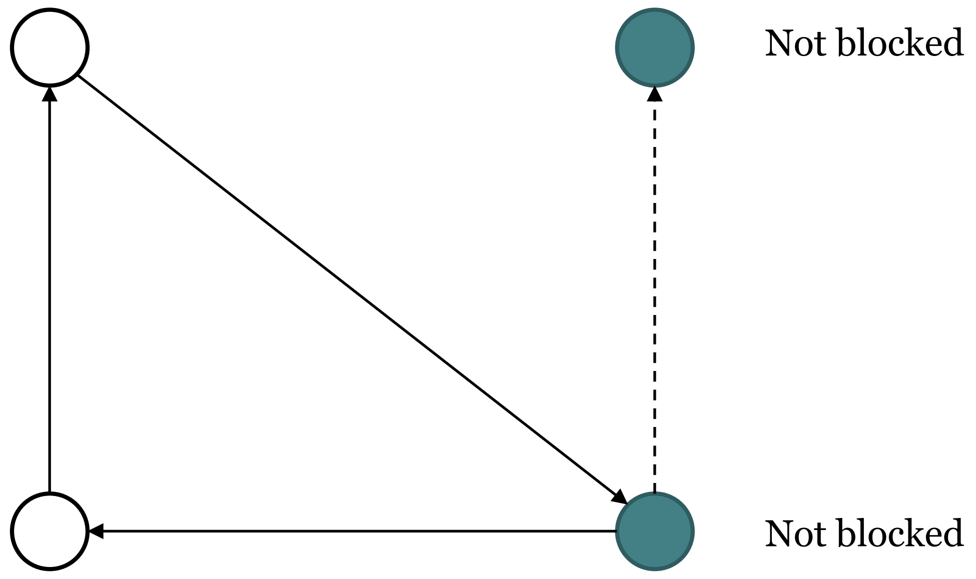
Central Detection with WFG Reduction

Static analysis of an OR-deadlock by reduction of a WFG snapshot


Not blocked
As a result, it releases the resource to the next blocked process

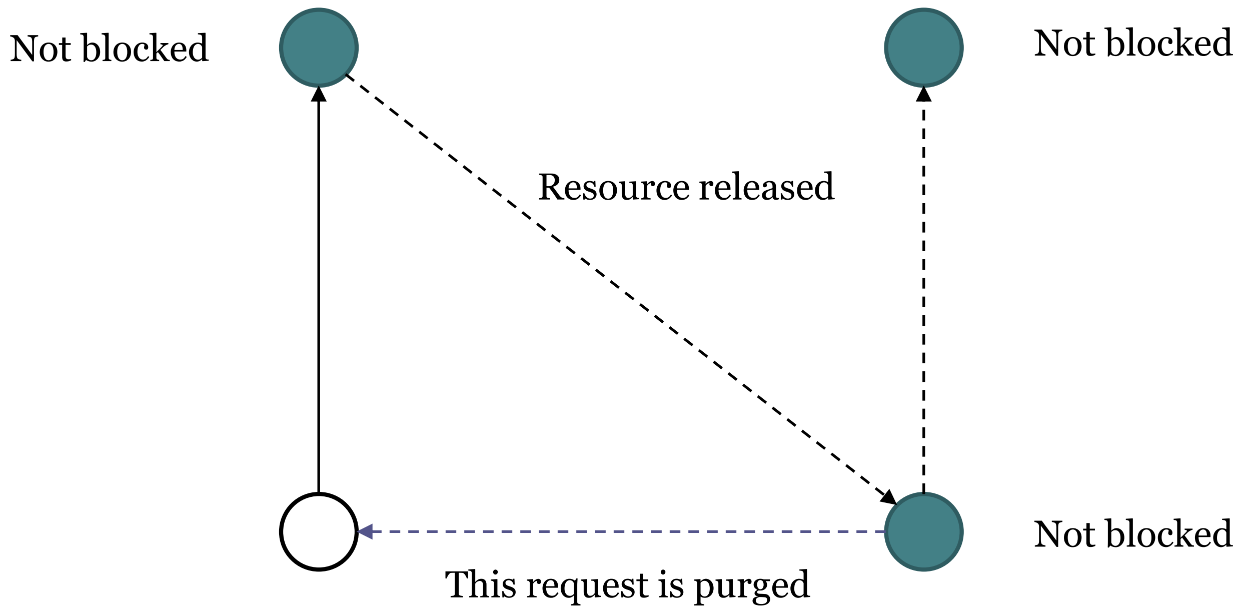
Central Detection with WFG Reduction

Static analysis of an OR-deadlock by reduction of a WFG snapshot



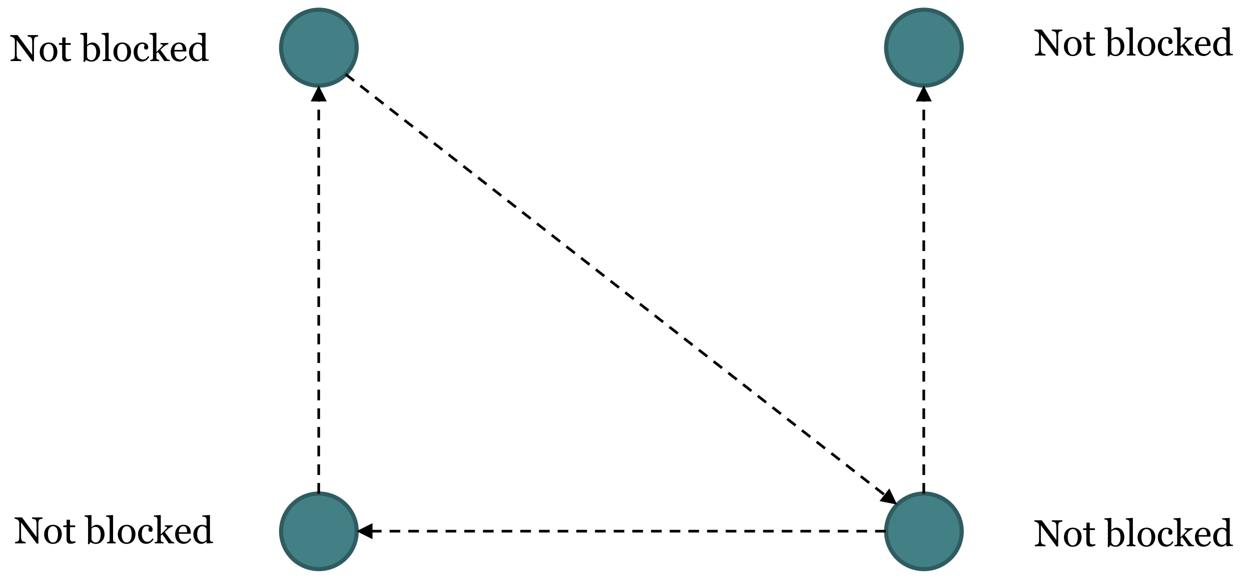
Central Detection with WFG Reduction

Static analysis of an OR-deadlock by reduction of a WFG snapshot



Central Detection with WFG Reduction

Static analysis of an OR-deadlock by reduction of a WFG snapshot



No deadlock! A cycle is not a sufficient condition for models: OR and P-out-of-Q

Knot

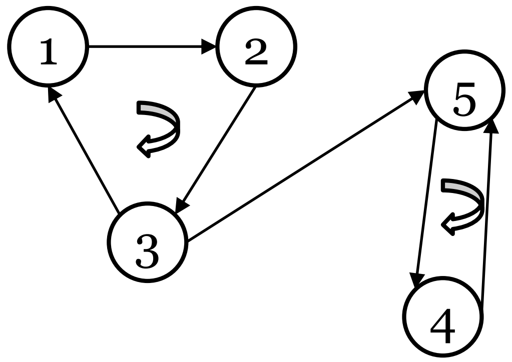
- Knot is a sufficient condition to OR and P-out-of-Q
- Ri is the reachable set of Pi
  - Set of all processes there is a path from Pi
- A nonempty set of process S is said to be a knot, if
  - $\forall Pi \in S$ are blocked
  - $Pi \in S$ iff $\forall Pj \in Ri \rightarrow Pi \in Rj$
    - i.e., every process reached from a process Pi of a knot also reaches Pi
  - There is no grant message (releasing of resource) over the network among these processes

R1={P2,P3,P1,P5,P4}
R2={P3,P1,P2,P5,P4}
R3={P1,P2,P3,P5,P4}
R4={P5,P4}, R5={P4,P5}

S={P1,P2,P3} is not a knot (deadlock) because P5 is in R1, R2, R3, but P5 does not reach P1, P2 and P3

S={P4,P5} is a knot (deadlock) because P4 is reachable from each one in R4 and P5 is reachable from each one in R5

Two cycles (two AND deadlock)



Outline

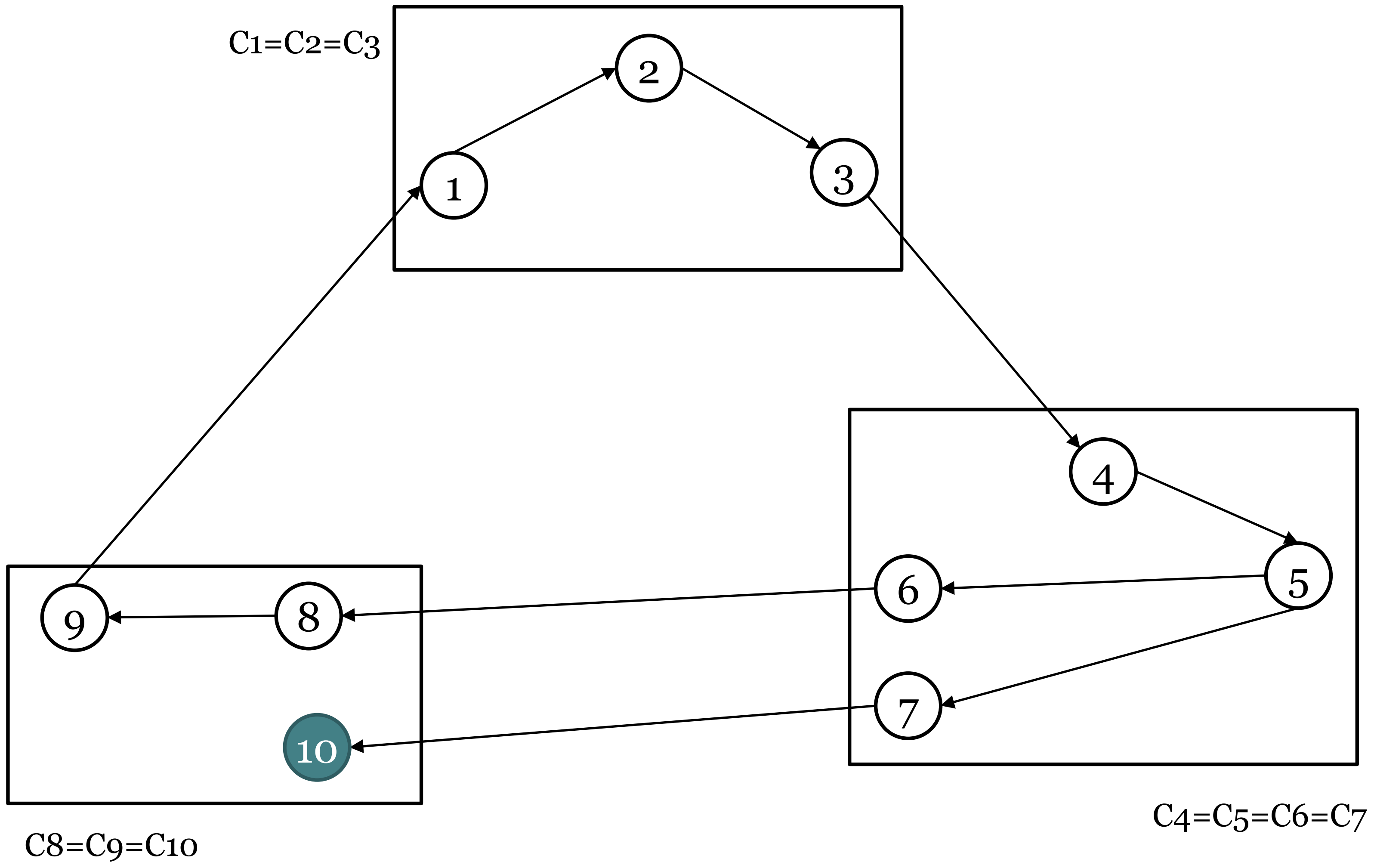
- Deadlock
- Prevention
- Detection
- **Chandy-Misra-Haas for AND**
- Chandy-Misra-Haas for OR

Chandy-Misra-Haas AND

- 1983 - Chandy, Misra, Haas, “Distributed Deadlock Detection”, ACM TOCS, vol. 1, no 2, May 1983.
- Algorithm for the AND model (edge-chasing)
  - Resource deadlock detection
- There are some mechanism to deal with local deadlocks
  - Consider n processes P1, P2, ... Pn in a single site/controller
  - P1 is locally dependent if P1 → P2 → ... Pn and Pn → P1
- Sites Ci may have more than one process Pi
  - Single site Ci can have more than one label (reference)
  - Ex.: if Pi and Pj are inside Ck, then Ck=Ci=Cj is a single site
- The algorithm detects if a blocked process is deadlocked among different sites/controllers with a partial view of the WFG

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas AND



Chandy-Misra-Haas AND

- It uses a special message called *probe*, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process Pi and it is being sent by the home site of process Pj to the home site of process Pk
    - Probe messages travel among sites along the edges of the global WFG graph
- Data Structures
    - Processes Pk have a boolean array: dependent[k,i]
    - dependent[k,i]=true if Ck knowns that Pi → Pk
    - Initially, dependent[k,i]=false for all k, i
- Summary
    - A cycle in WFG is sufficient to detect an AND-deadlock
    - A deadlock for Pi is detected when a probe message returns to the initial process Pi

Chandy-Misra-Haas AND

```
Start from Pi (site Ci):
   if Pi is locally dependent
      declare deadlock
   else $\forall Pj, Pk : Pi \rightarrow Pj \rightarrow Pk,\ Pj \in Ci,\ Pk \notin Ci$
      send probe(i,j,k) to site Ck (Pk)
```

$\longrightarrow$ Another mechanism deal with local deadlocks

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

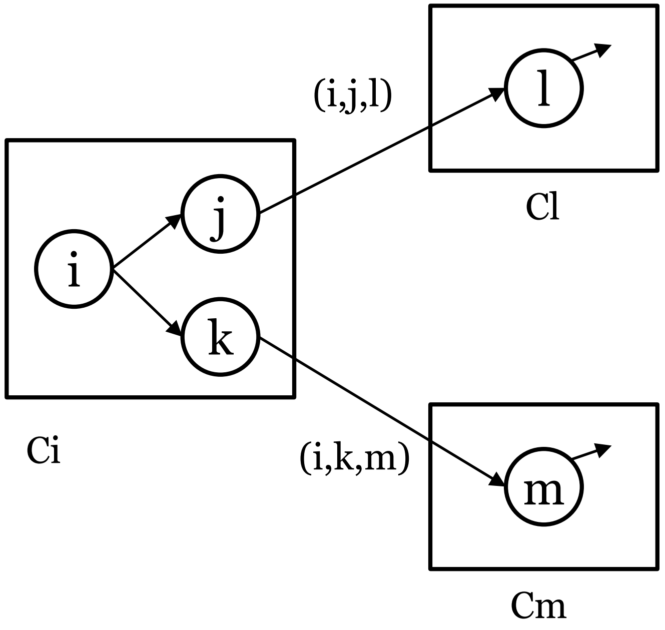
Chandy-Misra-Haas AND

```
Start from Pi (site Ci):
   if Pi is locally dependent
      declare deadlock
   else $\forall Pj, Pk : Pi \rightarrow Pj \rightarrow Pk,\ Pj \in Ci,\ Pk \notin Ci$
      send probe(i,j,k) to site Ck (Pk)
```

→ Another mechanism deal with local deadlocks

A blocked process (task) Pi starts deadlock detection in Ci

Ci sends probe(i,j,k) messages to all dependent sites Ck



Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas AND

```
Start from Pi (site Ci):
   if Pi is locally dependent
      declare deadlock
   else $\forall Pj, Pk : Pi \rightarrow Pj \rightarrow Pk,\ Pj \in Ci,\ Pk \notin Ci$
      send probe(i,j,k) to site Ck (Pk)
```

```
Active Pk:
   dependent[k,i] = false
```

$\longrightarrow$

When a process Pk becomes active (receive the resource), it does not wait for other process; so, its boolean array is cleared

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas AND

```
Start from Pi (site Ci):
   if Pi is locally dependent
      declare deadlock
   else $\forall Pj, Pk : Pi \rightarrow Pj \rightarrow Pk,\ Pj \in Ci,\ Pk \notin Ci$
      send probe(i,j,k) to site Ck (Pk)

Receiving probe(i,j,k) (site Ck):
   if (
    Pk is blocked/idle,
    dependent[k,i] = false, and
    Pk has not replied to all requests Pj
   ) {
      dependent[k,i] = true
      if k=i
         declare Pi as deadlocked
      else $\forall Pm, Pn : Pk \rightarrow Pm \rightarrow Pn,\ Pm \in Ci,\ Pn \notin Ci$
         send probe(i,m,n) to site Cn (Pn)
   }

Active Pk:
   dependent[k,i] = false
```

When a process Pk receives probe(i,j,k), it ignores the probe if it is active

It forwards the detection if it is the first time that the detection from i reaches Pk
i.e. Now, Pk knows that Pi → Pk

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas AND

```
Start from Pi (site Ci):
   if Pi is locally dependent
      declare deadlock
   else $\forall Pj, Pk:\ Pi \rightarrow Pj \rightarrow Pk,\ Pj \in Ci,\ Pk \notin Ci$
      send probe(i,j,k) to site Ck (Pk)

Receiving probe(i,j,k) (site Ck):
   if (
    Pk is blocked/idle,
    dependent[k,i] = false, and
    Pk has not replied to all requests Pj
   ) {
      dependent[k,i] = true
      if k=i
         declare Pi as deadlocked
      else $\forall Pm, Pn:\ Pk \rightarrow Pm \rightarrow Pn,\ Pm \in Ci,\ Pn \notin Ci$
         send probe(i,m,n) to site Cn (Pn)
   }

Active Pk:
   dependent[k,i] = false
```

$\longrightarrow$ During the forward
1 – mark the dependence
discovered: Pi → Pk

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas AND

```
Start from Pi (site Ci):
   if Pi is locally dependent
      declare deadlock
   else $\forall Pj, Pk : Pi \rightarrow Pj \rightarrow Pk,\ Pj \in Ci,\ Pk \notin Ci$
      send probe(i,j,k) to site Ck (Pk)

Receiving probe(i,j,k) (site Ck):
   if (
    Pk is blocked/idle,
    dependent[k,i] = false, and
    Pk has not replied to all requests Pj
   ) {
      dependent[k,i] = true
      if k=i
         declare Pi as deadlocked
      else $\forall Pm, Pn : Pk \rightarrow Pm \rightarrow Pn,\ Pm \in Ci,\ Pn \notin Ci$
         send probe(i,m,n) to site Cn (Pn)
   }

Active Pk:
   dependent[k,i] = false
```

$\longrightarrow$ During the forward
1 – mark the dependence discovered: Pi → Pk
2 – forward the detection to new sites (more than one send may occur)

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas AND

```
Start from Pi (site Ci):
   if Pi is locally dependent
      declare deadlock
   else $\forall Pj, Pk : Pi \rightarrow Pj \rightarrow Pk,\ Pj \in Ci,\ Pk \notin Ci$
      send probe(i,j,k) to site Ck (Pk)

Receiving probe(i,j,k) (site Ck):
   if (
    Pk is blocked/idle,
    dependent[k,i] = false, and
    Pk has not replied to all requests Pj
   ) {
      dependent[k,i] = true
      if k=i
         declare Pi as deadlocked
      else $\forall Pm, Pn : Pk \rightarrow Pm \rightarrow Pn,\ Pm \in Ci,\ Pn \notin Ci$
         send probe(i,m,n) to site Cn (Pn)
   }

Active Pk:
   dependent[k,i] = false
```

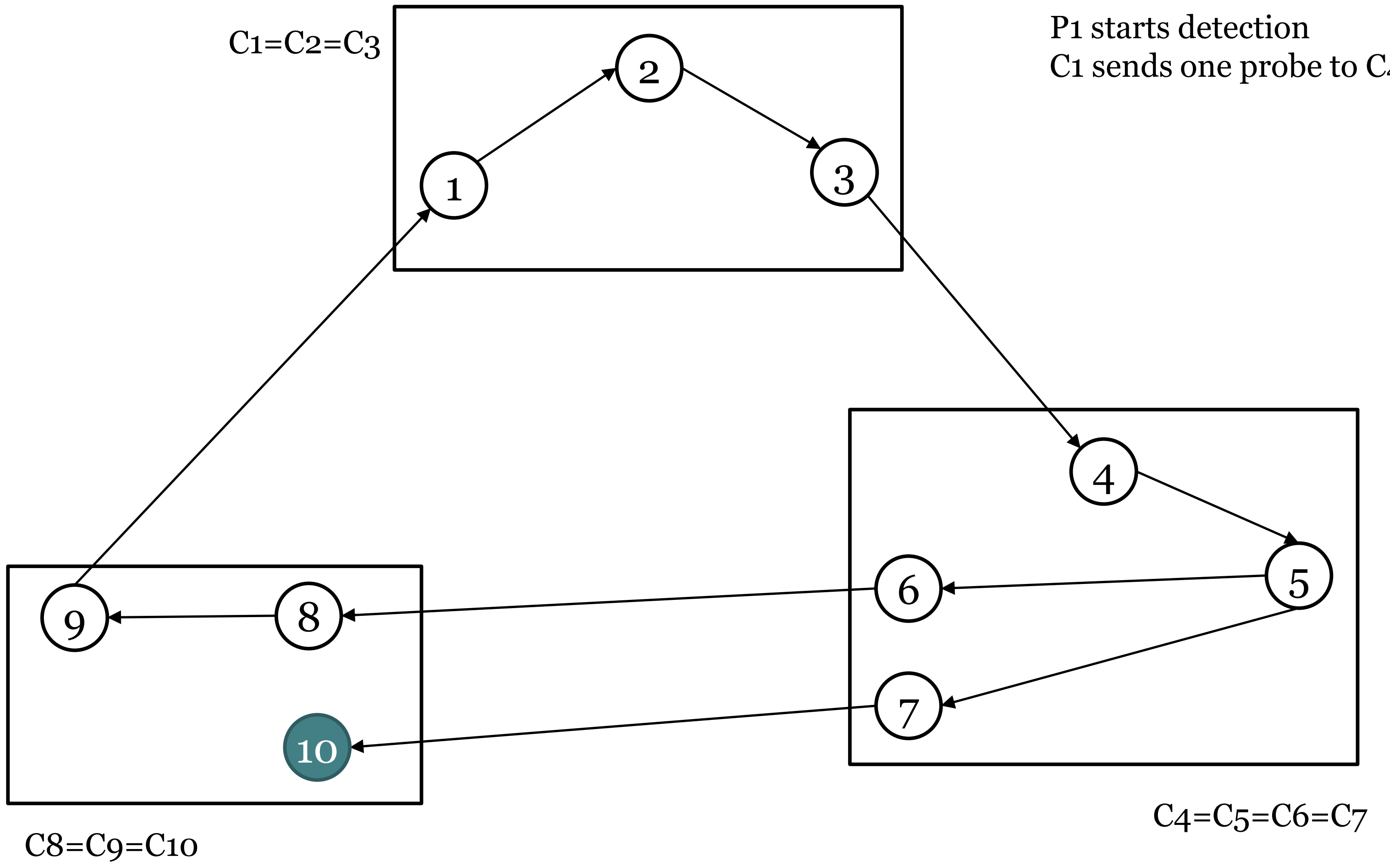
$\longrightarrow$

During the forward
1 – mark the dependence discovered: Pi → Pk
2 – forward the detection to new sites (more than one send may occur)

Deadlock, when a probe completes a cycle (return to origin)

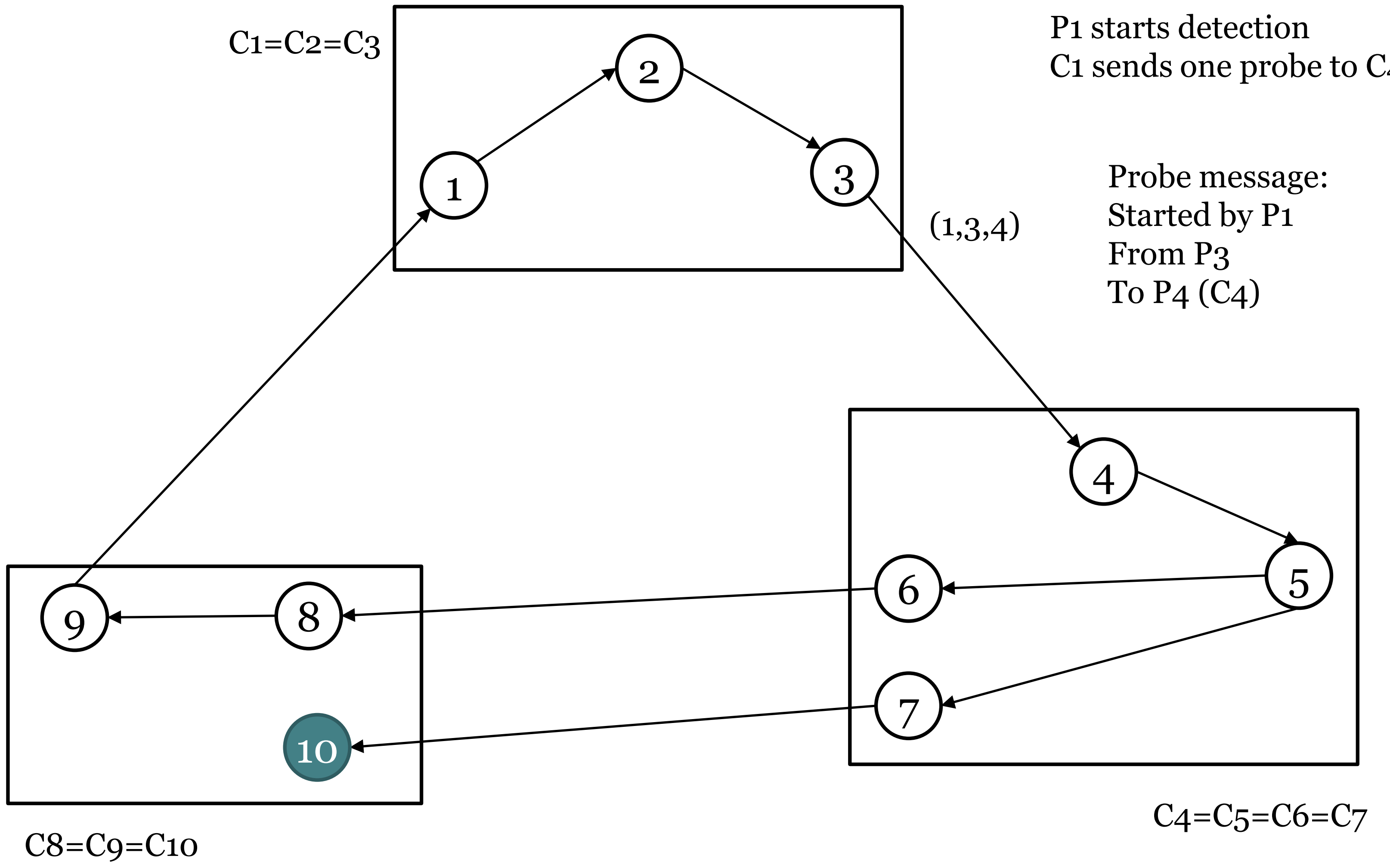
Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas AND



P1 starts detection
C1 sends one probe to C4

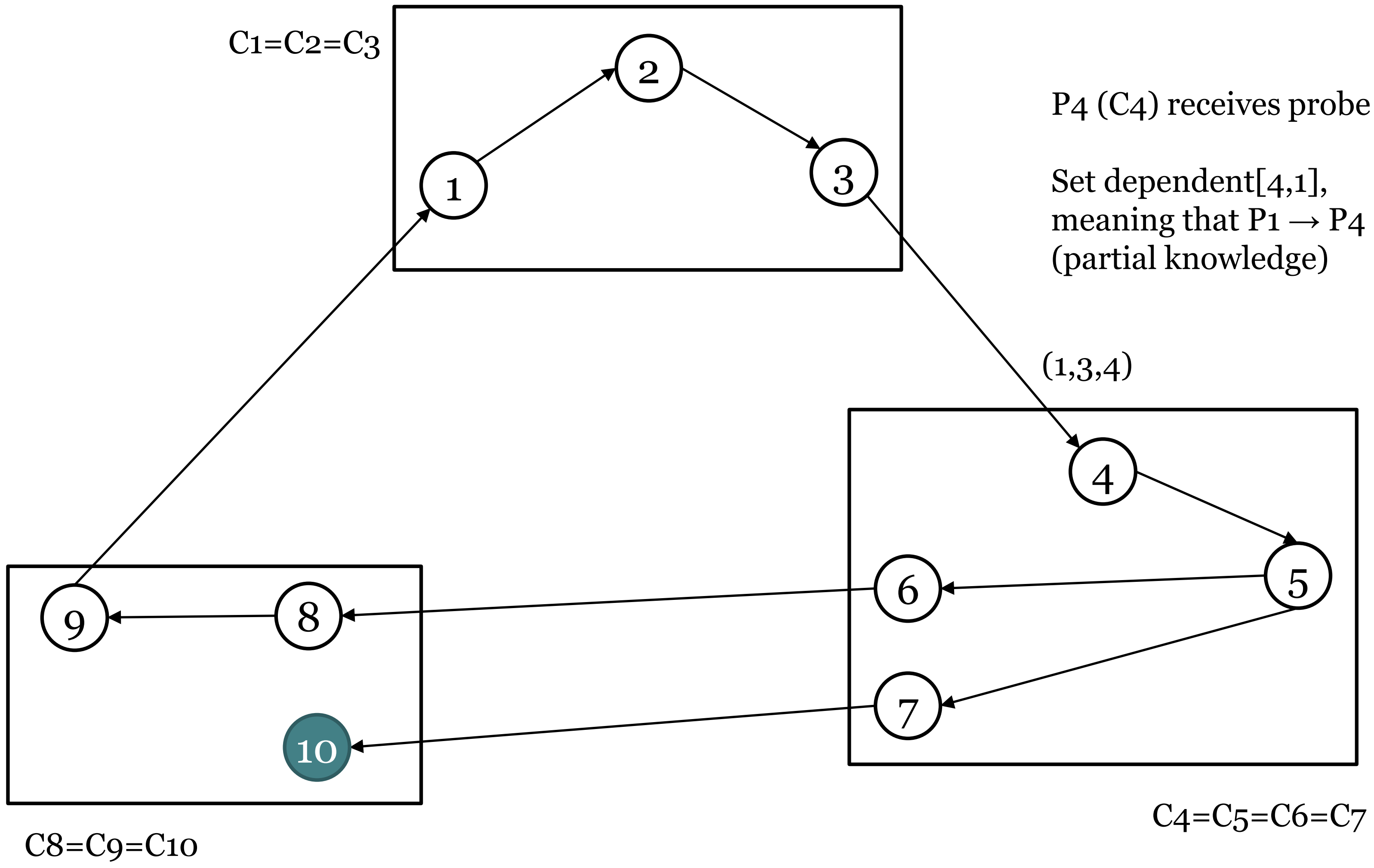
Chandy-Misra-Haas AND



P1 starts detection
C1 sends one probe to C4

Probe message:
Started by P1
From P3
To P4 (C4)

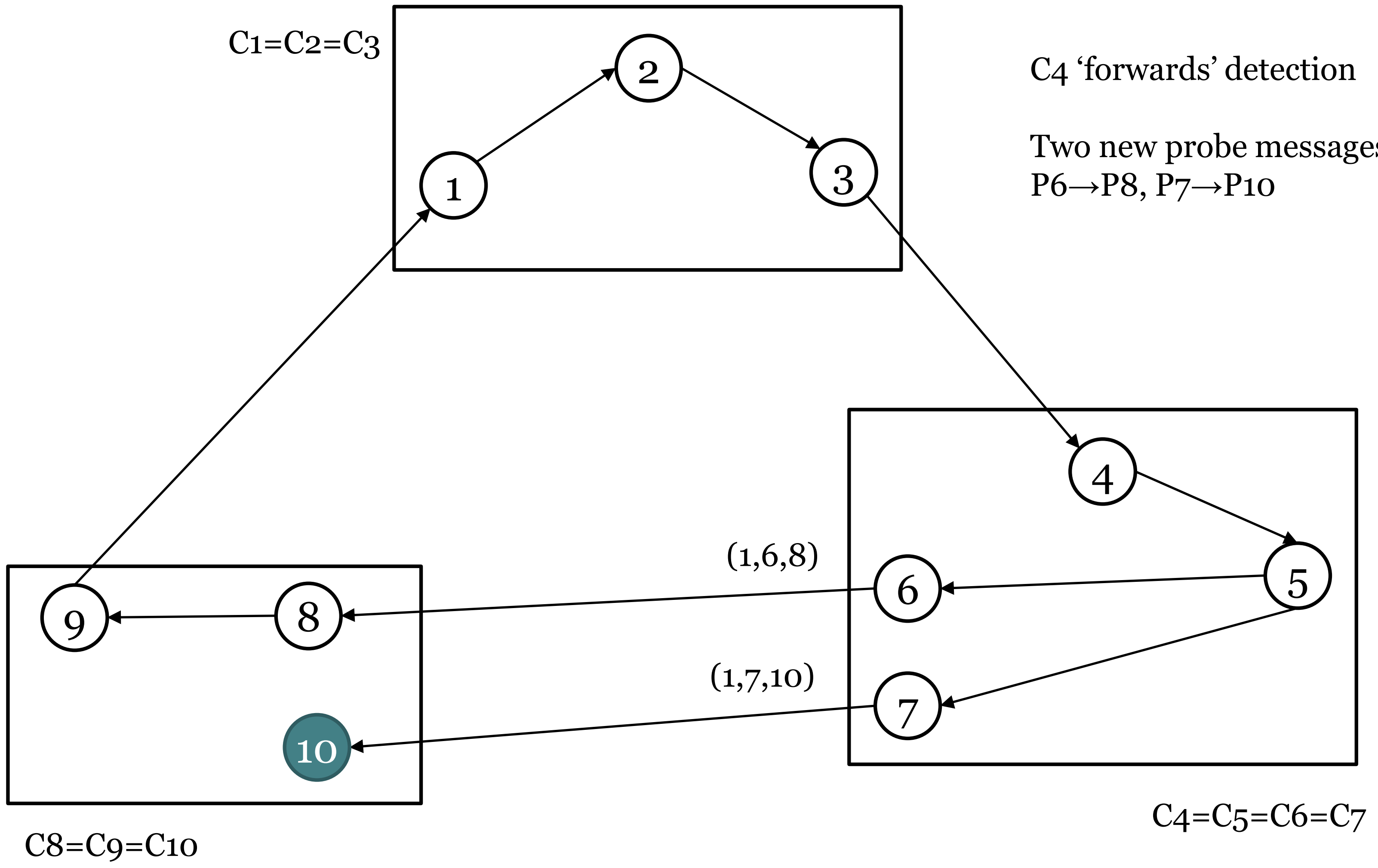
Chandy-Misra-Haas AND



P4 (C4) receives probe

Set dependent[4,1], meaning that P1 → P4 (partial knowledge)

Chandy-Misra-Haas AND

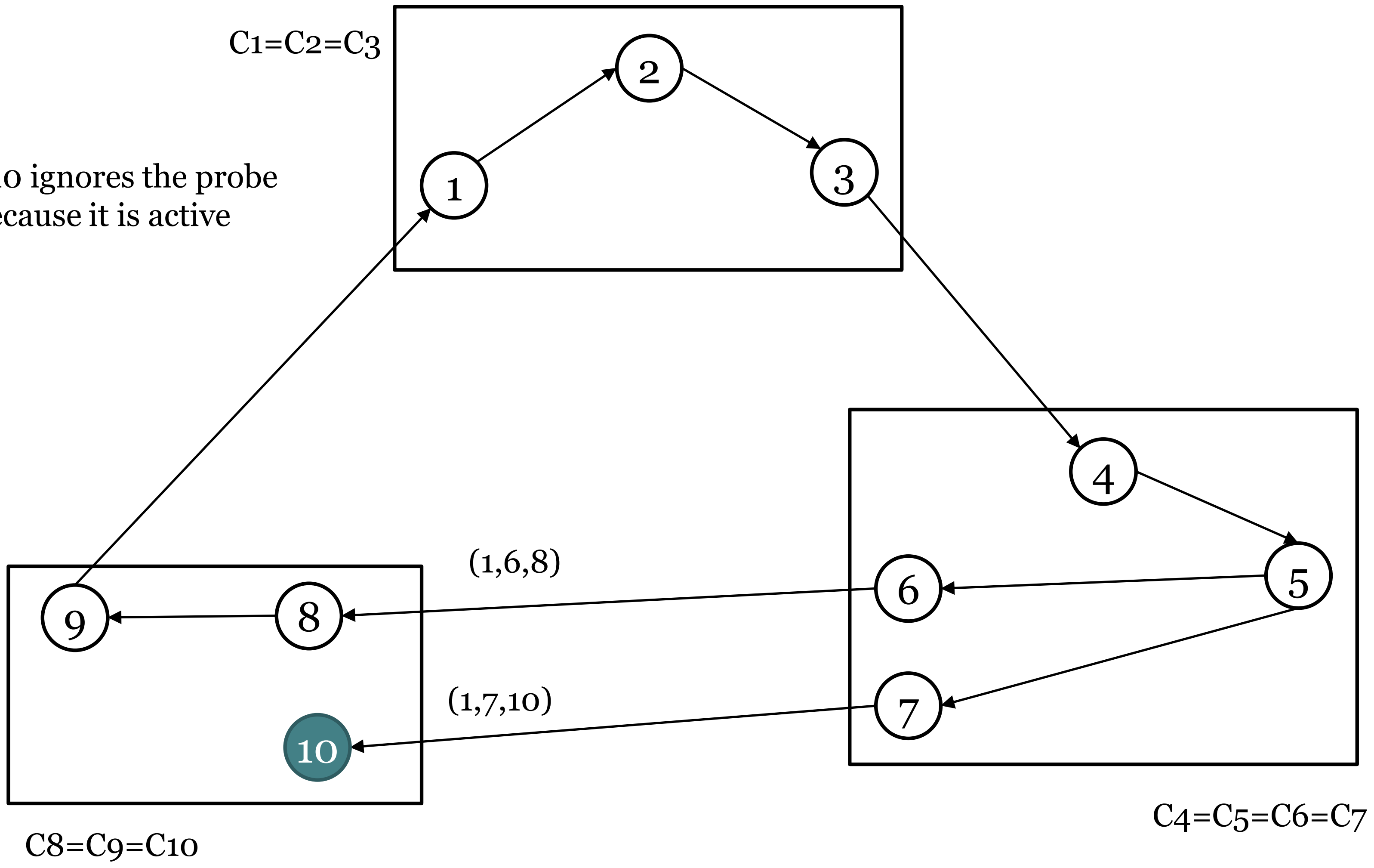


C4 ‘forwards’ detection

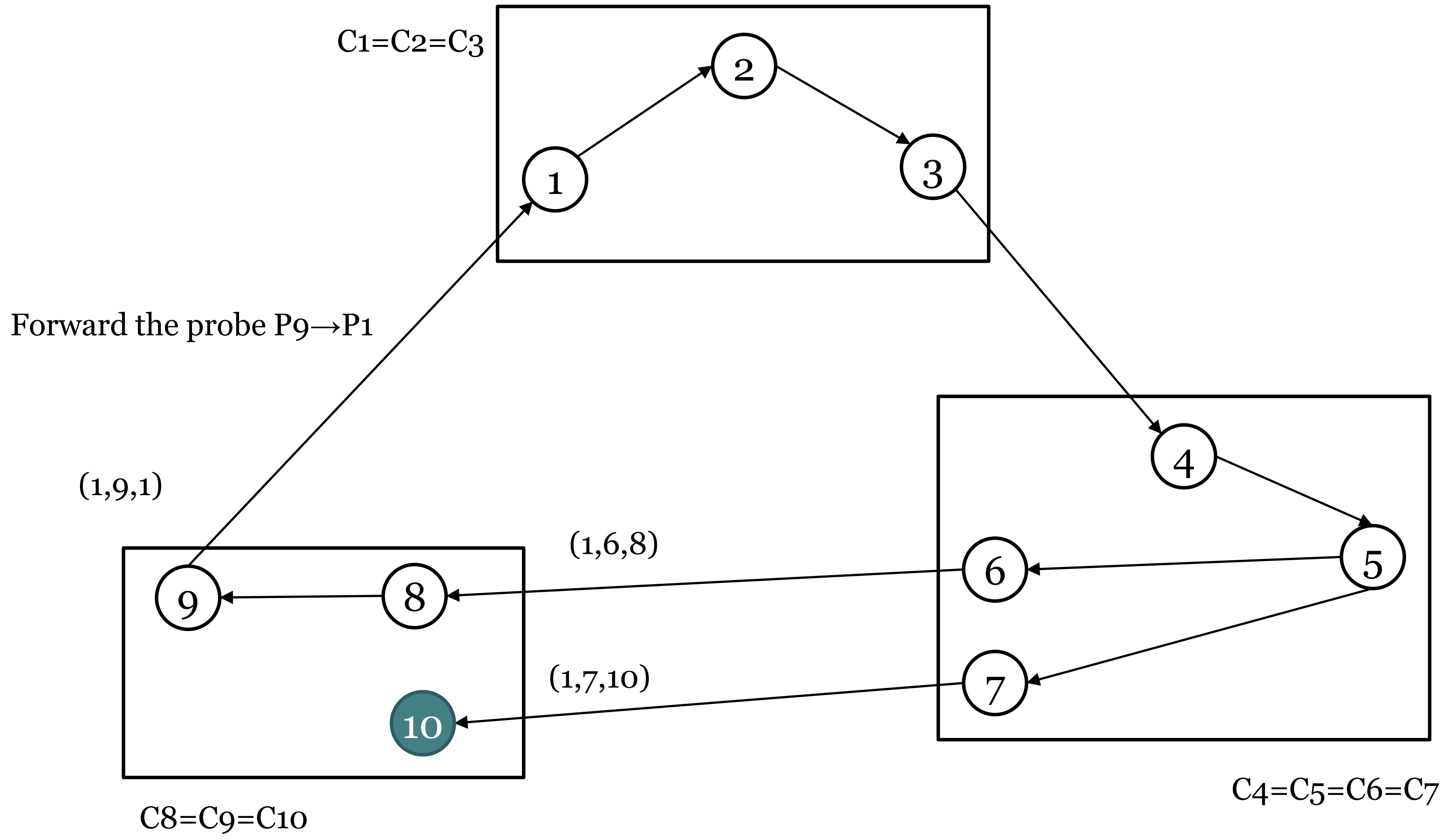
Two new probe messages:
P6→P8, P7→P10

Chandy-Misra-Haas AND

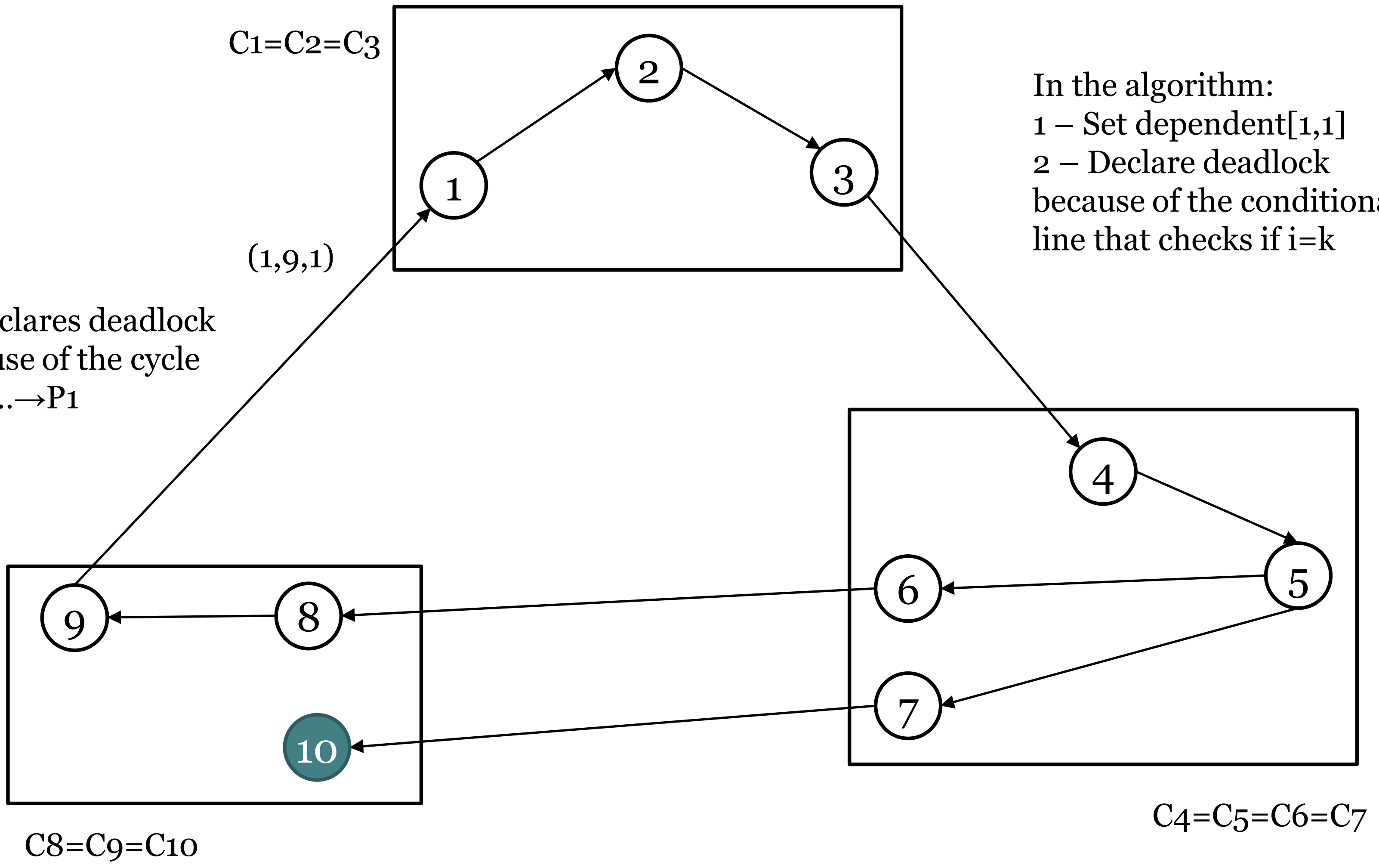
P10 ignores the probe
because it is active



Chandy-Misra-Haas AND



Chandy-Misra-Haas AND

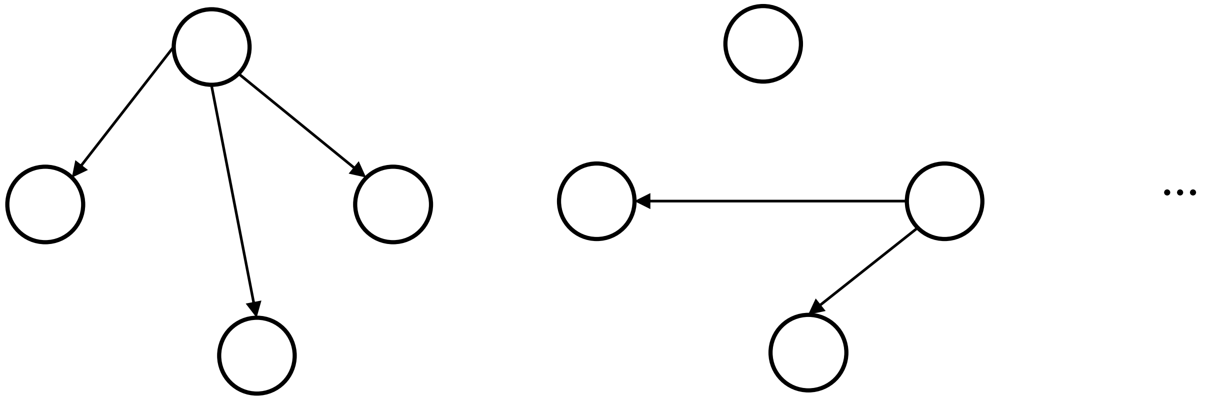


In the algorithm:
1 – Set dependent[1,1]
2 – Declare deadlock because of the conditional line that checks if i=k

C1 declares deadlock because of the cycle P1→…→P1

Chandy-Misra-Haas AND

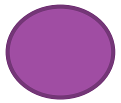
- One probe message (per detection initiation) is sent on every edge of a cycle of the WFG among different sites
- Delay of the deadlock detection is O(n)
- According to Mikhail Nesterenko, Kshemkalyani & Singhal:
  - Consider that there is at least one site per process in deadlock
  - If a process is waiting for n-1 other process, and
  - The next process is waiting for n-2 other process, and
  - The next process is waiting for n-3 other process, and
  - …
  - Then, the worst case is n(n-1)/2 messages: O(n^2)



Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]
http://vega.cs.kent.edu/~mikhail/classes/aos.f03/l10deadlocks2.PDF

Chandy-Misra-Haas AND

- After the detection of deadlock by Pi, Pi may kill itself breaking the cycle
- Now, consider that all processes start deadlock detection at the same time. What will happen?
  - i.e., for *n* processes, there are *n* probe messages traveling on the cycle



Chandy-Misra-Haas AND

- After the detection of deadlock by Pi, Pi may kill itself breaking the cycle
- Now, consider that all processes start deadlock detection at the same time. What will happen?
  - i.e., for *n* processes, there are *n* probe messages traveling on the cycle
- Mass suicide!
- We may include the largest or smallest process ID within the probe during the cycle
- Smarter solutions can kill the process with more cycles, thus reducing the side effect

Outline

- Deadlock
- Prevention
- Detection
- Chandy-Misra-Haas for AND
- **Chandy-Misra-Haas for OR**

Chandy-Misra-Haas OR

- 1983 - Chandy, Misra, Haas, “Distributed Deadlock Detection”, ACM TOCS, vol. 1, no 2, May 1983
- Algorithm for the OR model (diffusing)
  - Communication deadlock detection
- The algorithm detects if a blocked process is deadlocked with a diffusion on the WFG
- Any blocked process may start the detection: root of the diffusion computation
- Two types of messages are used in this diffusion computation:
  - Query(i,m,j,k): the m-th query (detection) initiated by Pi, sending from Pj to Pk
  - Reply(i,m,k,j): reply from Pk to Pj referring to the query(i,m,j,k) of Pj

Chandy-Misra-Haas OR

- Summary
  - A blocked process Pi starts the detection by sending query messages to all processes it is waiting for: set WFi
  - Active processes ignore all messages: query and reply
  - Each blocked process Pj forwards a new query (detection) and reply to a repeated query
    - If forwarded again, the algorithm never finishes
  - When a blocked process Pj receives all pending replies, it can also reply, once all processes below them in the diffusion tree also replied, i.e., they are all blocked and waiting for resources
    - If Pj is the root of the diffusion, it declares deadlock

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas OR

- Data Structures
- Each process Pk has four arrays of size *n* (for a total of *n* process)
    - latest[i]: highest m of all queries(i,m,j,k) (initially zero)
    - engager[i]: register the process to reply (initially any value). It is *j* if Pj is responsible for latest[i], $i \neq j$
    - num[i]: number of pending replies (initially zero), i.e., amount of queries(i,m,k,j) sent by Pk less replies(i,m,j,k) received by Pk
    - wait[i]: check if Pk did not wake up (initially false); true iff Pk is waiting (blocked) since the last update of lasted[i]
- Note: engager[i] and num[i] register information of the diffusion tree
- Note: An engaging query is the first m-th query(i,m,j,k) received by Pk for the detection initiated by Pi

Chandy-Misra-Haas OR

```
Start from Pi:
   latest[i]++; m=latest[i]
   wait[i]=true; num[i]=|WF_i|
   ∀ Pj ∈ WF_i: send query(i,m,i,j)

Active Pk:
   wait[…]=false
   ignore any reply and query
```

$\longrightarrow$ All active processes clean its array wait and ignore all messages (reply "means" blocked)

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas OR

```
Start from Pi:
   latest[i]++; m=latest[i]
   wait[i]=true; num[i]=|WF_i|
   $\forall Pj \in WF_i$: send query(i,m,i,j)

Active Pk:
   wait[…]=false
   ignore any reply and query
```

$\longrightarrow$

A process Pi starts its m-th detection by incrementing latest[i] and setting its position in the array wait

Then, Pi checks the processes it is waiting for and sends query for all of them

Now, num[i] is set with |WFi| pending replies

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas OR

```
Blocked Pk receives a query(i,m,j,k):
   // if engaging query for Pi
   if (m>latest[i]) {
      latest[i]=m; engager[i]=j
      wait[i]=true; num[i]=|$WF_k$|
      $\forall Pr \in WF_k$: send query(i,m,k,r)


   // if not engaging query
   } else if wait[i] and m=latest[i]
      send reply(i,m,k,j)
```

$\longrightarrow$

When a blocked process receives a query message (Pi → Pj → Pk)

If it is an engaging query, it forwards the detection:
- save j in engager[i] to reply in the future (parent in the diffusion tree)
- Save the amount of pending replies (num[i]) it must wait before it can reply to its engager[i]

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas OR

```
Blocked Pk receives a query(i,m,j,k):
   // if engaging query for Pi
   if (m>latest[i]) {
      latest[i]=m; engager[i]=j
      wait[i]=true; num[i]=|$WF_k$|
      ∀ $Pr \in WF_k$: send query(i,m,k,r)

   // if not engaging query
   } else if wait[i] and m=latest[i]
      send reply(i,m,k,j)
```

$\longrightarrow$

In a knot, directly or indirectly, every process is waiting for all other processes

So, we have to add a boundary condition to stop the forwarding of queries

We stop it by avoiding duplication of forwarding

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas OR

```
Blocked Pk receives a query(i,m,j,k):
   // if engaging query for Pi
   if (m>latest[i]) {
      latest[i]=m; engager[i]=j
      wait[i]=true; num[i]=|$WF_k$|
      $\forall Pr \in WF_k$: send query(i,m,k,r)


   // if not engaging query
   } else if wait[i] and m=latest[i]
      send reply(i,m,k,j)

Blocked Pk receives a reply(i,m,r,k):
   if wait[i] and m=latest[i]
      num[i]--
      if num[i]=0
         if i=k
            declare Pk as deadlocked
         else
            send reply(i,m,k,engager[i])
```

Pk may receive a reply from an old detection initiated by Pi

When Pk receives a correct reply (m is equal latest[i]) and it is still blocked

If there is no more pending replies, it sends its reply to engager[i] or declares deadlock if Pk has started the detection

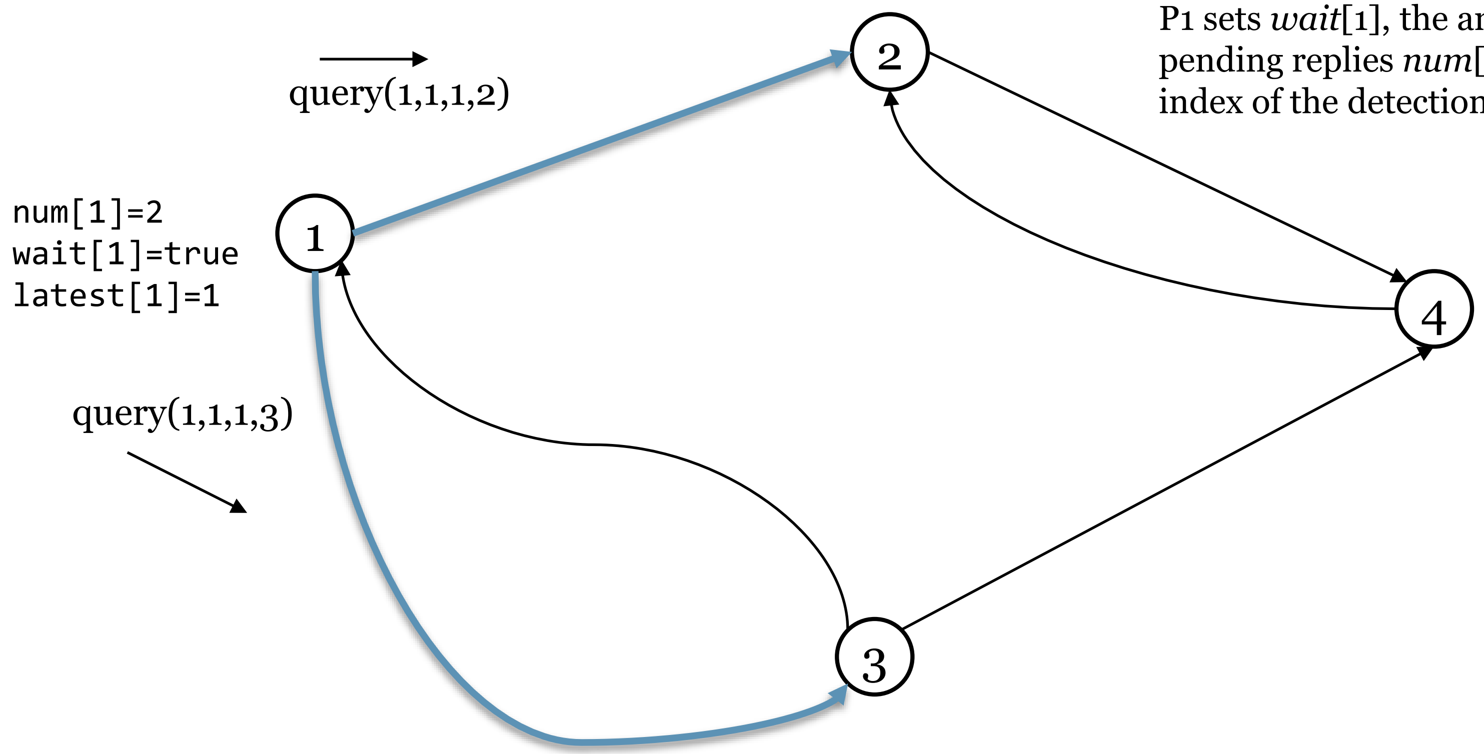
$\longrightarrow$ In other words, the processes of the knot in the WTG are blocked and waiting for some resources and there is no active process that will release resources, once active processes ignore queries and do not send replies

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Chandy-Misra-Haas OR

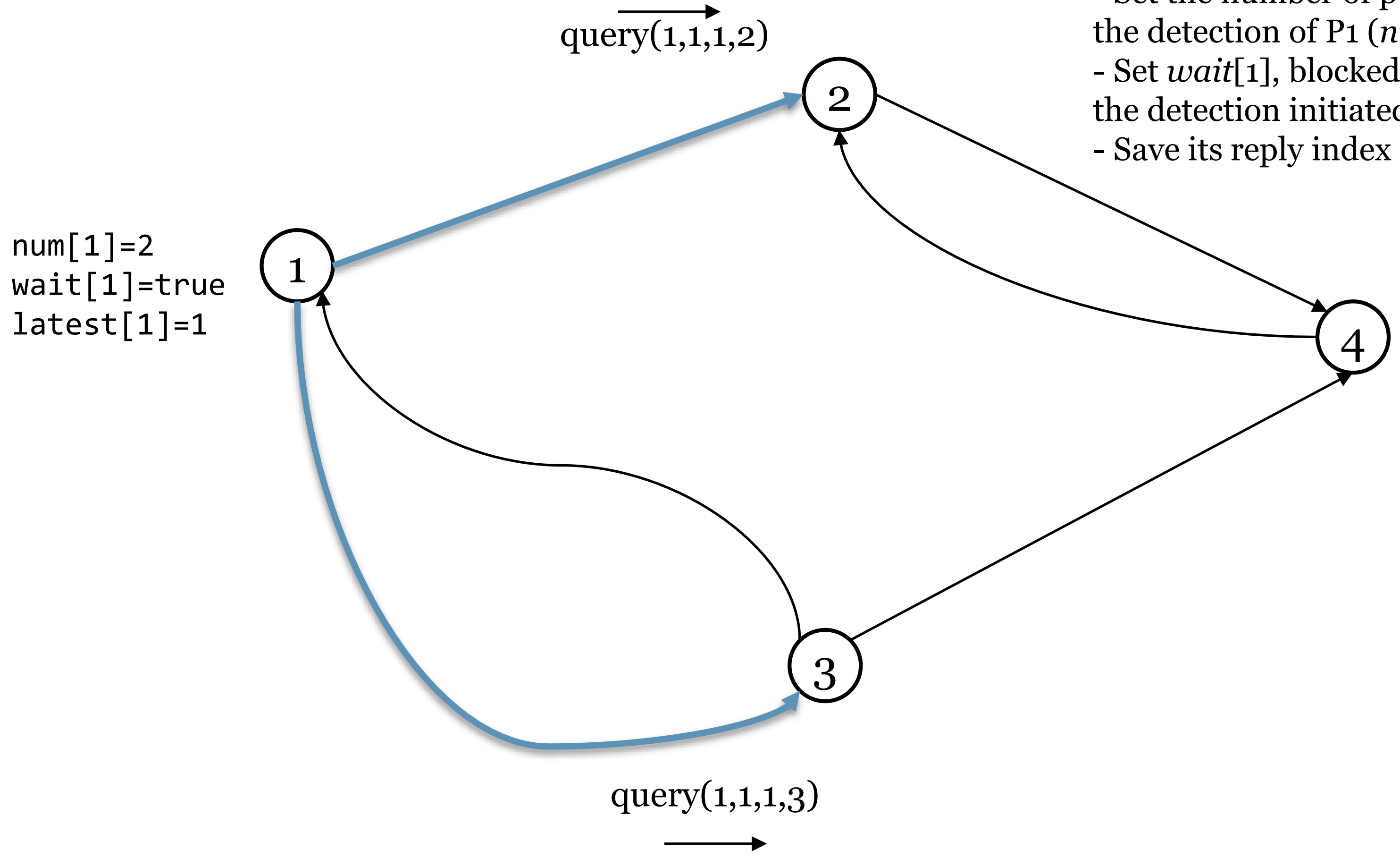
P1 starts detection
It sends query messages to P2 and P3

P1 sets *wait*[1], the amount of pending replies *num*[1]=2, and the index of the detection *latest*[1]=1



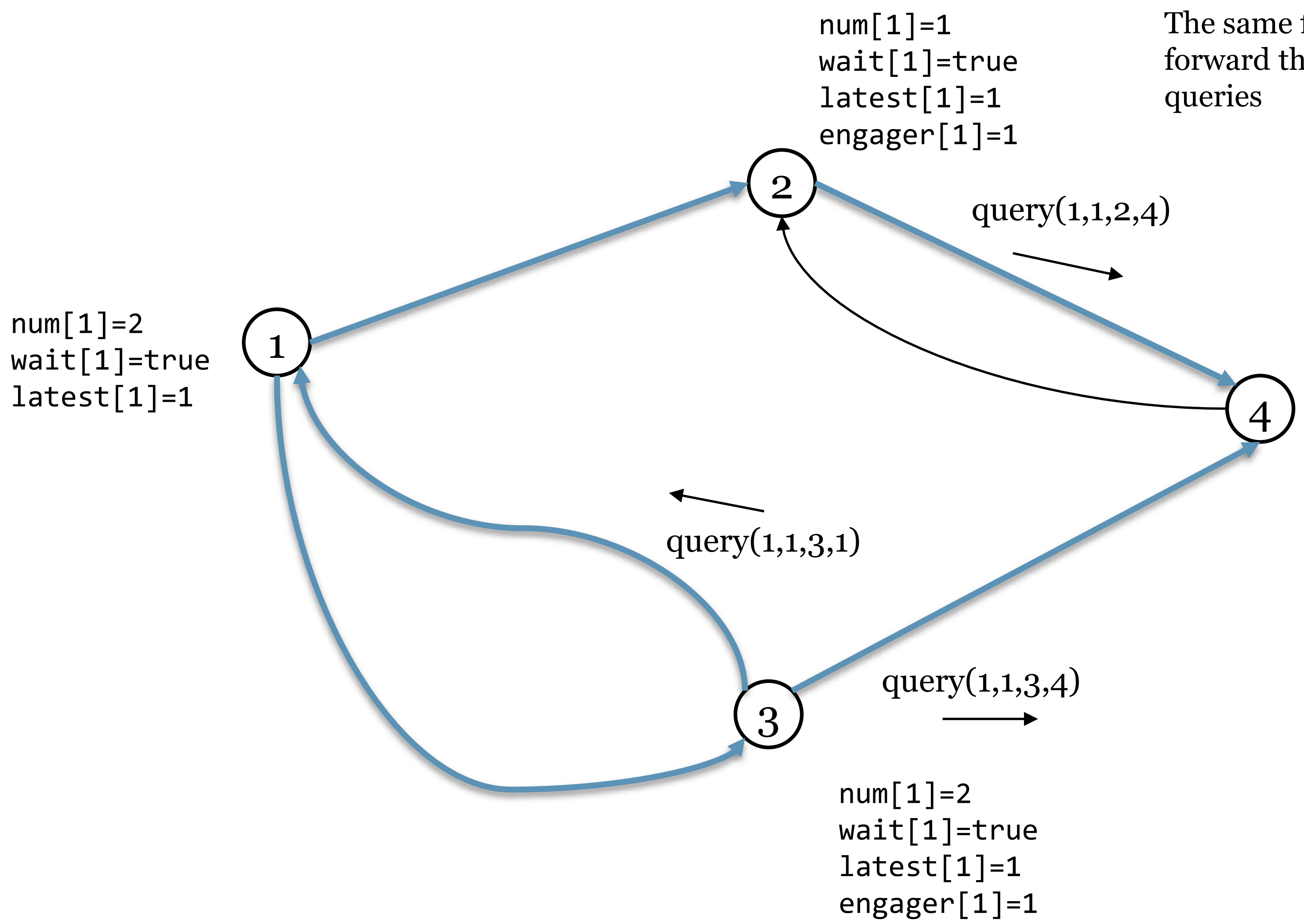
Chandy-Misra-Haas OR

P2 receives the engaging query(1,1,1,2)
- Update the index (m=1) of the detection initiated by P1 (*latest*[1]=*m*=1)
- Set the number of pending replies for the detection of P1 (*num*[1]=1)
- Set *wait*[1], blocked since the start of the detection initiated by P1
- Save its reply index (engager[1]=j=1)



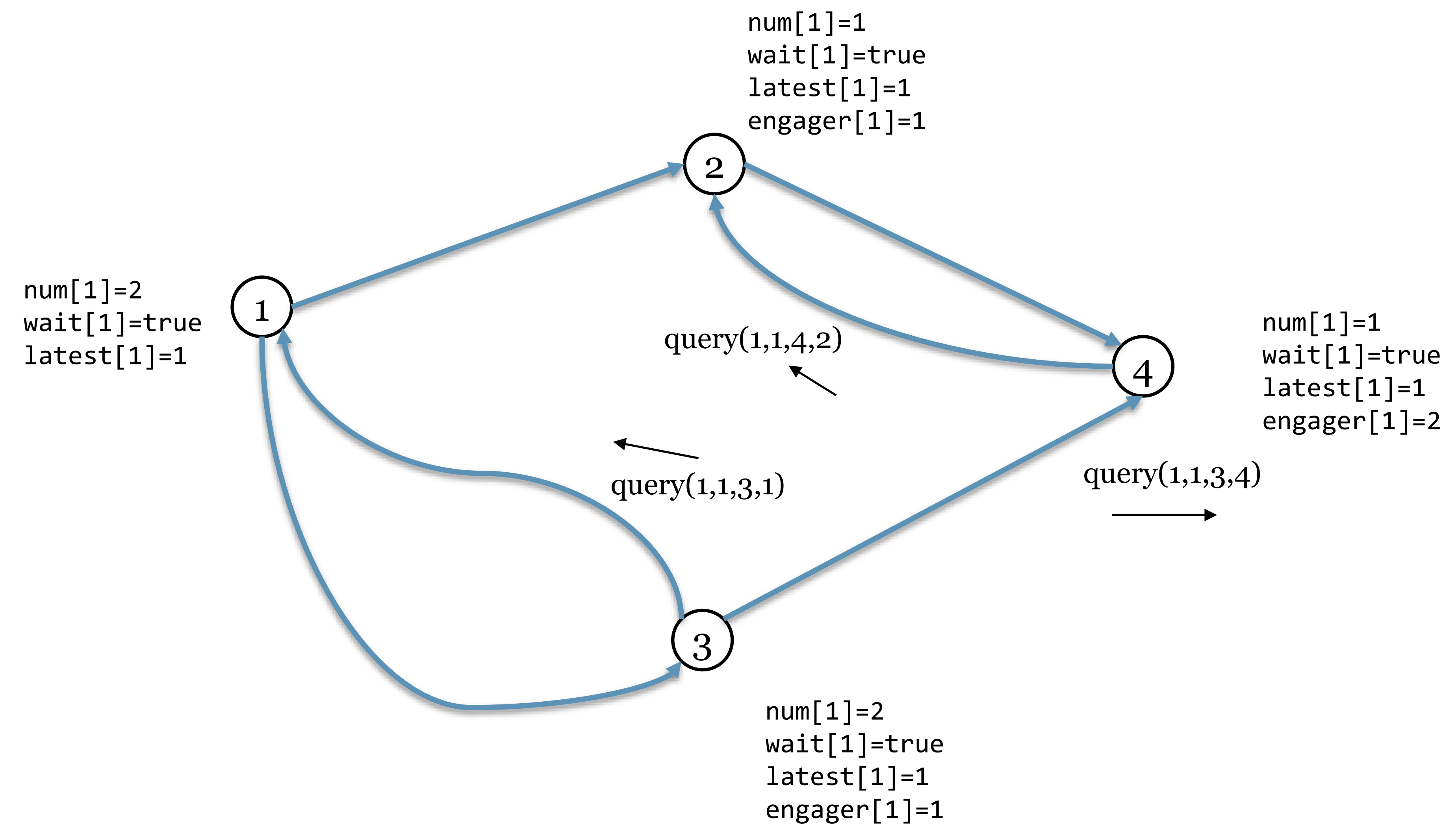
Chandy-Misra-Haas OR

The same for P3. Then, they forward the detection with new queries



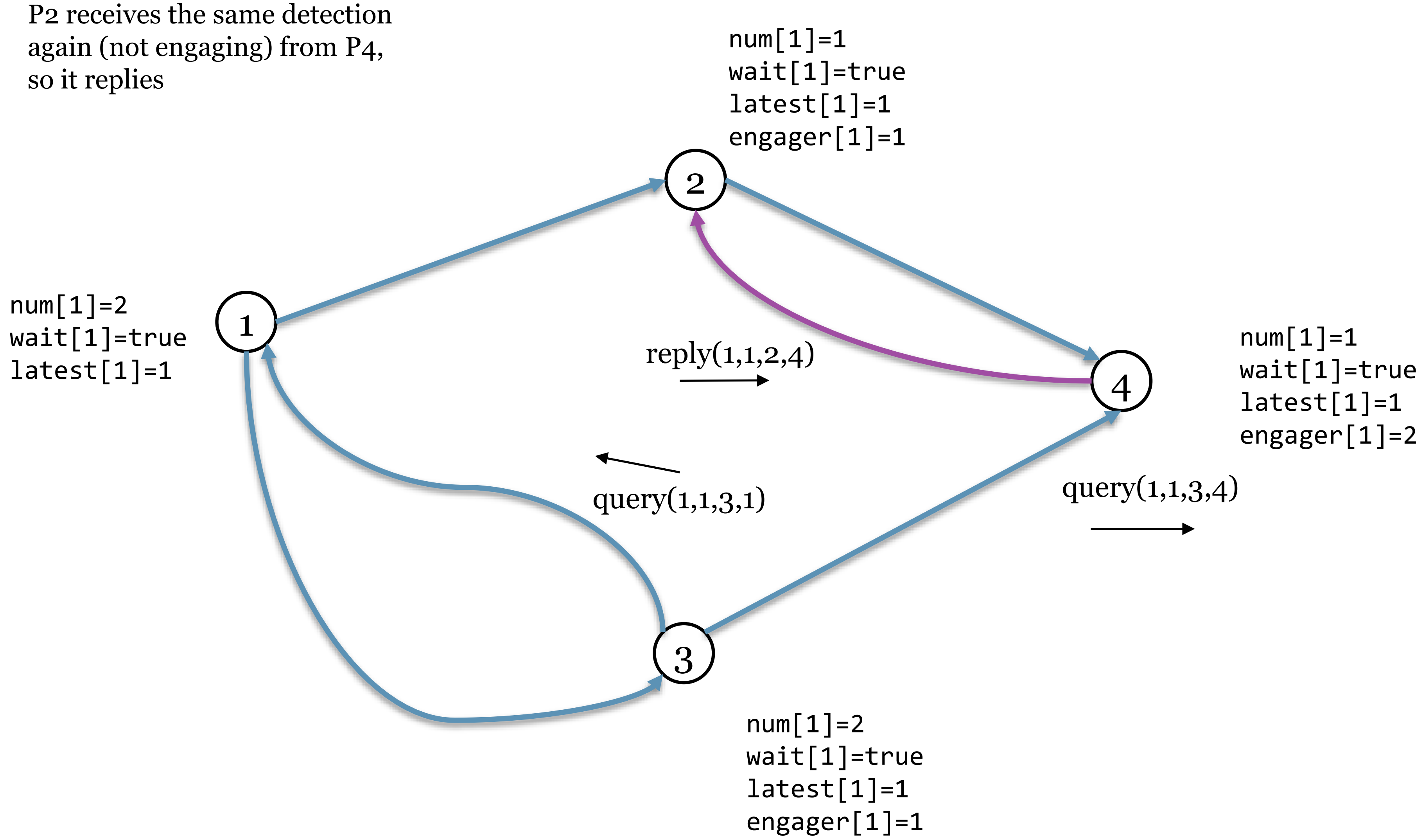
Chandy-Misra-Haas OR

P4 receives the query from P2 and forwards the detection



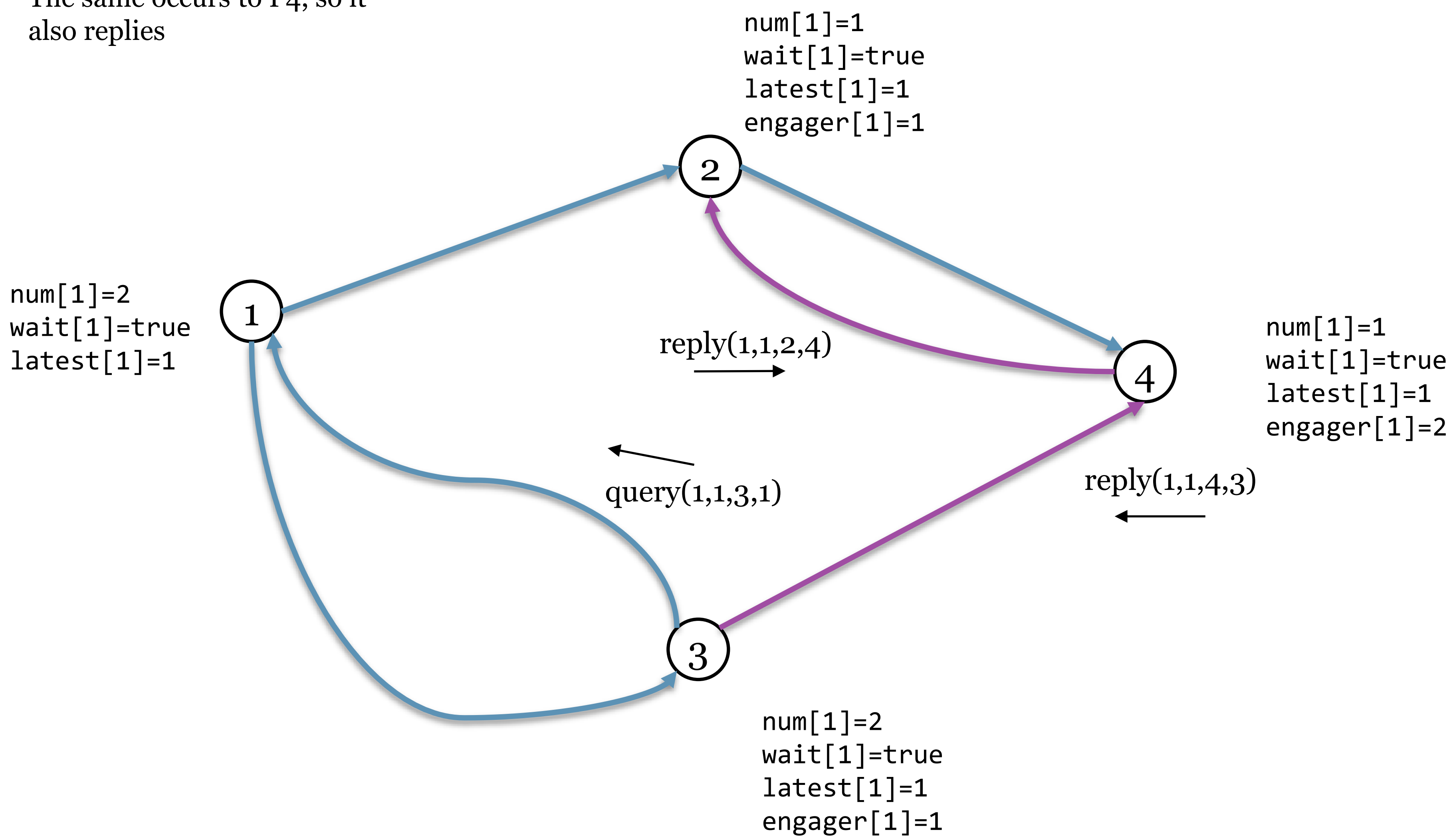
Chandy-Misra-Haas OR

P2 receives the same detection again (not engaging) from P4, so it replies



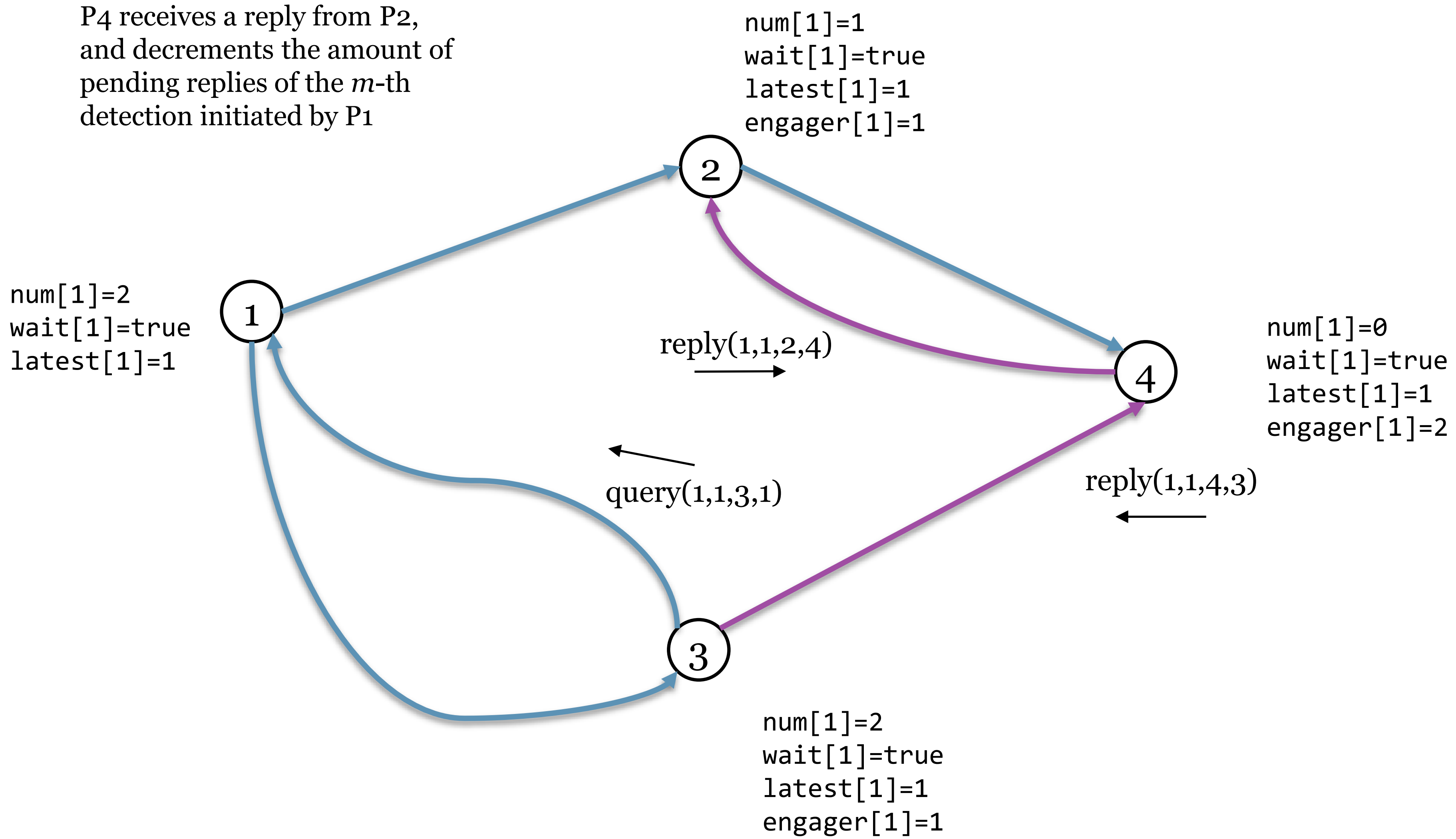
Chandy-Misra-Haas OR

The same occurs to P4, so it also replies



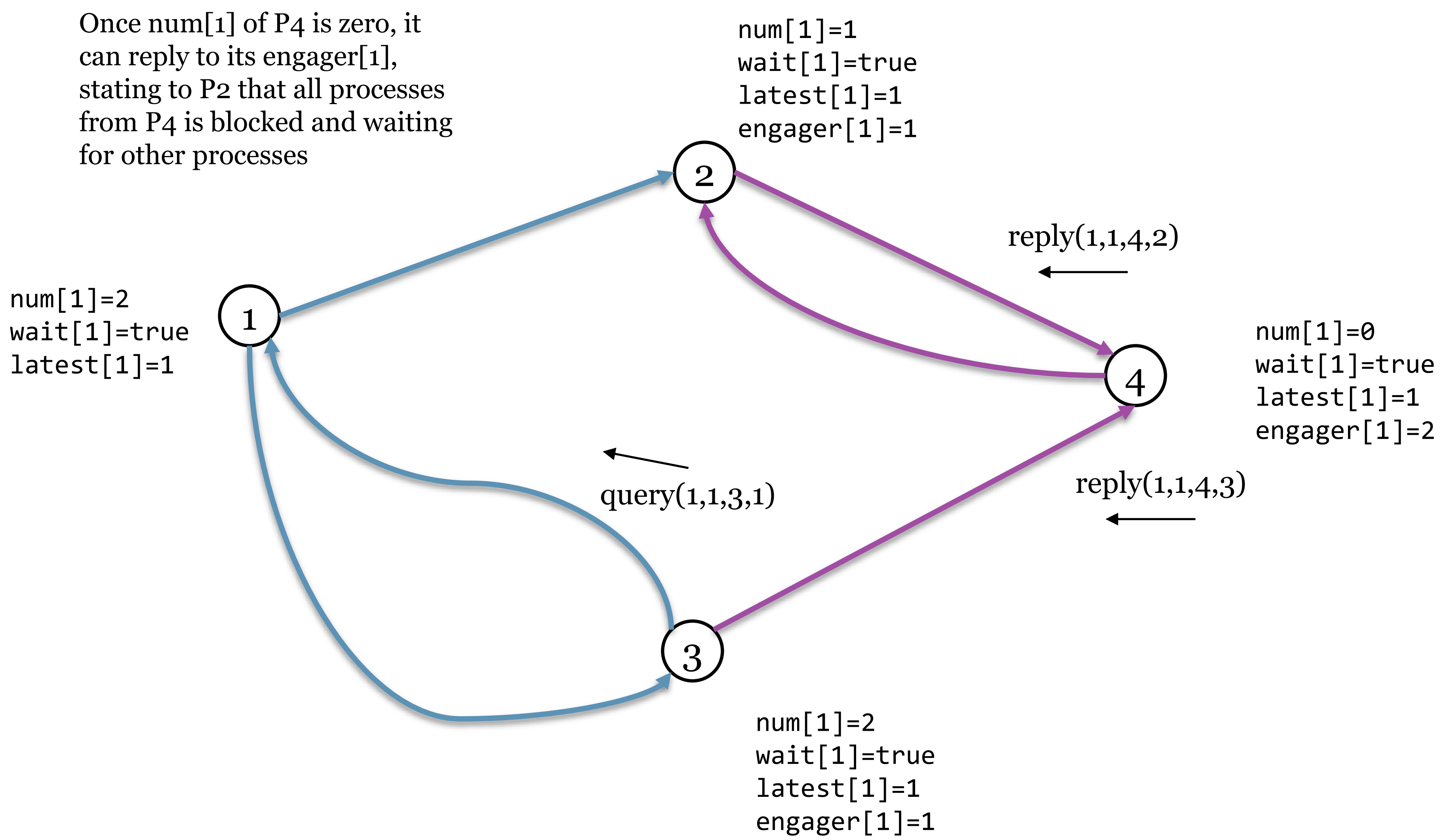
Chandy-Misra-Haas OR

P4 receives a reply from P2, and decrements the amount of pending replies of the *m*-th detection initiated by P1



Chandy-Misra-Haas OR

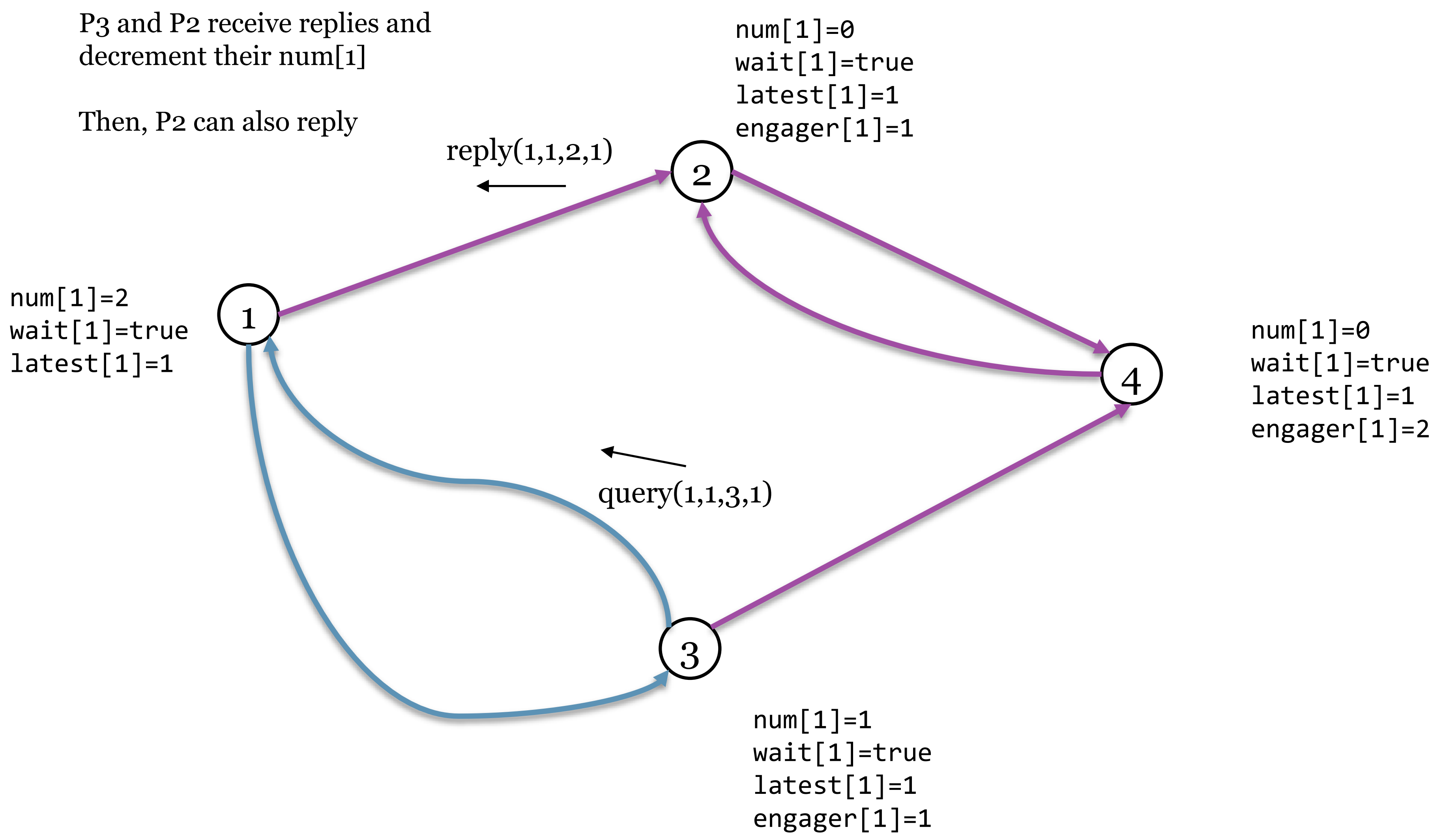
Once num[1] of P4 is zero, it can reply to its engager[1], stating to P2 that all processes from P4 is blocked and waiting for other processes



Chandy-Misra-Haas OR

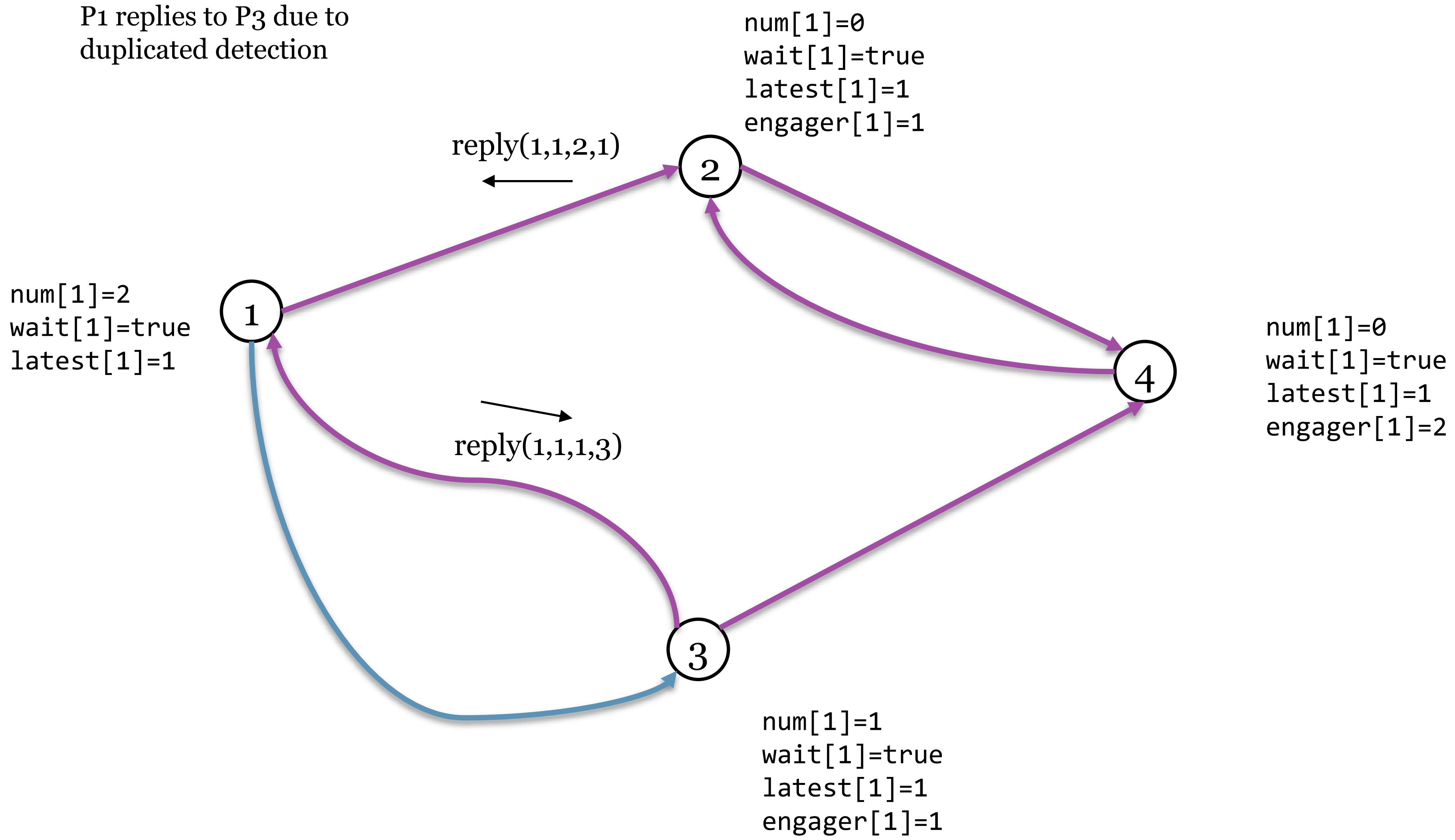
P3 and P2 receive replies and decrement their num[1]

Then, P2 can also reply



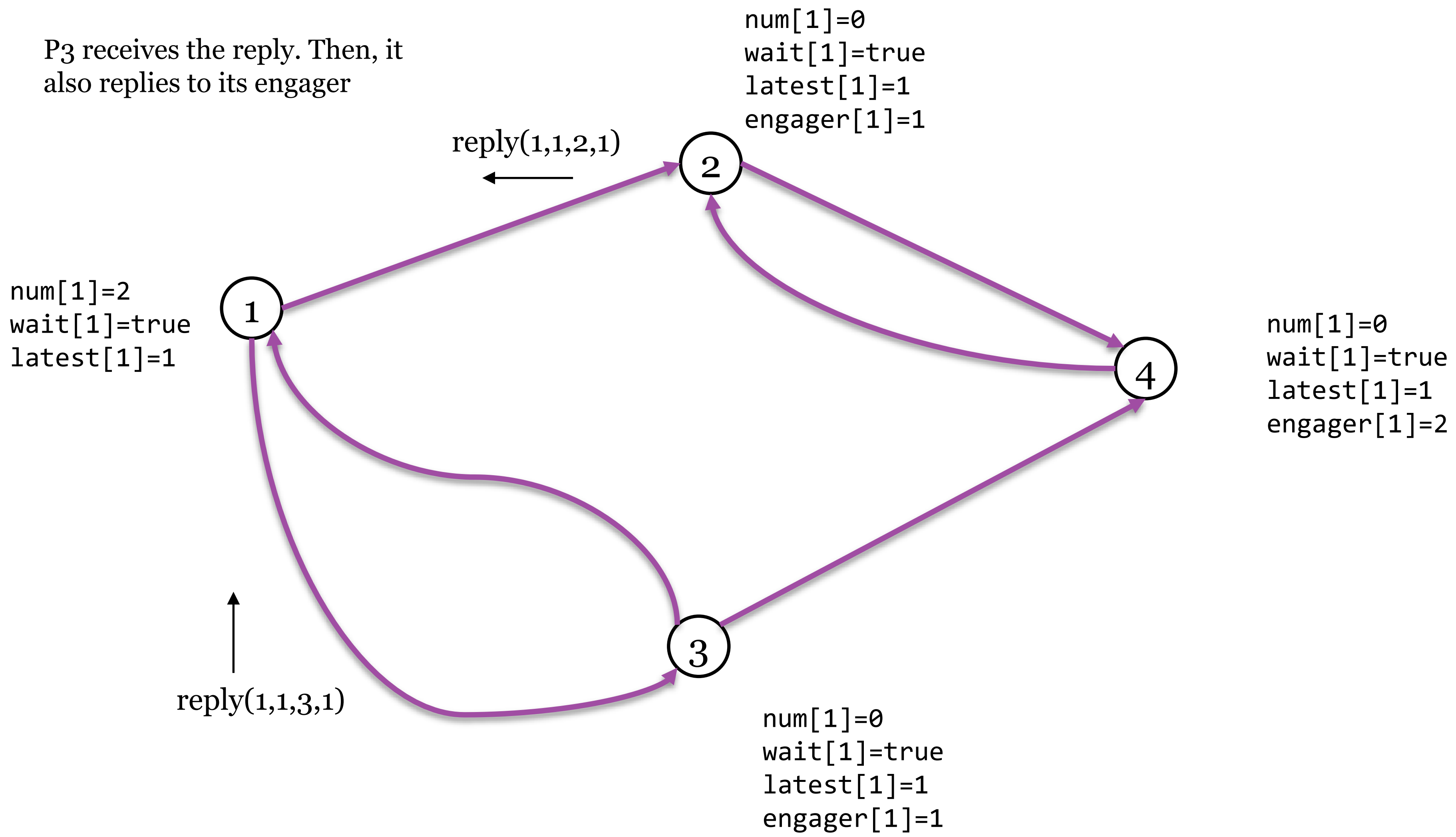
Chandy-Misra-Haas OR

P1 replies to P3 due to duplicated detection



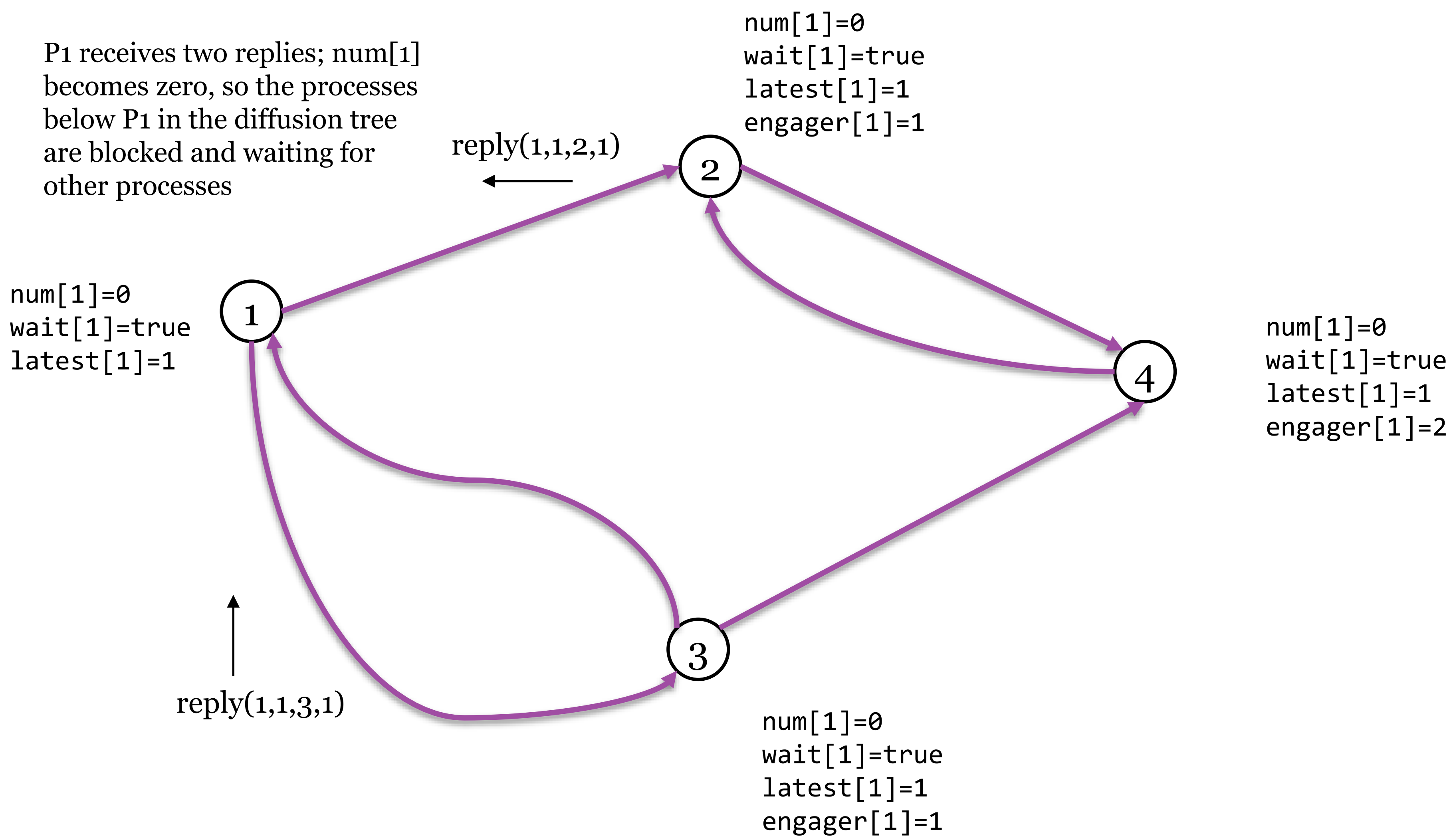
Chandy-Misra-Haas OR

P3 receives the reply. Then, it also replies to its engager



Chandy-Misra-Haas OR

P1 receives two replies; num[1] becomes zero, so the processes below P1 in the diffusion tree are blocked and waiting for other processes



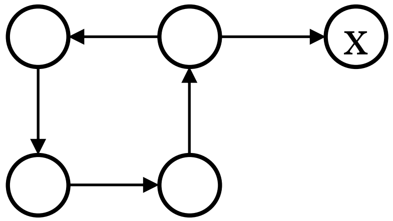
Chandy-Misra-Haas OR

- Every edge of the knot is painted twice (query and reply), resulting O(n)
- As all processes may start detection at the same time, worst case is $O(n^2)$

Fontes: Chandy-Misra-Haas [Kshemkalyani, Singhal]

Additional Material about WFG

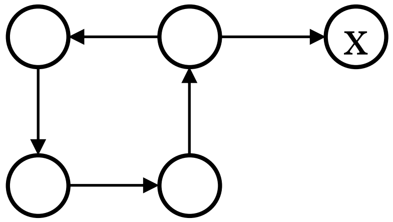
- Many other materials may consider an end (leaf) node of a WFG as a blocked node, i.e., the node is blocked and waiting for another resource even if not explicit by the WFG (e.g. the node *x* below)
- If you consider it valid, you should consider that a blocked node may have no outgoing edge, e.g., |WFx| = zero
- In this case, the description of some algorithms may be wrong, once they may expect that blocked processes have at least one outgoing edge
- One example is the Chandy-Misra-Haas algorithm for distributed detection of OR-deadlock
    - The descriptions of the algorithm in the literature do not reply to a query if an end node has no outgoing edges, e.g., the node *x* below would not reply even being blocked
    - In order to correct it, we should change the code like the one on the right



```
Blocked Pk receives a query(i,m,j,k):
   // if engaging query for Pi
   if (m>latest[i]) {
      latest[i]=m; engager[i]=j
      wait[i]=true; num[i]=|WF_k|
      $\forall Pr \in WF_k$: send query(i,m,k,r)
      if num[i]=0
         send reply(i,m,k,j)
   // if not engaging query
   } else if wait[i] and m=latest[i]
      send reply(i,m,k,j)
```

Additional Material about WFG

- Instead of changing the description of the codes and to be coherent with the definition of blocked processes, we just prefer to accept blocked processes with at least one outgoing edge in WFGs
- In other materials, if you see it and the author stated that this process is blocked, you may interpret it as a blocked subgraph in deadlock, as the example below
  - In this case, you do not have to change the original description of the algorithms



means