Deadlock Detection Algorithms

Lesson 8

Introduction to Suzuki–Kasami's Broadcast Algorithm

Core Algorithm: The Token Mechanism

The Suzuki-Kasami algorithm relies on a unique **privilege token** to enforce mutual exclusion.

- **Requesting Entry:** If a site wishes to enter the **Critical Section (CS)** but does not possess the token, it initiates a **broadcast** of a **REQUEST message** to all other sites in the network.
- **Token Transfer:** The site currently holding the token is responsible for sending it to the requesting site upon receiving the REQUEST message.
- **Delayed Transfer:** If the token holder receives a REQUEST message while it is actively **executing the CS**, the token is only sent *after* the execution of the CS has been successfully completed.

Key Design Challenges

Although the token-passing idea seems simple, two major issues must be efficiently addressed in a distributed, asynchronous environment:

1. Distinguishing Outdated vs. Current Requests

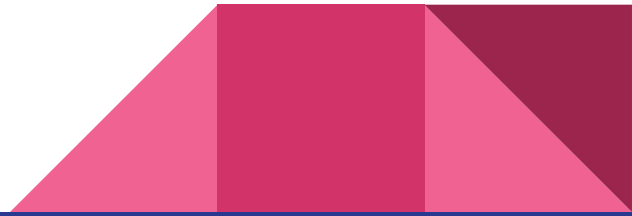
This challenge arises due to **variable message delays** in the network:

- **The Problem:** A site may receive a token REQUEST message *after* the corresponding request has already been satisfied (i.e., the sender has already used and exited the CS).
- **The Consequence:** If the token holder cannot determine the request's status, it may dispatch the token to a site that no longer needs it.
- **The Impact:** This action **does not violate correctness** (mutual exclusion is maintained), but it **seriously degrades performance** by wasting messages and increasing delay for sites genuinely requesting the token.
- **The Necessity:** Therefore, appropriate **mechanisms** must be implemented to determine if a token request message is outdated.

Determining Outstanding Requests Post-CS

When a site finishes its CS execution, it must identify a suitable successor to dispatch the token.

- **The Problem:** The status of a request is transient. When site Si receives a request from site Sj, Sj may have an outstanding request at that moment. However, Sj may have been served by another site in the interim.
- **The Complication:** After the corresponding request for the CS has been satisfied at Sj, the core issue becomes **how to efficiently inform site Sj (and all other sites)** about this satisfied status. The algorithm needs a mechanism to track which recorded requests are still active.



Operational Principles

Fairness and Priority

- After a site has executed the CS, it gives **priority** to other sites that have outstanding requests for the CS.
- The site's own **pending requests** (requests it might make in the future) are deferred until all waiting peer sites have been served.

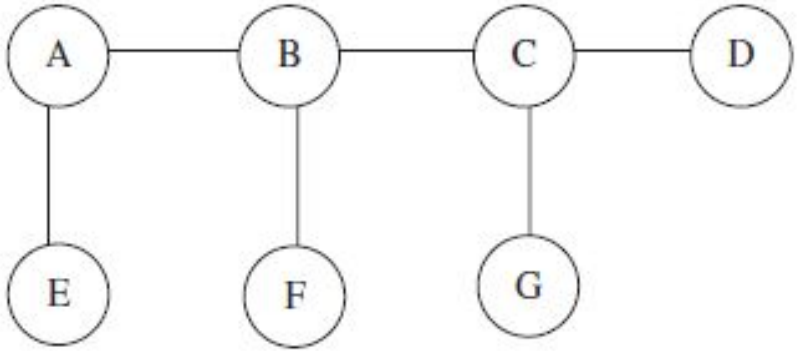
Asymmetric Behavior

- Suzuki-Kasami’s algorithm is **not symmetric** (unlike algorithms like Ricart and Agrawala's).
- **Asymmetry Definition:** A site **retains the token** even if it does not have a request for the CS.
- **Contrast:** This contradicts the symmetric principle that "no site possesses the right to access its CS when it has not been requested."

Raymond's Tree-Based Algorithm

- Raymond’s tree-based mutual exclusion algorithm organises the network into a spanning tree so that the number of messages needed for each critical section request is reduced.
- The tree structure ensures that messages follow a fixed set of paths instead of travelling across the entire network. The algorithm exchanges **only O(log N) messages under light load**, and **approximately four messages under heavy load** to execute the critical section, where N is the number of nodes in the network.
- The algorithm assumes that every message is eventually delivered, although the delay and order of arrival cannot be predicted.
- All nodes are assumed to be completely reliable.
- When the network is viewed as a graph, each node is a vertex and each communication link is an edge.
- A spanning tree connects all N nodes without cycles.
- A minimal spanning tree is one whose total cost is minimal, where the cost typically reflects link characteristics.

- The nodes are arranged as an unrooted tree.
- Messages travel along the undirected edges of this tree.
- In the following example with A, B, C, D, E, F and G, the structure forms a spanning tree and is also minimal because no other spanning tree exists for that arrangement.
- Each node keeps information only about its direct neighbours.
- For example, node C communicates only with nodes B, D and G. It does not need to store information about nodes A, E or F for the algorithm to operate.



Privilege Concept in Raymond’s Tree-Based Algorithm

- The algorithm uses the idea of a privilege, which plays a role similar to a token in other mutual exclusion methods.
- The privilege identifies the node that currently has the right to enter the critical section.
- At any given time, only one node is allowed to hold this privilege.
- If no node is requesting access to the critical section, the privilege simply stays with the node that used it most recently.

Raymond’s Algorithm - Privilege Passing Sequence

Request Creation

- A node that wants to enter the critical section checks if it already holds the privilege.
- If it does not, it creates a request for the privilege.
- This request does not go to every node in the system.
- It is forwarded only along the spanning tree toward the node that currently holds the privilege.

Path Decision Through the Tree

- Each node keeps exactly one neighbour marked as its **next hop** toward the current privilege holder.
- This next hop is the neighbour **from which the node last received a request**, or **to which it last forwarded its own request**.
- Because the tree has no cycles and each node maintains only one next hop, every request has a single direction to follow.
- When a node creates a new request, it forwards the request to this next hop.
- The next hop then forwards the request to *its* next hop, and this continues step by step through the tree until it eventually reaches the node that currently holds the privilege.

Privilege Holder Receives the Request

- When the request finally reaches the privileged node, that node examines whether it is inside its critical section. If it is idle, it prepares to send the privilege.
- If it is inside the critical section, it will send the privilege immediately after finishing.

Privilege Passing

The privilege is not broadcast. It travels back along the same spanning-tree path from which the request came.

Requesting Node Receives the Privilege

- Once the PRIVILEGE message reaches the requesting node, that node becomes the privileged node.
- It now has the right to enter the critical section.

Holding the Privilege After Completion

- When the node finishes its critical section, the privilege stays with it if no other request is waiting.
- If there are pending requests, the privilege moves to the next requester by repeating the same tree-guided steps.

Deadlock Detection in Distributed Systems

Deadlocks are recognised as a significant challenge in distributed systems. Processes request resources independently, and the order of these requests is not predetermined. A process may also request a new resource while holding others. If the sequence of allocations is not controlled, the system can reach a point where no further progress is possible.

A deadlock arises when a group of processes waits for resources held by one another. This creates a closed chain of dependencies in which none of the processes can proceed, leaving the entire group permanently blocked.

Why Deadlocks Occur More Easily in Distributed Systems

In distributed environments, resource ownership and requests are spread across multiple machines without a single centralized point having complete knowledge of the global state. Requests may arrive in any order, and dependencies can develop across different sites.

In distributed systems, a deadlock can occur when there is a cycle of dependencies: one process waits for another to release a resource, while that **other process simultaneously waits** for the first to release a different resource.

This **circular wait** creates a cycle of dependency that prevents any involved process from proceeding, causing a system-wide deadlock if not detected and resolved effectively.

Strategies for Handling Deadlocks

Deadlocks can be managed using three approaches, each dealing with the problem at a different stage.

Deadlock prevention attempts to avoid deadlocks entirely. This may be done by requiring a process to acquire all of its needed resources at once before execution begins, or by pre-empting a resource from a process that currently holds it. Although effective, prevention can reduce resource utilisation.

Advantages:

- Ensures that deadlocks never occur.
- System always remains in a safe, non-blocking state.

Disadvantages:

- Requires restrictive rules such as acquiring all resources upfront or pre-empting held resources.
- Can lower resource utilisation because processes may hold resources longer than necessary.

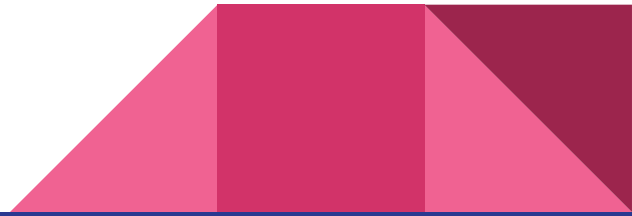
Deadlock avoidance grants a resource request only if doing so keeps the global system in a safe state. If accepting a request could lead to a possible deadlock later, the request is denied. This approach relies on the system being able to evaluate global safety.

Advantages:

- Prevents unsafe states by checking whether a request keeps the system safe.
- Reduces the chance of entering complex deadlock situations.

Disadvantages:

- Requires evaluating global safety before granting each request, adding overhead.
- Depends on having enough information to determine whether the system will remain safe.



Deadlock detection allows resource requests to proceed normally. The system is then examined to see whether a deadlock has formed. If a deadlock is identified, one of the processes involved must be aborted to release its resources. Detection avoids unnecessary restrictions during normal execution but requires periodic checking.

Advantages:

- Allows normal execution without restricting resource requests.
- Does not require predicting future resource needs or enforcing strict allocation rules.

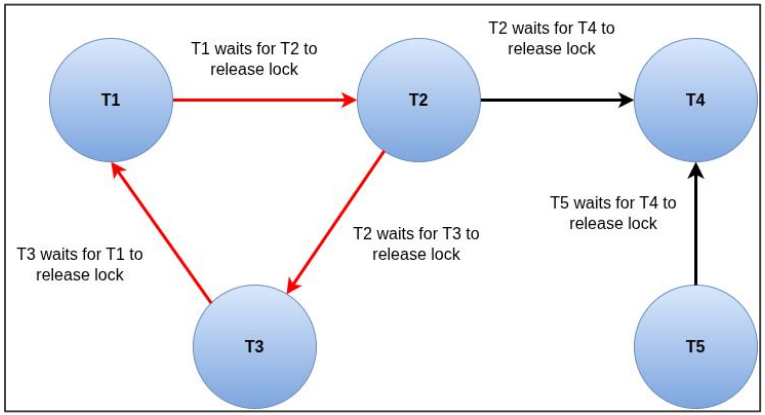
Disadvantages:

- Needs periodic monitoring to identify deadlocks.
- Deadlock resolution may require aborting a process, which leads to loss of work.

Wait-For Graph (WFG)

- A wait-for graph is used to represent the waiting relationships between processes in a distributed system.
- Each node in this graph represents a process, and each directed edge represents a situation where one process is waiting for another to release a resource.
- By modelling these dependencies visually, the system's state can be analysed for possible deadlocks.

A deadlock is indicated when the WFG contains a directed cycle or a more complex knot. Such a cycle shows that each process in the loop is waiting for another process in the same loop, and none of them can proceed. This makes the WFG a convenient structure for identifying deadlock formation.



Interpreting Dependencies in a Distributed WFG

A process at one site may wait for a process at another site to release a resource.

Another process may simultaneously be waiting for a different process at yet another site.

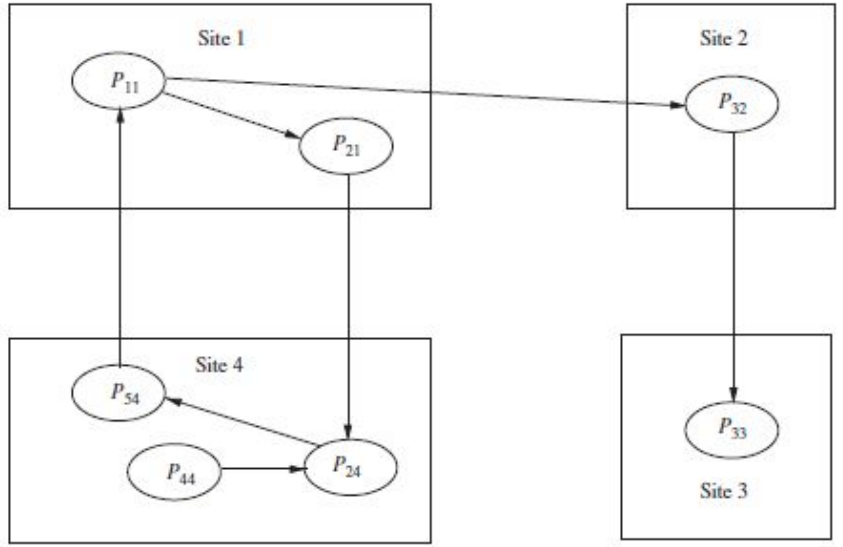
These relationships extend across the system and reveal how deadlocks can span multiple machines.

Example Possible Deadlock Scenario Across Sites

Consider an example WFG where:

- Process P11 at Site 1 waits for P21 at Site 1 and also waits for P32 at Site 2.
- Process P32 at Site 2 waits for a resource held by P33 at Site 3.
- Process P21 at Site 1 waits for P24 at Site 4.
- Processes at Site 4, such as P54 and P44, depend on P24 as well.

If P33 later begins waiting for P24, a circular dependency involving multiple sites can be formed. Whether this results in a deadlock depends on the request model being used.

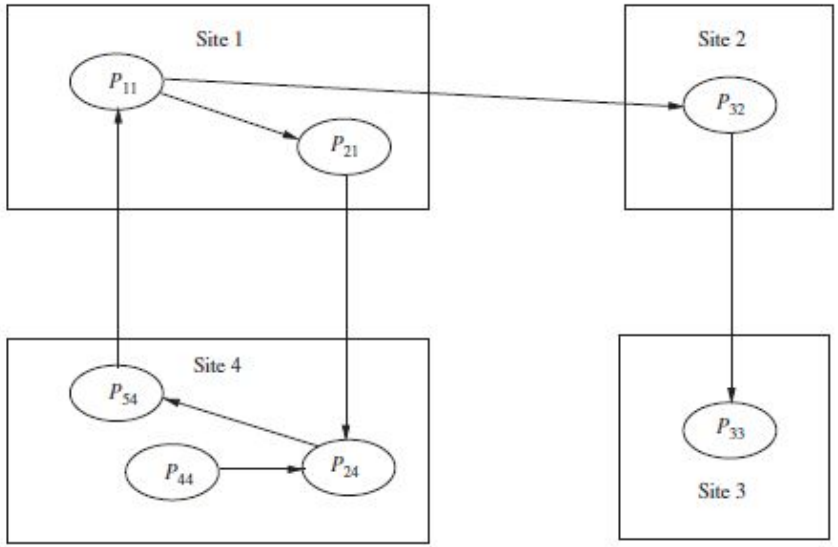


Example Existing Deadlock Scenario Across Sites

Consider the following in the WFG shown where:

- P11 at Site 1 waits for P21 at Site 1.
- P21 at Site 1 waits for P24 at Site 4.
- P24 at Site 4 waits for P54 at Site 4.
- P54 at Site 4 waits for P11 at Site 1.

These directed edges form a closed cycle P11 → P21 → P24 → P54 → P11. Because every process in this set is waiting for another process in the same set, the system is already in a deadlocked state, and this deadlock spans Site 1 and Site 4.



Models of Deadlocks: Distributed Systems

Resource Request Hierarchy

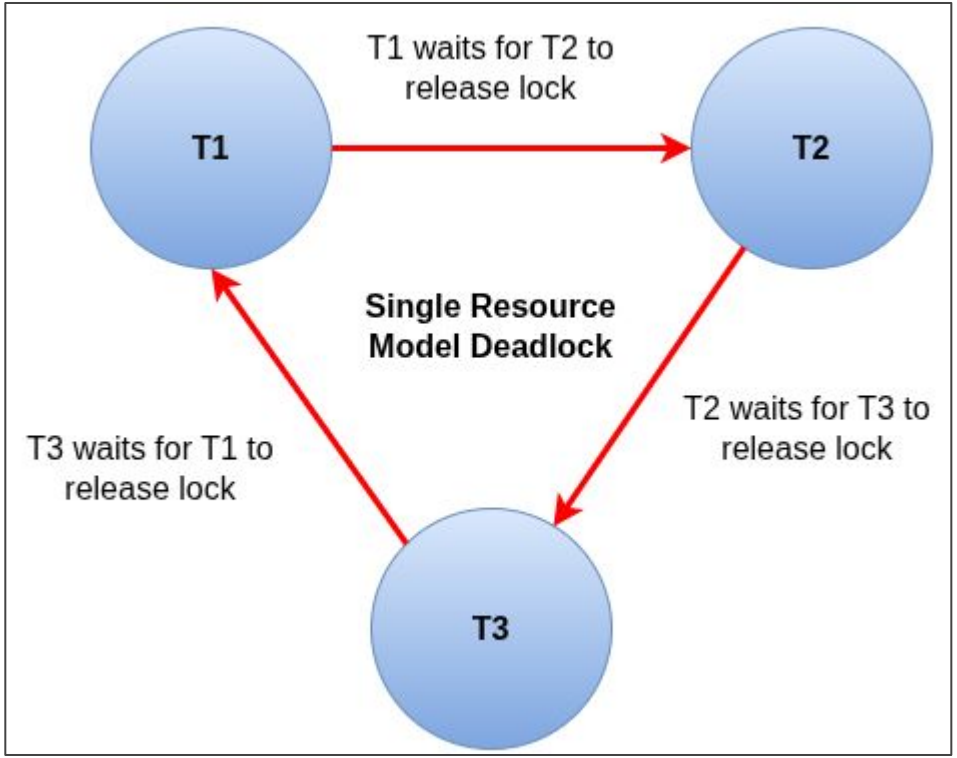
Distributed systems must manage diverse resource requests; a process might require a **single resource** or a **combination of resources** for its execution.

- A **hierarchy of request models** is introduced to classify these variations, ranging from **very restricted forms** to those with **no restrictions whatsoever**.
- This hierarchy is used to categorize **deadlock detection algorithms** based on the **complexity of the resource requests** they permit.

The Single-Resource Model

The single-resource model represents the **simplest** resource request structure in a distributed system.

- **Constraint:** A process can have at most **one outstanding request** for only **one unit of a resource**.
- **Wait-For Graph (WFG) Property:** Since a process can only have one outstanding request, the maximum **out-degree** of a node (process) in the WFG can be **1**.
- **Deadlock Detection:** In this restricted model, the **presence of a cycle** in the Wait-For Graph (WFG) **shall indicate that there is a deadlock**.



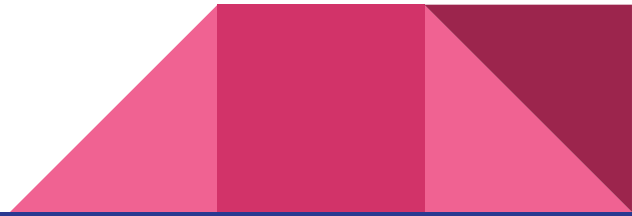
The AND Model

Request Constraint: A process can request **more than one resource simultaneously**.

Satisfaction Condition: The request is satisfied **only after all** the requested resources are granted to the process (hence the 'AND' logic).

Resource Location: The requested resources **may exist at different locations** (sites) across the distributed system.

Wait-For Graph (WFG) Property: The out-degree of a node (process) in the WFG for the AND model can be **more than 1**. Each node in the WFG is called an **AND node**.



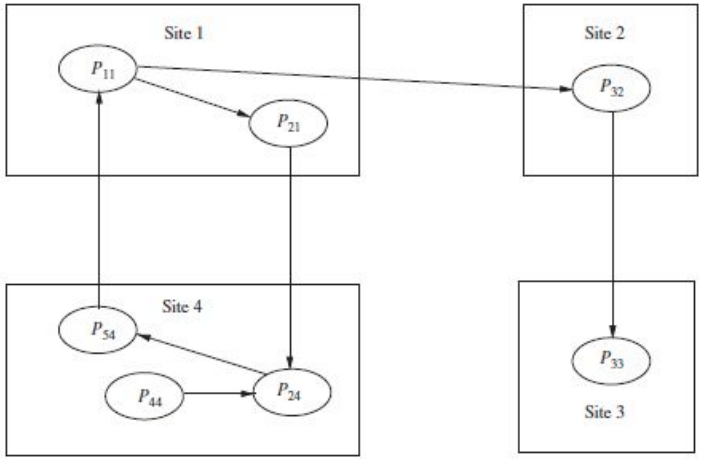
Deadlock Implications in the AND Model

Cycle Implies Deadlock: If a cycle is detected in the WFG, it **implies a deadlock**.

- *Example:* Process **P11** has two outstanding resource requests.

  In the AND model, **P11** will become active from its idle state **only after both** resources are granted.

  The cycle **P11 → P21 → P24 → P54 → P11** corresponds to a deadlock situation.



Deadlock Does NOT Imply Cycle: The reverse is **not true** (not vice versa). A process may be deadlocked **even if it is not part of a cycle**.

- *Example:* Process P44 might not be part of any cycle but is deadlocked because it is dependent on another process (like P24) which is already part of a deadlock.

The OR Model

Request Constraint: In the OR model, a process can make a request for **numerous resources simultaneously**.

Satisfaction Condition: The request is satisfied if **any one** of the requested resources is granted to the process (hence the 'OR' logic).

Resource Location: The requested resources **may exist at different locations** (sites).

Wait-For Graph (WFG) Property: If all requests in the WFG are OR requests, the processes are referred to as **OR nodes**.

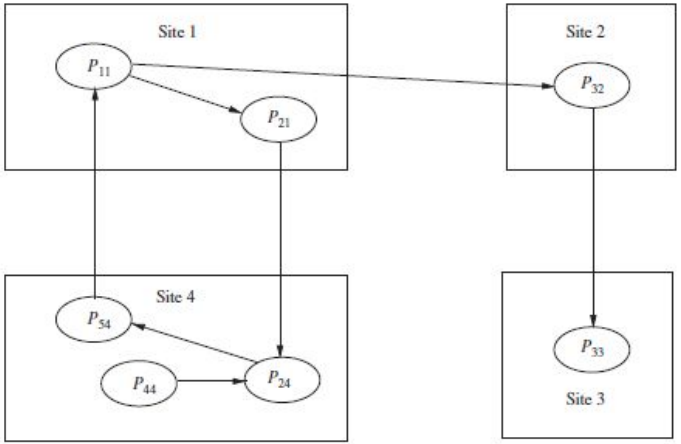
Deadlock Implications in the OR Model

Cycle Does Not Imply Deadlock: The **presence of a cycle** in the WFG of an OR model **does not imply a deadlock**.

Example Scenario (Non-Deadlock):

P11's Dependencies (Wait-For Edges)

- Process P11 is an OR node with two simultaneous outgoing dependencies:
- P11 is waiting for a resource held by P21.
  P11 is also waiting for a resource held by P32.



Cycle Resolution Chain

Even if a cycle exists it is not a deadlock. P11 can be unblocked:

1. If an external event occurs when P33 releases its resource.
2. This release allows P32 to run. After completing its work, P32 releases the resource it holds.
3. The release from P32 satisfies the dependency P11 → P32.

Advanced Deadlock Models

Beyond the Single-Resource, AND, and OR models, further generalizations exist to handle highly complex resource request combinations.

The AND-OR Model

- The **AND-OR model** is a generalization of the previous two models, allowing a request to specify **any combination of AND and OR** logic in the resource request.
- *Example:* A request might be of the form **x AND (y OR z)**.
- **Location:** The requested resources may exist at **different locations**.
- **Detection Challenge:** To detect deadlocks in the AND-OR model, **there is no familiar construct of graph theory using the Wait-For Graph (WFG)** due to the high complexity.
- **Deadlock Property:** Since a deadlock is a **stable property** (once it exists, it does not go away by itself), this property can be exploited and a deadlock can be detected by repeated application of the test for the OR-model deadlock. However, this is a very **inefficient strategy**.

The (p/q) Model (P-out-of-Q Model)

- The **(p/q) model** (called the **P-out-of-Q model**) is another form of the AND-OR model.
- **Request Condition:** This model allows a request to obtain **any k available resources** from a pool of n resources.

Expressive Power (Equivalence)

- The **(p/q) model** is another form of the AND-OR model.
- The two models are the **same in expressive power**.
- This means any request that can be formulated using the complex AND/OR logic can also be formulated using the simpler **P-out-of-Q** structure, and **vice-versa**.

Compactness

- The (p/q) model lends itself to a much more **compact formation of a request**.
- Instead of writing out a long, complex boolean expression (e.g., (R1 AND R2) OR (R3 AND R4), the (p/q) model allows stating the requirement simply as needing p resources from a pool of q resources.

Unrestricted Model

- In the unrestricted model, no assumptions are made about how processes request resources.
- There is no limitation on the structure or form of these requests, making it the most general model.

Assumption

- The unrestricted model makes only one assumption: once a deadlock forms, it does not disappear on its own. This means the deadlock is stable and will continue to exist until an external action resolves it.
- Because nothing else is assumed about how processes request resources or how the system communicates, the deadlock property can be examined separately from the details of the underlying distributed system.
- In other words, the correctness of deadlock detection does not depend on whether the system uses message passing, synchronous communication, or any other mechanism.

Algorithms written for this model are general enough to detect any stable condition, not just deadlocks. However, this generality comes with a cost. Since no assumptions are made about the behaviour of the system, these algorithms must handle every possible situation, which introduces significant overhead. As a result, they tend to be more useful for theoretical study than for practical distributed systems.

Chandy-Misra-Haas Algorithm for the AND Model

- The Chandy-Misra-Haas algorithm detects deadlocks in a distributed system by **following wait-for edges using a special message called a probe**.
- It works under the **AND request model**, where a process must wait for *all* requested resources.
- The algorithm detects a deadlock **when a probe message returns to the process that initiated it**.

Probe Format

A probe is a triple:

(i, j, k)

- **i** → the *initiator* (the process that started the deadlock detection)
- **j** → the *current sender* of the probe
- **k** → the process that **j is waiting for**

This triple is carried unchanged by each process (except j and k fields which update per hop).

When a Process Sends a Probe

A process **Pj** starts deadlock detection when:

- Pj becomes **blocked**, AND
- Pj is not currently participating in any other ongoing detection effort.

It creates and sends a probe:

(i = j, j = j, k = process Pj is waiting for)

This probe travels along the edges of the **global wait-for graph (WFG)**.

How the Probe Moves Through the System

At each process receiving the probe:

Let the probe be: **(i, j, k)** and suppose it arrives at process **Pk**.

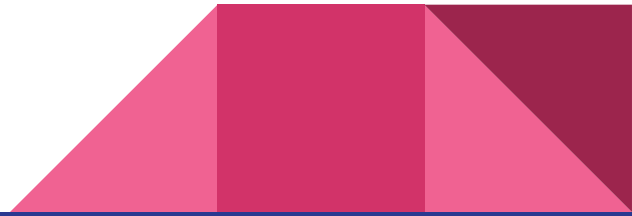
- If **Pk is not waiting for anyone**, the probe stops.
- If **Pk *is* waiting for some process Pm**, Pk forwards the probe as:

(i, j = k, k = m)

meaning:

- initiator is still **i**,
- current sender is now **k**,
- next dependent process is **m**.

This continues hop-by-hop along the WFG.



Deadlock Detection Condition

A **deadlock is detected** when the probe returns to the initiator.

That is:

If a probe (i, j, k) arrives at process i → DEADLOCK

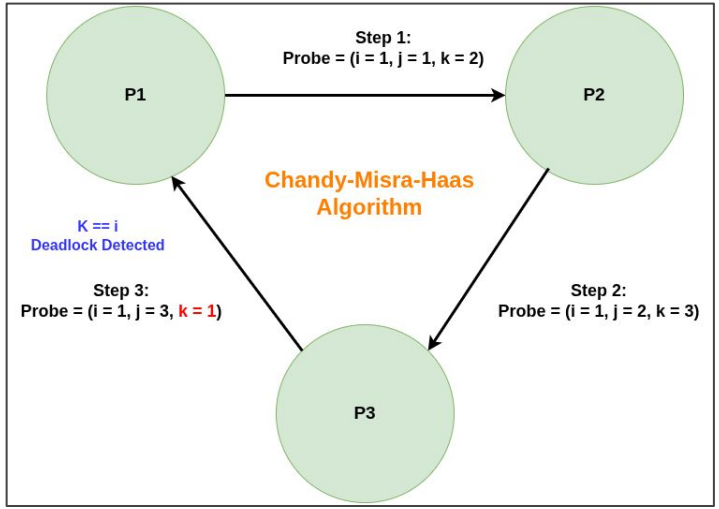
For the cycle:

- P1 waits for P2
- P2 waits for P3
- P3 waits for P1

The probes travel:

- **(1,1,2)** → from P1 to P2
- **(1,2,3)** → from P2 to P3
- **(1,3,1)** → from P3 back to P1

Since **k = i = 1**, deadlock is confirmed.



Chandy-Misra-Haas Algorithm for the OR Model

This version of the algorithm is used for **OR-type resource requests**, where a blocked process becomes free as soon as *any one* of its outstanding requests is satisfied.

Deadlock detection is carried out using a **diffusion computation**, which spreads through the wait-for dependencies and then gathers replies.

When a Deadlock Check Begins

- A **blocked process** starts deadlock detection.
  It sends **query messages** to every process in its *dependent set* (meaning every process it is currently waiting for).
- Only blocked processes participate.
- If an **active** (no blocked) process receives a query or reply message, it simply discards it.

Message Types

Two message types are used during diffusion computation:

- **query(i, j, k):** part of a detection started by process ***i***; sent from process ***j*** to process ***k***
- **reply(i, j, k):** reply to a query belonging to detection initiated by ***i***; sent from process ***j*** to process ***k***

When a Blocked Process Receives a Query(i, j, k)

When a blocked process **Pk** receives a query for a detection initiated by process **Pi**, it performs one of the following:

1. If this is the first query for this detection (the engaging query):

- The *first* query message that a blocked process receives for a particular deadlock detection initiated by Pi is called the **engaging query**.
- If it is an engaging query, it sends **query messages** to *every* process in its own dependent set.
- It sets a local variable **numk** to the number of query messages it just sent.
- It notes that it has now received its *engaging query* for detection **i**.

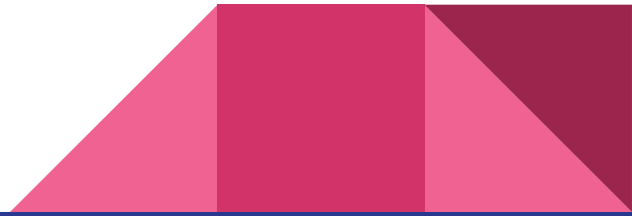
This marks the beginning of Pk's participation in this particular diffusion computation.

2. If this is *not* the engaging query:

Pk immediately sends a **reply(i, j, k)** back to the sender *j*, *but only if*:

- Pk has remained **continuously blocked** since receiving the engaging query.

If Pk was unblocked at any moment after receiving the engaging query, the query is discarded. This prevents false positives.



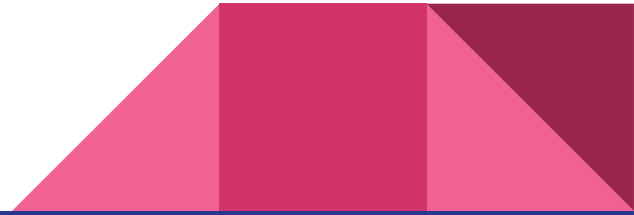
Local State Maintained by Each Blocked Process

Each process **Pk** maintains:

- **waitki**: a boolean that indicates whether Pk has remained continuously blocked since receiving its engaging query.
- **numk:** the number of outstanding query messages that Pk has sent and still expects replies for.

Whenever Pk receives a **reply(i, j, k)** message:

- It decrements **numk**, but only if **waitki** is still true.

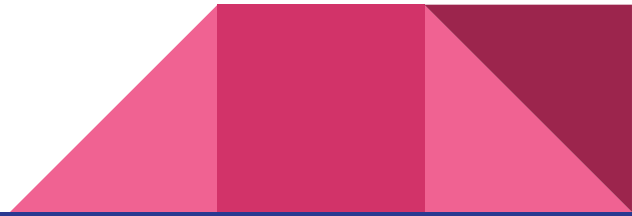


When a Process Sends Its Own Reply

A blocked process **Pk** sends a reply for its engaging query **only when:**

- It has received replies to **all** query messages it originally sent out for this detection.

This means **numk becomes zero**.



Deadlock Detection

The **initiator Pi** detects a deadlock when:

- It receives reply messages for **every** query message it originally sent.

This means the entire diffusion computation has “echoed back” to Pi, indicating that the dependency never escaped the cycle, meaning all dependent paths eventually lead back to blocked processes.

Thus, Pi declares deadlock.