DISTRIBUTED COMPUTING
MODULE – 3

Distributed mutual exclusion algorithms

Mutual exclusion is a fundamental problem in distributed computing systems. Mutual exclusion ensures that concurrent access of processes to a shared resource or data is serialized, that is, executed in a mutually exclusive manner. Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion. The decision as to which process is allowed access to the CS next is arrived at by message passing, in which each process learns about the state of all other processes in some consistent way.
There are three basic approaches for implementing distributed mutual exclusion:
1. Token-based approach.
2. Non-token-based approach.
3. Quorum-based approach.

In the token-based approach, a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over. Mutual exclusion is ensured because the token is unique. The algorithms based on this approach essentially differ in the way a site carries out the search for the token. In the non-token-based approach, two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next. A site enters the critical section (CS) when an assertion, defined on its local variables, becomes true. Mutual exclusion is enforced because the assertion becomes true only at one site at any given
time.

In the quorum-based approach, each site requests permission to execute the CS from a subset of sites (called a quorum). The quorums are formed in such a way that when two sites concurrently request access to the CS, at least one site receives both the requests and this site is responsible to make sure that only one request executes the CS at any time.

System Model

- The system consists of N sites, S_1, S_2, ..., S_N.
- We assume that a single process is running on each site. The process at site S_i is denoted by p_i .
- A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle).
- In the 'requesting the CS' state, the site is blocked and can not make further requests for the CS.
- In the 'idle' state, the site is executing outside the CS. In token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS (called the idle token state).
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

Requirements of Mutual Exclusion Algorithms

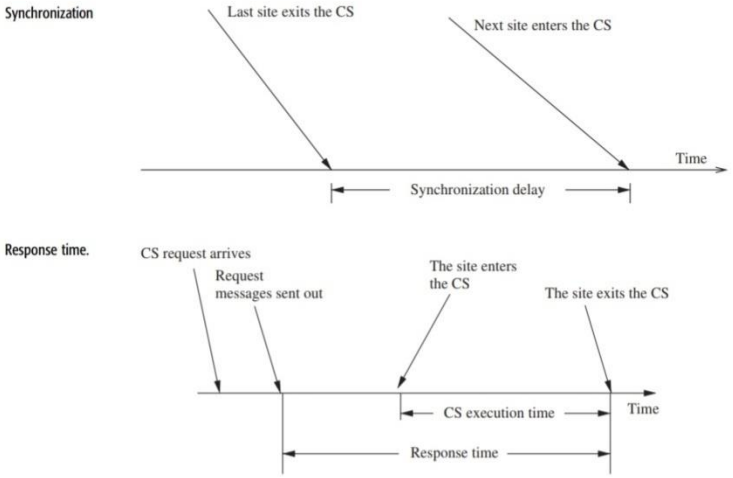
- Safety Property: At any instant, only one process can execute the critical section.
- Liveness Property: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
- Fairness: Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

Performance Metrics

The performance is generally measured by the following four metrics:

- Message complexity: The number of messages required per CS execution by a site.

- Synchronization delay: After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 1).



- Response time: The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).
- System throughput: The rate at which the system executes requests for the CS.

system throughput=1/(SD+E) where SD is the synchronization delay and E is the average critical section execution time

Low and High Load Performance:

We often study the performance of mutual exclusion algorithms under two special loading conditions, viz., "low load" and "high load". The load is determined by the arrival rate of CS execution requests. Under low load conditions, there is seldom more than one request for the critical section present in the system simultaneously. Under heavy load conditions, there is always a pending request for critical section at a site.

Lamport's algorithm

Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on $request\_queue_i$. ((ts_i, i) denotes the timestamp of the request.)
- When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on $request\_queue_j$ and returns a timestamped REPLY message to S_i.

Executing the critical section

Site S_i enters the CS when the following two conditions hold:

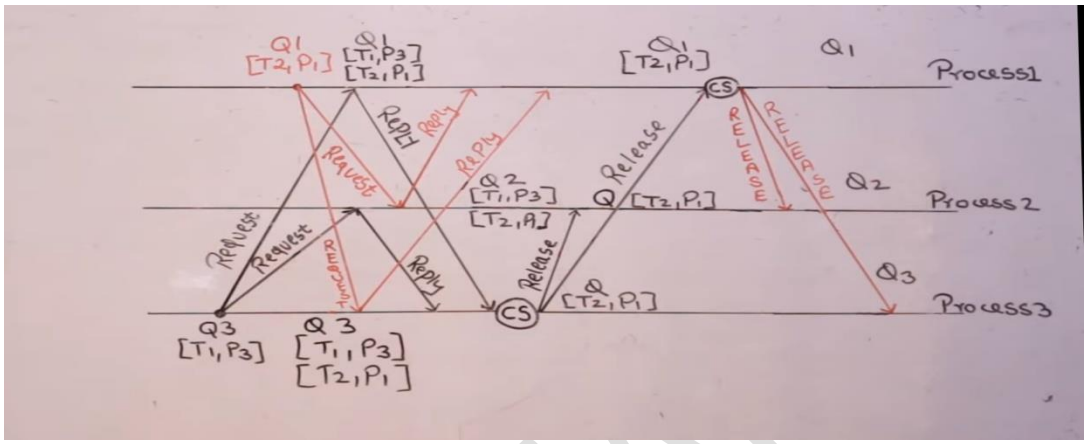
L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

L2: S_i's request is at the top of $request\_queue_i$.

Releasing the critical section

- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp. The algorithm executes CS requests in the increasing order of timestamps. Every site Si keeps a queue, request_queuei, which contains mutual exclusion requests ordered by their timestamps. This algorithm requires communication channels to deliver messages in FIFO order. When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. Clearly, when a site receives a REQUEST, REPLY, or RELEASE message, it updates its clock using the timestamp in the message.



Example explanation video link - https://youtu.be/QHb14nAUVSs

Correctness

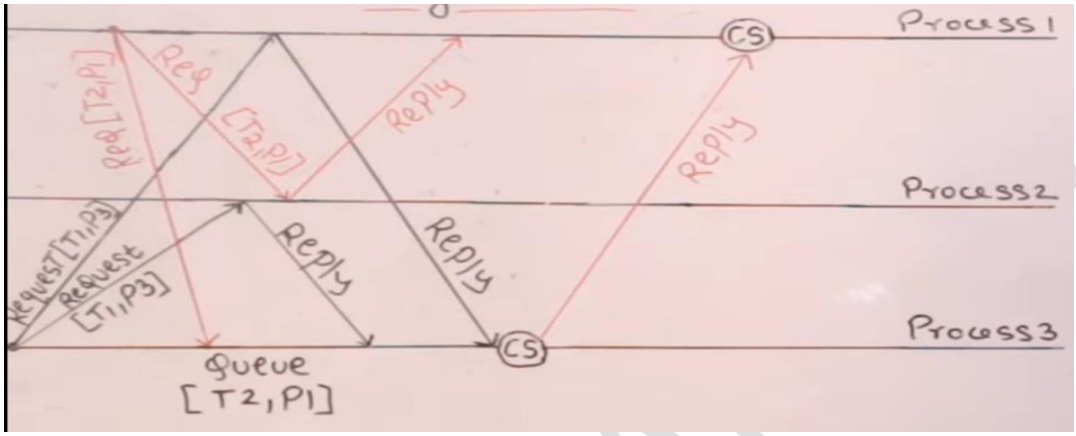
Lamport's algorithm achieves mutual exclusion. Proof is by contradiction. Suppose two sites Si and Sj are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites concurrently. This implies that at some instant in time, say t, both Si and Sj have their own requests at the top of their request_queues and condition L1 holds at them. Without loss of generality, assume that Si's request has smaller timestamp than the request of Sj . From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of Si must be present in request_queuej when Sj was executing its CS. This implies that Sj's own request is at the top of its own request_queue when a smaller timestamp request, Si's request, is present in the request_queuej – a contradiction! Hence, Lamport's algorithm achieves mutual exclusion.

Ricart–Agrawala algorithm

- Requesting critical section
  - Si sends time stamped REQUEST message
  - Sj sends REPLY to Si, if
    - Sj is not requesting nor executing CS
    - If Sj is requesting CS and Si's time stamp is smaller than its own request.
    - Request is deferred otherwise.
- Executing CS: after it has received REPLY from all sites in its request set.
- Releasing CS: Send REPLY to all deferred requests. i.e., a site's REPLY messages are blocked only by sites with smaller time stamps

Correctness

Ricart–Agrawala algorithm achieves mutual exclusion. Proof is by contradiction. Suppose two sites Si and Sj are executing the CS concurrently and Si's request has higher priority (i.e., smaller timestamp) than the request of Sj . Clearly, Si received Sj's request after it has made its own request. (Otherwise, Si's request will have lower priority.) Thus, Sj can concurrently execute the CS with Si only if Si returns a REPLY to Sj (in response to Sj's request) before Si exits the CS. However, this is impossible because Sj's request has lower priority. Therefore, the Ricart–Agrawala algorithm achieves mutual exclusion.



Example explained video- https://youtu.be/zb-8pClSjSU

Performance

For each CS execution, the Ricart–Agrawala algorithm requires (N – 1) REQUEST messages and (N – 1) REPLY messages. Thus, it requires $2(N-1)$ messages per CS execution. The synchronization delay in the algorithm is T.

Quorum-based mutual exclusion algorithms respresented a departure from the trend in the following two ways:

1. A site does not request permission from all other sites, but only from a subset of the sites. This is a radically different approach as compared to the Lamport and Ricart–Agrawala algorithms, where all sites participate in conflict resolution of all other sites. In quorum-based mutual exclusion algorithm, the request set of sites are chosen such that $\forall i \forall j : 1 \leq I, j \leq N :: R_i \cap R_j \neq \phi$ Consequently, every pair of sites has a site which mediates conflicts between that pair.
2. In quorum-based mutual exclusion algorithm, a site can send out only one REPLY message at any time. A site can send a REPLY message only after it has received a RELEASE message for the previous REPLY message. Therefore, a site Si *locks* all the sites in Ri in exclusive mode before executing its CS.

Quorum-based mutual exclusion algorithms significantly reduce the message complexity of invoking mutual exclusion by having sites ask permission from only a subset of sites. Since these algorithms are based on the notion of "Coteries" and "Quorums," we first describe the idea of coteries and quorums. A coterie *C* is defined as a set of sets, where each set $g \in C$ is called a quorum. The following properties hold for quorums in a coterie:

- **Intersection property** For every quorum g, $h \in C$, $g \cap h = \phi$.

For example, sets {1,2,3}, {2,5,7}, and {5,7,9} cannot be quorums in a coterie because the first and third sets do not have a common element.

- **Minimality property** There should be no quorums g, h in coterie C such that $g \supseteq h$. For example, sets {1,2,3} and {1,3} cannot be quorums in a coterie because the first set is a superset of the second.

Maekawa's algorithm

Requesting the critical section:

(a) A site S_i requests access to the CS by sending REQUEST(i) messages to all sites in its request set R_i.

(b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST(i) for later consideration.

Executing the critical section:

(c) Site S_i executes the CS only after it has received a REPLY message from every site in R_i.

Releasing the critical section:

(d) After the execution of the CS is over, site S_i sends a RELEASE(i) message to every site in R_i.

(e) When a site S_j receives a RELEASE(i) message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

Example explained video- https://youtu.be/zb-8pClSjSU

Correctness

Maekawa's algorithm achieves mutual exclusion. Proof is by contradiction. Suppose two sites Si and Sj are concurrently executing the CS. This means site Si received a REPLY message from all sites in Ri and concurrently site Sj was able to receive a REPLY message from all sites in Rj. If Ri ∩ Rj = {Sk}, then site Sk must have sent REPLY messages to both Si and Sj concurrently, which is a contradiction.

Performance

Note that the size of a request set is √N. Therefore, an execution of the CS requires √N REQUEST, √N REPLY, and √N RELEASE messages, resulting in 3√N messages per CS execution. Synchronization delay in this algorithm is 2T. This is because after a site Si exits the CS, it first releases all the sites in Ri and then one of those sites sends a REPLY message to the next site that executes the CS. Thus, two sequential message transfers are required between two successive CS executions.

SUZUKI-KASAMI'S BROADCAST ALGORITHM

Example explained video- https://youtu.be/0jlpJi0kfOI

Requesting the critical section:

(a) If requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(i, sn) message to all other sites. ("sn" is the updated value of $RN_i[i]$.)

(b) When a site S_j receives this message, it sets $RN_j[i]$ to $max(RN_j[i], sn)$. If S_j has the idle token, then it sends the token to S_i if $RN_j[i] = LN[i] + 1$.

Executing the critical section:

(c) Site S_i executes the CS after it has received the token.

Releasing the critical section: Having finished the execution of the CS, site S_i takes the following actions:

(d) It sets $LN[i]$ element of the token array equal to $RN_i[i]$.

(e) For every site S_j whose i.d. is not in the token queue, it appends its i.d. to the token queue if $RN_i[j] = LN[j] + 1$.

(f) If the token queue is nonempty after the above update, S_i deletes the top site i.d. from the token queue and sends the token to the site indicated by the i.d.

In Suzuki–Kasami's algorithm, if a site that wants to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all other sites. A site that possesses the token sends it to the requesting

site upon the receipt of its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.
Although the basic idea underlying this algorithm may sound rather simple, there are two design issues that must be efficiently addressed:
How to distinguishing an outdated REQUEST message from a current REQUEST message Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied. If a site cannot determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it. This will not violate the correctness, however, but it may seriously degrade the performance by wasting messages and increasing the delay at sites that are genuinely requesting the token. Therefore, appropriate mechanisms should implemented to determine if a token request message is outdateded.
How to determine which site has an outstanding request for the CS After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them. The problem is complicated because when a site Si receives a token request message from a site Sj , site Sj may have an outstanding request for the CS. However, after the corresponding request for the CS has been satisfied at Sj , an issue is how to inform site Si (and all other sites) efficiently about it.

Correctness
Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

Performance
The beauty of the Suzuki–Kasami algorithm lies in its simplicity and efficiency. No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request. If a site does not hold the token when it makes a request, the algorithm requires N messages to obtain the token. The synchronization delay in this algorithm is 0 or T.

Deadlock detection in distributed systems

- A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set.
- Deadlocks can be dealt with using any one of the following three strategies: deadlock prevention, deadlock avoidance, and deadlock detection.
- Deadlock prevention is commonly achieved by either having a process acquire all the needed resources simultaneously before it begins execution or by pre-empting a process that holds the needed resource.
- In the deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system is safe.
- Deadlock detection requires an examination of the status of the process–resources interaction for the presence of a deadlock condition.
- To resolve the deadlock, we have to abort a deadlocked process.

System model
A distributed system consists of a set of processors that are connected by a communication network. The communication delay is finite but unpredictable. A distributed program is composed of a set of n asynchronous processes *P*1, *P*2, , *P*i, , *P*n that communicate by message passing over the communication network. Without loss of generality we assume that each process is running on a different processor. The processors do not share a common global memory and communicate solely by passing messages over the communication network. There is no physical global clock in the system to which processes have instantaneous access. The communication medium may deliver messages out of order, messages may be lost, garbled, or duplicated due to timeout and retransmission, processors may fail, and communication links may go down. The system can be modeled as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels.
We make the following assumptions:

- The systems have only reusable resources.

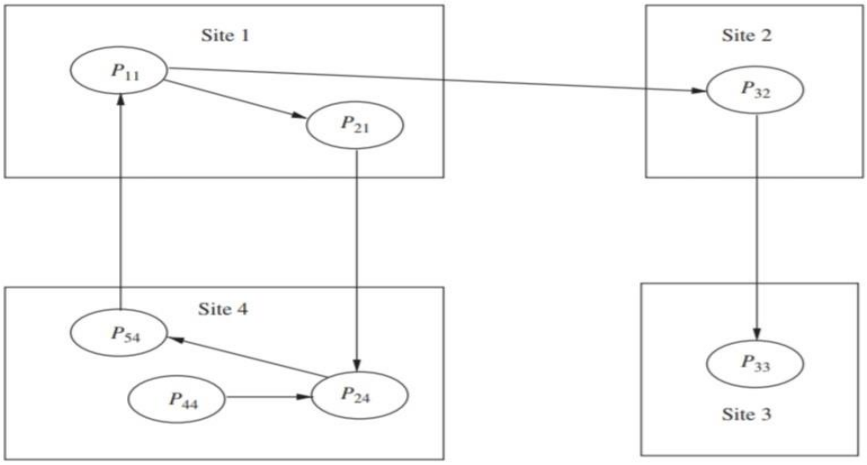
- Processes are allowed to make only exclusive access to resources.
- There is only one copy of each resource.

A process can be in two states, *running* or *blocked*. In the running state (also called *active* state), a process has all the needed resources and is either executing or is ready for execution. In the blocked state, a process is waiting to acquire some resource.

Wait-for graph (WFG)

In distributed systems, the state of the system can be modeled by directed graph, called a *wait-for graph* (WFG). In a WFG, nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource. A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

Figure shows a WFG, where process P11 of site 1 has an edge to process P21 of site 1 and an edge to process P32 of site 2. Process P32 of site 2 is waiting for a resource that is currently held by process P33 of site 3. At the same time process P21 at site 1 is waiting on process P24 at site 4 to release a resource, and so on. If P33 starts waiting on process P24, then processes in the WFG are involved in a deadlock depending upon the request model.



Deadlock Handling Strategies

- There are three strategies for handling deadlocks, viz., deadlock prevention, deadlock avoidance, and deadlock detection.
- Handling of deadlock becomes highly complicated in distributed systems because no site has accurate knowledge of the current state of the system and because every inter-site communication involves a finite and unpredictable delay.
- Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource.
- This approach is highly inefficient and impractical in distributed systems.
- In deadlock avoidance approach to distributed systems, a resource is granted to a process if the resulting global system state is safe (note that a global state includes all the processes and resources of the distributed system).
- However, due to several problems, deadlock avoidance is impractical in distributed systems.
- Deadlock detection requires examination of the status of process-resource interactions for presence of cyclic wait.
- Deadlock detection in distributed systems seems to be the best approach to handle deadlocks in distributed systems.

Issues in deadlock detection

Deadlock handling using the approach of deadlock detection entails addressing two basic issues: first, detection of existing deadlocks and, second, resolution of detected deadlocks.

- **Detection of deadlocks**
  Detection of deadlocks involves addressing two issues: maintenance of the WFG and searching of the WFG for the presence of cycles (or knots). Since, in distributed systems, a cycle or knot may involve several sites, the search for cycles greatly depends upon how the WFG of the system is represented across the system. Depending upon the way WFG information ismaintained and the search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems.

  Correctness criteria
  A deadlock detection algorithm must satisfy the following two conditions:

  • **Progress (no undetected deadlocks)** The algorithm must detect all existing deadlocks in a finite time.

  Once a deadlock has occurred, the deadlock detection activity should continuously progress until the deadlock is detected. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

  • **Safety (no false deadlocks)** The algorithm should not report deadlocks that do not exist (called *phantom or false* deadlocks). In distributed systems where there is no global memory and there is no global clock, it is difficult to design a correct deadlock detection algorithm because sites may obtain an out-of-date and inconsistent WFG of the system. As a result, sites may detect a cycle that never existed but whose different segments existed in the system at different times. This is the main reason why many deadlock detection algorithms reported in the literature are incorrect.
- **Resolution of a detected deadlock**
  Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock. It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution. Note that several deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph. Therefore, when a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system. If this information is not cleaned in a timely manner, it may result in detection of phantom deadlocks. Untimely and inappropriate cleaning of broken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

Models of deadlocks

@ The single-resource model

The single-resource model is the simplest resource model in a distributed system, where a process can have at most one outstanding request for only one unit of a resource. Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock. In a later section, an algorithm to detect deadlock in the single-resource model is presented.

@ The AND model

In the AND model, a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process. The requested resources may exist at different locations. The out degree of a node in the WFG for AND model can be more than 1. The presence of a cycle in the WFG indicates a deadlock in the AND model. Each node of the WFG in such a model is called an AND node.
Consider the example WFG described in the Figure. Process P11 has two outstanding resource requests. In case of the AND model, P11 shall become active from idle state only after both the resources are granted. There is a cycle P11→P21→P24→P54→P11, which corresponds to a deadlock situation.

In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process *P*44 in Figure. It is not a part of any cycle but is still deadlocked as it is dependent on *P*24, which is deadlocked. Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

@ The OR Model

In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted. The requested resources may exist at different locations. If all requests in the WFG are OR requests, then the nodes are called OR nodes. Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model. To make it more clear, consider Figure If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied. After P32 finishes execution and releases its resources, process P11 can continue with its processing.

In the OR model, the presence of a knot indicates a deadlock. In a WFG, a vertex v is in a knot if for all u :: u is reachable from v : v is reachable from u. No paths originating from a knot shall have dead ends.

A deadlock in the OR model can be intuitively defined as follows [6]: a process Pi is blocked if it has a pending OR request to be satisfied. With every blocked process, there is an associated set of processes called dependent set. A process shall move from an *idle* to an *active* state on receiving a grant message from any of the processes in its dependent set. A process is permanently blocked if it never receives a grant message from any of the processes in its dependent set. Intuitively, a set of processes S is deadlocked if all the processes in S are permanently blocked. To formally state that a set of processes is deadlocked, the following conditions hold true:

1. Each of the process is the set S is blocked.
2. The dependent set for each process in S is a subset of S.
3. No grant message is in transit between any two processes in set S.

Hence, deadlock detection in the OR model is equivalent to finding knots in the graph.

@ The AND-OR Model

A generalization of the previous two models (OR model and AND model) is the AND-OR model. In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request. For example, in the ANDOR model, a request for multiple resources can be of the form x *and* (y *or* z). The requested resources may exist at different locations. To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG. Since a deadlock is a stable property (i.e., once it exists, it does not go away by itself), this property can be exploited and a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock. However, this is a very inefficient strategy.

@ The $\binom{p}{q}$ Model

Another form of the AND-OR model is the $\binom{p}{q}$ model (called the P-out-of-Q model), which allows a request to obtain any *k* available resources from a pool _of n resources. Both the models are the same in expressive power. However, $\binom{p}{q}$ model lends itself to a much more compact formation of a request. Every request in the $\binom{p}{q}$ model can be expressed in the AND-OR model and vice-versa. Note that AND requests for p resources can be stated as $\binom{p}{p}$ and OR requests for p resources can be stated as $\binom{p}{1}$

@ The Unrestricted Model

In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests. In this model, only one assumption that the deadlock is stable is made and hence it is the most general model. This way of looking at the deadlock problem helps in separation of concerns: concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication).