**Advance Operating System**

**1a) Define Global State.**

* A global state consists of a set of local states and a set of states of the communication channels.
* The state of the communication channel in a consistent global state should be the sequence of messages sent along the channel before the sender’s state was recorded, excluding the sequence of messages received along the channel before the receiver’s state was recorded.
* It is difficult to record channel states to ensure the above rule ⇒ global states are very often recorded without using channel states.

**1b) Write a difference between logical clock and vector clock.**

**Logical clock**

It is a mechanism for capturing causal and chronological relationships in a distributed system. A physically synchronous global clock may not be present in a distributed system. In such systems a logical clock allows global ordering on events from different processes.

Lamport’s logical clocks impose only a partial order on the set of events and not powerful enough to perform a causal ordering of events.

**Logical Clock Algorithm**

*Ci* is the local clock for process *Pi*

* if *a* and *b* are two successive events in *Pi*, then  
  *Ci*(*b*) = *Ci*(*a*) + *d*1, where *d*1 > 0
* if *a* is the sending of message *m* by *Pi*, then *m* is assigned timestamp *tm* = *Ci*(*a*)
* if *b* is the receipt of *m* by *Pj*, then  
  *Cj*(*b*) = max{*Cj*(*b*), *tm* + *d*2}, where *d*2 > 0

**Vector clock**

It is an algorithm for generating a partial ordering of events in a distributed system. It detects causality violations. Like the Lamport timestamps interprocess messages contain the state of the sending process's logical clock.

**Vector Clock Algorithm**

* if *a* and *b* are successive events in *Pi*, then *Ci*[*i*](*b*) = *Ci*[*i*](*a*) + *d*1
* if *a* is sending of *m* by *Pi* with vector timestamp *tm*  
  *b* is receipt of *m* by *Pj* then  
  *Cj*[*k*](*b*) = max{*Cj*[*k*](*b*), *tm*[*k*]}

**1c) Write a difference between Operating System and Distributed Operating System**

|  |  |
| --- | --- |
| **Operating System** | **Distributed Operating System** |
| A operating system is an ordinary operating system but runs on individual system | A distributed operating system is an ordinary centralized operating system but runs on multiple independent CPUs. |
| Environment users are aware of multiplicity of machines. | Environment users are not aware of multiplicity of machines. |
| Control over file placement is done manually by the user. | It can be done automatically by the system itself. |
| Performance is badly affected if certain part of the hardware starts malfunctioning. | It is more reliable or fault tolerant i.e distributed operating system performs even if certain part of the hardware starts malfunctioning. |
| Remote resources are accessed by either logging into the desired remote machine or transferring data from the remote machine to user's own machines. | Users access remote resources in the same manner as they access local resources. |

**1d) Define Deadlock?**

***Deadlock***is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)  
***Mutual Exclusion****:* One or more than one resource are non-sharable (Only one process can use at a time)  
***Hold and Wait:***A process is holding at least one resource and waiting for resources.  
***No Preemption:***  A resource cannot be taken from a process unless the process releases the resource.  
***Circular Wait:***A set of processes are waiting for each other in circular form.

**1e) How to prevent Deadlock?**

There are three ways to prevent deadlock

1. Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.  
   One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.

Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

1. Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.
2. Ignore the problem all together: If deadlock is very rare, then let it happen and reboot the system. This is the approach that both Windows and UNIX take.

**1f) Mention any two performance measures to measurer algorithm performance.**

Recall that the program performance is the amount of computer memory and time needed to run a program.

Analytically - performance analysis

Experimentally - performance measurement

The performance of a program depends on

the number and type of operations performed, and

the memory access pattern for the data and instructions

**1g) Define Agreement Protocol?**

A processor refines its value as it learns of the values of other processors (This entire process of reaching an **agreement** is called an **agreement protocol**).

• To achieve some common goal in Distributed system

• To deal with Various faulty processess which might produce wrong results and send wrong information to other processes

• To achieve reliability of Distributed system

Agreement problems have been studied under the following system model:

* + - 1. There are n processors in the system and at most m processors can be faulty.
      2. The processors can directly communicate with other processors by message passing. Thus, the system is logically fully connected.
      3. A receiver processor always knows the identity of the sender processor of the message.
      4. The communication medium is reliable, and only processors are prone to failures.

**1h) Write about operation on distributed file.**

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files.

The file system is responsible for controlling access to the data and for performing low- level operations such as buffering frequently used data and issuing disk I/O requests.

Below operation happened on distributed file

* **Access transparency:** Clients are unaware that files are distributed and can access them in the same way as local files are accessed.
* **Location transparency:** A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.
* **Concurrency transparency:** All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.
* **Failure transparency:** The client and client programs should operate correctly after a server failure.
* **Heterogeneity:** File service should be provided across different hardware and operating system platforms.
* **Scalability:** The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).
* **Replication transparency:** To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.
* **Migration transparency:** Files should be able to move around without the client's knowledge.
* Support fine-grained distribution of data: To optimize performance, we may wish to locate individual objects near the processes that use them.
* **Tolerance for network partitioning:** The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

**1i) Define Mutual Exclusion.**

The **mutual exclusion (ME)** problem frequently arises when concurrent access to shared resources by several sites is involved (e.g., directory management where updates and reads to a directory must be done atomically to ensure correctness.) Mutual exclusion is a fundamental issue in the design of distributed systems, and an efficient and robust technique for mutual exclusion is essential.

Because of resource distribution, transmission delays, and lack of global information, mutual exclusion algorithms developed for single-computer systems are not applicable in a distributed environment.

**1j) Define Fault Tolerance.**

Fault tolerance is the way in which an operating system (OS) responds to a hardware or software failure. The term essentially refers to a system’s ability to allow for failures or malfunctions, and this ability may be provided by software, hardware or a combination of both. To handle faults gracefully, some computer systems have two or more duplicate systems.

Fault tolerance software may be part of the OS interface, allowing the programmer to check critical data at specific points during a transaction.

Fault tolerance can include:

* Responding to a power failure (the lowest level of fault tolerance)
* Immediately using a backup system in the event of a system failure
* Allowing mirrored disks to immediately take over for a failed disk
* Multiple processors working together and comparing data and output for errors, then immediately correcting the detected errors.

**2a) Explain Chandy-Lamport’s Global state Recording algorithm.**

Each distributed system has a number of processes running on a number of different physical servers. These processes communicate with each other via communication channels using text messaging.

A process could record it own local state at a given time but the messages that are in transit (on its way to be delivered) would not be included in the recorded state and hence the actual state of the system would be incorrect after the time in transit message is delivered.

Any process in the distributed system can initiate this global state recording algorithm using a special message called **MARKER**. This marker traverse the distributed system across all communication channel and cause each process to record its own state. In the end, the state of entire system (Global state) is recorded. This algorithm does not interfere with normal execution of processes.

**Assumptions of the algorithm:**

* There are finite number of processes in the distributed system and they do not share memory and clocks.
* There are finite number of communication channels and they are unidirectional and FIFO ordered.
* There exists a communication path between any two processes in the system
* On a channel, messages are received in the same order as they are sent.

**Algorithm:**

* **Marker sending rule for a process *P* :**
  + Process **p** records its own local state
  + For each outgoing channel **C** from process **P**, **P** sends marker along **C** before sending any other messages along **C**.  
    (**Note:** Process Q will receive this marker on his incoming channel C1.)
* **Marker receiving rule for a process *Q* :**
  + If process **Q** has not yet recorded its own local state then
    - Record the state of incoming channel C1 as an empty sequence or null.
    - After recording the state of incoming channel C1, process **Q** Follows the marker sending rule
  + If process **Q** has already recorded its state
    - Record the state of incoming channel **C1** as the sequence of messages received along channel C1 after the state of **Q** was recorded and before **Q** received the marker along C1 from process P.

**2b) Describe issues in Distributed operating system.**

**Some important issues that arise in the design of a distributed operating system are:**

* + Unavailability of up to date Global knowledge
  + Naming
  + Scalability
  + Compatibility

Binary level,

Execution level

Protocol level

* + Process Synchronization
  + Resource management

Data migration

Computational migration

Distributed scheduling

* + Security
  + Structuring

Monolithic kernel

Collective kernel

structure Object-oriented operating system

* + Client Server Computing Model

**Lack of Global Knowledge**

* *Communication delays* are at the core of the problem
* Information may become false before it can be acted upon
* these create some fundamental problems:
  + no global clock -- scheduling based on fifo queue?
  + no global state -- what is the state of a task? What is a correct program?

**Naming**

* named objects: computers, users, files, printers, services
* namespace must be large
* unique (or at least unambiguous) names are needed
* logical to physical mapping needed
* mapping must be changeable, expandable, reliable, fast

**Scalability**

* How large is the system designed for?
* How does increasing number of hosts affect overhead?
* broadcasting primitives, directories stored at every computer -- these design options will not work for large systems.

**Compatibility**

* Binary level: same architecture (object code)
* Execution level: same source code can be compiled and executed (source code).
* Protocol level: only requires all system components to support a common set of protocols.

**Process synchronization**

* test-and-set instruction won't work.
* Need all new synchronization mechanisms for distributed systems.

**Distributed Resource Management**

* Data migration: data are brought to the location that needs them.
  + distributed filesystem (file migration)
  + distributed shared memory (page migration)
* Computation migration: the computation migrates to another location.
  + remote procedure call: computation is done at the remote machine.
  + processes migration: processes are transferred to other processors.

**Security**

* Authetication: guaranteeing that an entity is what it claims to be.
* Authorization: deciding what privileges an entity has and making only those privileges available.

**Structuring**

* the monolithic kernel: one piece
* the collective kernel structure: a collection of processes
* object oriented: the services provided by the OS are implemented as a set of objects.
* client-server: servers provide the services and clients use the services.

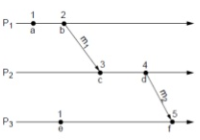
**Communication Networks**

* WAN and LAN
* traditional operating systems implement the TCP/IP protocol stack: host to network layer, IP layer, transport layer, application layer.
* Most distributed operating systems are not concerned with the lower layer communication primitives.

**2c) Write in detail about Lamport’s Logical Clocks.**

**Lamport’s Logical Clocks**

1. The order of events occurring at different processes is critical for many distributed applications. Example: P.o\_created and P.c\_created.
2. Ordering can be based on two simple situations:
3. If two events occurred in the same process then they occurred in the order observed following the respective process;
4. Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving it.
5. Ordering by Lamport is based on the happened before relation (denoted by →):
6. a → b, if a and b are events in the same process and a occurred before b;
7. a→b, if a is the event of sending a message m in a process, and b is the event of the same message m being received by another process;
8. I If a → b and b → c, then a → c (the relation is transitive).
9. If a→b, we say that event a causally affects event b. The two events are causally related.
10. There are events which are not related by the happened-before relation. If both a → e and e → a are false, then a and e are concurrent events; we can write a||e.
11. Using physical clocks, the happened before relation cannot be captured. It is possible that b → c and at the same time Tb > Tc (Tb is the physical time of b).
12. Logical clocks can be used in order to capture the happened-before relation.

* A logical clock is a monotonically increasing software counter.
* There is a logical clock CPi at each process Pi in the system.
* The value of the logical clock is used to assign timestamps to events. CPi(a) is the timestamp of event a in process Pi.
* There is no relationship between a logical clock and any physical clock.

**Rules of Lamport’s Logical Clocks**

*Ci* is the local clock for process *Pi*

* if *a* and *b* are two successive events in *Pi*, then  
  *Ci*(*b*) = *Ci*(*a*) + *d*1, where *d*1 > 0
* if *a* is the sending of message *m* by *Pi*, then *m* is assigned timestamp *tm* = *Ci*(*a*)
* if *b* is the receipt of *m* by *Pj*, then  
  *Cj*(*b*) = max{*Cj*(*b*), *tm* + *d*2}, where *d*2 > 0

**Logical Clock Conditions**

The value of *d* could be 1, or it could be an approximation to the elapsed real time. For example, we could take *d*1 to be the elapsed local time, and *d*2 to be the estimated message transmission time. The latter solves the problem of waiting forever for a virtual time instant to pass.

**Disadvantages of Lamport’s Clock**

Lamport’s logical clocks are not powerful enough to perform a causal ordering of events.

* If a → b, then C(a) < C(b).

However, the reverse is not always true (if the events occurred in different processes):

* If C(a) < C(b), then a → b is not necessarily true. ( It is only guaranteed that b → a is not true).

**2d) Explain Raymond Tree based algorithm with example.**

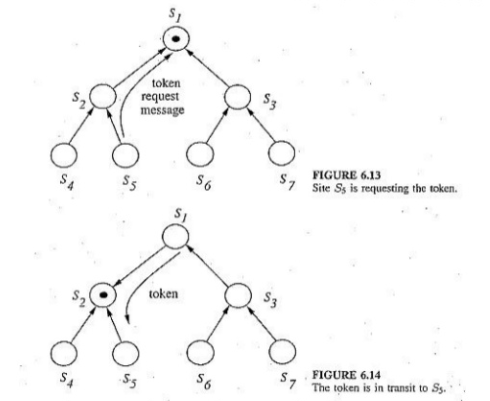
**Raymond’s Tree-Based Algorithm**

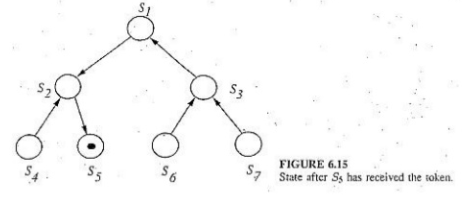
In Raymond’s tree-based distributed mutual exclusion algorithm, sites are logically arranged as a directed tree rooted at the site holding the token. Every site maintains a local variable holder that points to its parent site in the tree structure. At the root site, holder points to itself. Every site also keeps a FIFO queue, called request\_q, which stores the requests of those sites that have sent a request to this site, but have not yet been sent the token.

* 1. Requesting the critical section.
* A site Si that wants to enter its CS sends a REQUEST message to its parent site (indicated in the holder variable), provided it does not hold the token and its request\_q is empty. It then adds its request to its request\_q.
* When a site on the path receives the REQUEST message, it places the REQUEST in the request\_q and sends a REQUEST message to its parent along the directed path to the root, provided it has not sent out a REQUEST message for a previously received REQUEST on its request\_q.
* When the root site receives a REQUEST message, it sends the token to the site from which it received the REQUEST message, and sets its holder variable to point to that site.
* When a site receives the token, it deletes the top entry from its request\_q, sends the token to the site indicated in this entry, and sets its holder variable to point to that site. If the request\_q is nonempty at this point, then the site sends a REQUEST message to its parent site as indicated in the holder variable.
  1. Executing the critical section.

When a site receives the token and the deleted top entry from its request\_q is its own request entry, it sets the holder variable to point to itself and executes the CS.

* 1. Releasing the critical section.
* When a site exits the CS, it checks its request\_q. If its request\_q is nonempty, it deletes the top entry from its request\_q, sends the token to that site, and sets its holder variable to point to that site.
* If the request\_q is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by the holder variable.





The correctness of the algorithm in mutual exclusion enforcement is obvious since only one token is used. The algorithm is free from deadlock because the acyclic nature of the structure eliminates the possibility of circular wait among requesting sites. The algorithm is free from starvation can be rationalized based on two facts: (1) a site serves requests from its request\_q in the FCFS order, and (2) every site has a path leading to the site that has the token.

**1e) Compare and contrast Token based algorithm.**

**TOKEN-BASED ALGORITHMS**

* A unique **token** is shared among all the sites.
* If a site possesses the unique token, it is allowed to enter its critical section
* This approach uses sequence number to order requests for the critical section.
* Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
* This approach insures Mutual exclusion as the token is unique

The various algorithms differ in the way a site carries out the search for the token.

**Suzuki-Kasami’s Broadcast Algorithm**

**Singhal’s Heuristic Algorithm**

**Raymond’s Tree-Based Algorithm**

**Singhal’s Heuristic Algorithm**

Instead of broadcasting request messages for acquiring the token, Singhal’s algorithm has each site keep information about the state of other sites and use the information to select the sites that are likely to have the token. The objective of this effort is to reduce the number of messages required for the CS execution. It is called a heuristic algorithm because sites are heuristically selected for sending the request messages. In order to avoid the request being unanswered, a requesting site must send the request message to a site that either holds the token or is going to obtain the token in the near future. Hence one design requirement of this algorithm is that a site must select a subset of sites such that at least one of those sites is guaranteed to get the token in the near future.

A site Si maintains two arrays SVi[1.. N] and SNi[1...N] , to store information about other sites in the system. These arrays store the state and the highest known sequence number for each site, respectively. The token also contain two similar arrays, denoted by TSV[1... N] and TSN[1... N] . As in Suzuki-Kasami’s algorithm, sequence numbers are used to detect outdated requests. A site can be in one of the following states:

R – requesting the CS

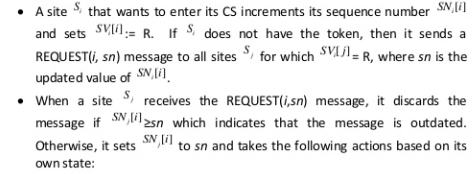
E – executing the CS

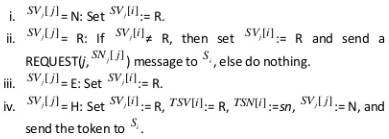
H – holding the idle token

N – none of the above

The Algorithm

* + 1. Requesting the critical section.





* + 1. Executing the critical section.

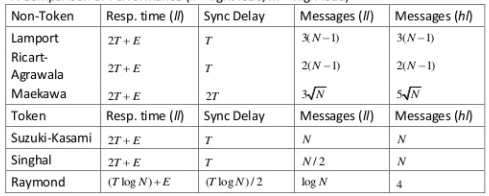
When Site Si receives the token, it first sets SVi[i] := E, then executes the CS.

* + 1. Releasing the critical section.



**Performance:** Singhal’s algorithm allows a site to enter CS without communicating with every site in the system. In low to moderate loads, the message traffic is N/2 because a site needs to send REQUEST to approximately half the sites on average. At high loads, however, when every site is requesting for CS, then most entries in every site’s SV array have the value R, hence the message traffic increases to N. Synchronization delay in the algorithm isT .

A Comparison of Performance (ll = light load, hl = high load)



**2f) Describe Ricart - Agrawala Algorithm with example.**

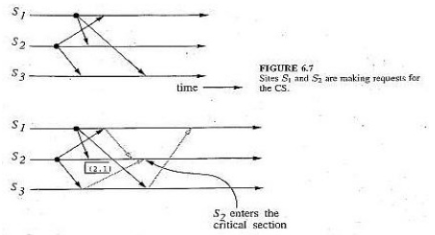
**The Ricart - Agrawala Algorithm**

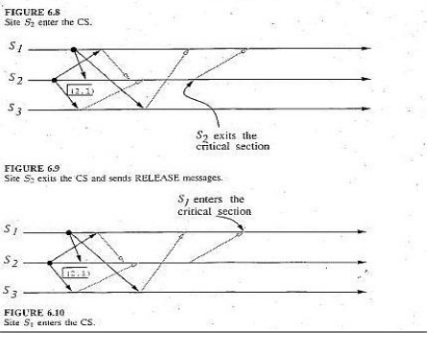
The Ricart-Agrawala algorithm is an optimization of Lamport’s algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. Again in this algorithm, every site Si, 1 < I < N, maintains request set Ri = {S1, S2, S3, ......, SN},

The Algorithm

* 1. Requesting the critical section
* When a site Si wants to enter the CS, it sends a REQUEST message to all the sites in its request set Ri.
* When a site Sj receives the REQUEST message from site Si, it sends a REPLY message to Si if site Sj is neither requesting nor executing the CS or if Sj is requesting and Si’s request’s timestamp is smaller than site Sj’s own request’s timestamp. The request is deferred otherwise.
  1. Executing the critical section Site Si enters the CS after it has received REPLY messages from all sites in its request set Ri.
  2. Releasing the critical section When site Si exits the CS, it sends REPLY messages to all sites it has deferred REPLY to.

Illustration of Ricart-Agrawala Algorithm





Space-time diagrams of example to illustrate how the Ricart-Agrawala algorithm works. The correctness of the algorithm can also be proved by contradiction.

**Performance:** The Ricart-Agrawala algorithm requires 2(N-1) messages per CS invocation: (N-1) REQUEST and (N-1) REPLY messages. Synchronization delay in the algorithm is also T, the average message delay, as in the Lamport’s algorithm.

**2g) Compare and contrast centralized and distributed deadlock system.**

**Centralized deadlock system**

In a network of n sites, one site is chosen as a control site. This site is responsible for deadlock detection. It has control over all resources of the system. If a site requires a resource it requests the control site, the control site allocates and de-allocates resources and maintains a wait for graph. And at regular interval of time, it checks the wait for graph to detect a cycle. If cycle exits then it will declare system as deadlock otherwise the system will continue working. The major drawbacks of this technique are as follows:

1. A site has to send request even for using its own resource.
2. There is a possibility of phantom deadlock.

**Centralized Deadlock‐Detection Algorithms**

The Ho‐Ramamoorthy Algorithms

1. The Two‐Phase Algorithm

2. The One‐phase Algorithm

**Distributed deadlock system**

In the distributed approach different nodes work together to detect deadlocks. No single point failure ( that is the whole system is dependent on one node if that node fails the whole system crashes) as workload is equally divided among all nodes. The speed of deadlock detection also increases.

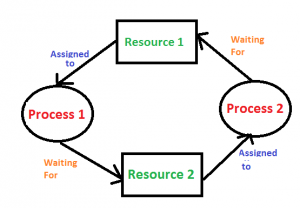
**Chandy-Misra-Haas’s distributed deadlock detection algorithm** is an edge chasing algorithm to detect deadlock in distributed systems.

**2h) Explain Different strategy to handle deadlock**

Deadlock handle by following two step:

* + 1. **Deadlock Detection**
    2. **Deadlock Recovery**

**Deadlock Detection**

1. If resources have single instance:  
   In this case for Deadlock detection we can run an algorithm to check for cycle in the Resource Allocation Graph. Presence of cycle in the graph is the sufficient condition for deadlock.  
   [](https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/06/deadlock.png)

In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is confirmed.

1. If there are multiple instances of resources:  
   Detection of the cycle is necessary but not sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

**Deadlock Recovery**  
A traditional operating system such as Windows doesn’t deal with deadlock recovery as it is time and space consuming process. Real-time operating systems use Deadlock recovery.

**Recovery method**

**1. Process Termination:**  
To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

* **(a). Abort all the Deadlocked Processes:**  
  Aborting all the processes will certainly break the deadlock, but with a great expenses. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.
* **(b). Abort one process at a time untill deadlock is eliminated:**  
  Abort one deadlocked process at a time, untill deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

**2. Resource Preemption:**  
To eliminate deadlocks using resource preemption, we preepmt some resources from processes and give those resources to other processes. This method will raise three issues –

* **(a). Selecting a victim:**  
  We must determine which resources and which processes are to be preempted and also the order to minimize the cost.
* **(b). Rollback:**  
  We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.
* **(c). Starvation:**  
  In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

**2i) illustrate any one Hierarchical deadlock detection algorithm.**

**Hierarchical Deadlock‐Detection Algorithms**

In hierarchical deadlock detection algorithm, sites are arranged in a hierarchical fashion and a site detects deadlocks involving only its descendant sites. Distributed deadlock algorithms delegate the responsibility of deadlock detection to individual sites while in hierarchical there are local detectors at each site which communicate their local wait for graphs(WFG) with one another.

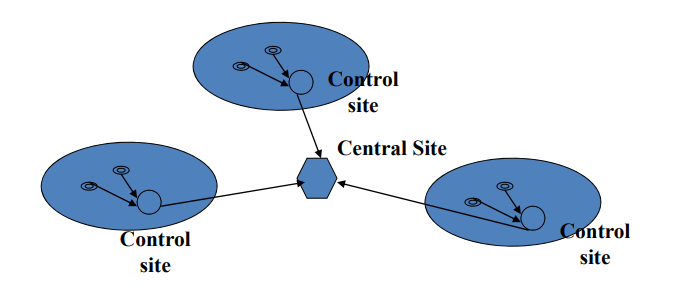
**Approach:**  
Deadlocks that are local to a single site are detected at that site using their local WFG. Each site also sends its local WFG to deadlock detector at the next level. Thus, distributed deadlocks involving 2 or more sites would be detected by a deadlock detector in lowest level that has control over these sites.

**1. Ho-Ramamoorthy algorithm:**

* Uses only two levels i.e. Master control nodes and Cluster control nodes.
* Cluster control nodes are used for detecting deadlock among their members and reporting dependencies outside their cluster to Master control node.
* The master control node is responsible for detecting inter-cluster deadlocks.
* Follows Ho-Ramamoorthy’s 1-phase algorithm. More than 1 control site organized in hierarchical manner.
* Each control site applies 1-phase algorithm to detect (intracluster) deadlocks.
* Central site collects info from control sites, applies 1-phase algorithm to detect intracluster deadlocks.

**Ho‐Ramamoorthy 1‐phase Algorithm**

* + 1. Each site maintains 2 status tables: resource status table and process status table.
    2. Resource table: transactions that have locked or are waiting for resources.
    3. Process table: resources locked by or waited on by transactions.
    4. Controller periodically collects these tables from each site.
    5. Constructs a WFG from transactions common to both the tables.
    6. No cycle, no deadlocks.
    7. A cycle means a deadlock.



**2j) Explain the concept of Distributed share Memory**

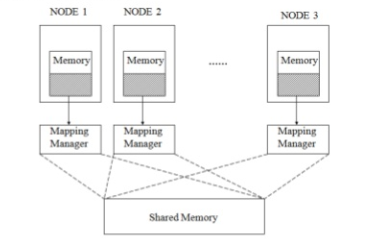
**What Is DSM?**

* + - 1. The distributed shared memory (DSM) implements the shared memory model in distributed systems, which have no physical shared memory.
      2. The shared memory model provides a virtual address space shared between all nodes.
      3. The overcome the high cost of communication in distributed systems, DSM systems move data to the location of access.

**How DSM Works?**

1. Data moves between main memory and secondary memory (within a node) and between main memories of different nodes.
2. Each data object is owned by a node
   1. Initial owner is the node that created object
   2. Ownership can change as object moves from node to node
3. When a process accesses data in the shared address space, the mapping manager maps shared memory address to physical memory (local or remote)

**Architecture of Distributed Memory**



**Advantages of distributed shared memory (DSM)**

1. Data sharing is implicit, hiding data movement (as opposed to ‘Send’/‘Receive’ in message passing model)
2. Passing data structures containing pointers is easier (in message passing model data moves between different address spaces)
3. Moving entire object to user takes advantage of locality difference
4. Less expensive to build than tightly coupled multiprocessor system: off-the-shelf hardware, no expensive interface to shared physical memory
5. Very large total physical memory for all nodes: Large programs can run more efficiently
6. No serial access to common bus for shared physical memory like in multiprocessor systems
7. Programs written for shared memory multiprocessors can be run on DSM systems with minimum changes

**2k) Describe any one Mutual Exclusion Algorithm with example.**

**Lamport’s Algorithm**

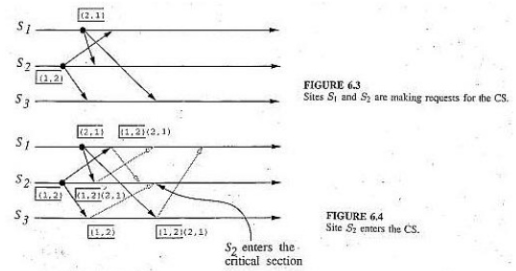
Lamport gave the following distributed mutual exclusion algorithm as an illustration of his scheme of logical clocks.

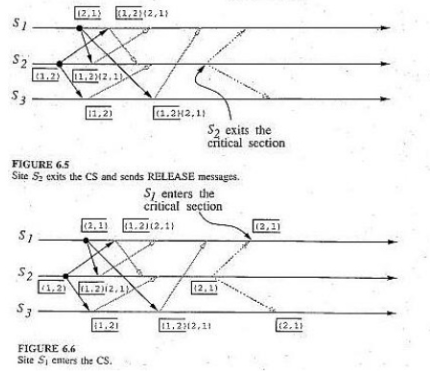
In this algorithm, every site Si, 1<i< N, maintains a request set Ri = {S1, S2,......, SN}, and a queue called request queue ‘RQi’ that contains mutual exclusion requests it has received and not yet released in order of their timestamps. The algorithm requires messages to be delivered in the FIFO order between every pair of sites.

* 1. Requesting the critical section
* When a site Si wants to enter the CS, it sends a REQUEST (tSi, i) message to all the sites in its request set Ri and places the request on Qi. Its placement in the queue is based on its timestamp (tSi,i).
* When a site Sj receives the REQUEST (tSi, i) message from site Si, it returns a timestamped REPLY message to Si, and place site Si’s request on request queue RQi.
  1. Executing the critical section

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

* [L1]: Si receives a message with timestamp larger than (tSi, i) from all other sites.
* [L2]: Si’s request is at the top of RQi.
  1. Releasing the critical section
* Site Si, upon exiting the CS, removes its request from the top of RQi and sends a time stamped RELEASE message to all sites in Ri.
* When a site Sj receives a RELEASE message from site Si, it removes Si’s request from RQj.
* 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.





The Lamport’s algorithm executes CS requests in the increasing order of timestamps.

Space-time diagrams of example to illustrate how the Lamport’s algorithm works. The algorithm’s correctness can be proved by contradiction. Illustration of Lamport’s Algorithm

**Performance**

Lamport’s algorithm requires 3(N-1) messages per CS invocation: (N-1) REQUEST, (N-1) REPLY, and (N-1) RELEASE messages. Synchronization delay in the algorithm is T, the average message delay.

**An Optimization**

Lamport’s algorithm can be optimized to require between 3(N-1) and 2(N-1) messages per CS invocation by suppressing REPLY messages in the following situation: Site Sj receives a REQUEST message from site Si after it has sent its own REQUEST message with timestamp higher than the timestamp of site Si’s request. This is possible since Sj’s REQUEST message would have satisfied condition [L1] in Si’s decision for entering the CS.

**2l) What do you mean by Distributed scheduling? How it’s different from normal scheduling.**

Scheduling refers to the execution of non-interactive processes or tasks (usually called `jobs') at designated times and places around a network of computers (see the Special Topics Guide on Scheduling). Distributed Scheduling refers to the chaining of different jobs into a coordinated workflow that spans several computers. For example, you schedule a processing job on machine1 and machine2, and when these are finished you need to schedule a job on machine3. This is distributed scheduling.

* Resource and CPU queue lengths are good indicators of load.
* Artificially increment CPU queue length for transferred jobs on their way.
* Set timeouts for such jobs to safeguard against transfer failures.
* Little correlation between queue length and CPU utilization for interactive jobs: use utilization instead.
* Monitoring CPU utilization is expensive.

**Static**

: decisions *hard-wired* into algorithm using prior knowledge of system.

**Dynamic**

: use state information to make  
decisions.

**Adaptive**

: special case of dynamic algorithms; dynamically change parameters of the  
algorithm.

Need to consider overheads of system state collection.

**3) Describe various system type architecture in details.**

**System Architecture Types**

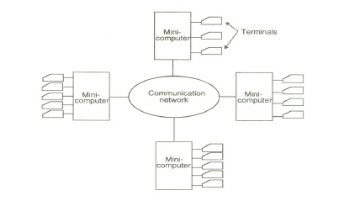
Distributed systems can be modeled into several types. Various models are used for building distributed computing systems. These models can be broadly classified into five categories, and they are described below:

1. Mini Computer Model
2. Workstation Model
3. Workstation Server Model
4. Processor Pool Model
5. Hybrid Model

**Mini Computer Model**

In this model, the distributed system consists of several minicomputers. Each computer supports multiple users and provides access to remote resources. The ratio of processors to users is normally less than one.

Minicomputer model is a simple extension of the centralized time-sharing system. As shown in below Figure ,a distributed computing system based on this model consists of a few minicomputers. They may be large supercomputers as well interconnected by a communication network. Each minicomputer usually has multiple users simultaneously logged on to it.

Several interactive terminals are connected to each minicomputer.

Each user is logged on to one specific minicomputer, with remote

access to other minicomputers. The network allows a user to

access remote resources that are available on some machine

other that the one on to which the user is currently logged.

The minicomputer model may be used when resource sharing

(such as sharing of information databases of different types,

with each type of database located on a different machine) with

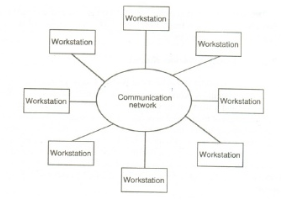
remote users is desired. The early ARpAnet is an example of

a distributed computing system based on the minicomputer model.

**Workstation Model**

In this model, the distributed system consists of several workstations; every user has a workstation where user’s work is performed. With the help of distributed file system, a user can access data regardless of the location of the data. The ratio of processors to users is normally one. The workstations are independent computers with memory, hard disks, keyboard and console. Workstations are connected with each other through communication network.

A company office or a university department may have several workstations scattered throughout a building or compass each workstation equipped with its own disk and serving as a single-user computer.

In this model a user logs onto one of the workstations called his or her home workstation and submits jobs for execution. When the system finds that the users workstation does not have sufficient processing power for executing the processes of the submitted jobs efficiently, it transfers one or more of the processes from the users workstation to some other workstation that is currently idle and gets the process executed there, and finally the result of execution is returned to the user’s, in. workstation.

This model is not so simple to implement because several issues

must be resolved. These issues are as follows:

* + How does the system find an idle workstation?
  + How is a process transferred from one workstation

to get it executed on another workstation?

* + What happens to a remote process if a user logs onto a

Work station that was idle until now and was being used

to execute a process of another workstation?

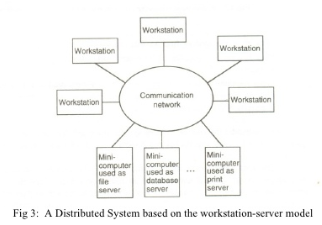
Three commonly used approaches for handling the third issue are as follows:

1. The first approach is to allow the remote process share the resources of the workstation along win its own logged -on .users processes. This method is easy to implement, but it defeats the main idea of workstations serving as personal computers, because if remote processes are allowed to execute simultaneously with the logged-on users own processes, the logged-on user does not get his or her guaranteed response.
2. The second approach is to kill the remote process. The main drawbacks of this method are that all processing done by the remote process gets lost and tie file system may be left in an inconsistent state; making this method unattractive.

The third approach is, migrate the remote process back to its home workstation, so that its execution can be continued there. This method is difficult to implement because it requires the system to support pre-emptive process migration facility.

**Workstation-Server Model**

A workstation with its own local disk is usually called a disk full workstation and a workstation without a local disk is called a diskless workstation. With high-speed networks, diskless workstations have become more popular than disk full workstations, making the workstation-server model more popular than the workstation model for building distributed computing systems.



As shown in Figure 3, a distributed computing system based on the

workstation-server model consists of a few minicomputers and several

workstations interconnected by a communication network. Most of the

workstation may be diskless, but a few of may be disk full.

When diskless workstations are used on a network, the file system to

be used by these workstations must be implemented either by a

diskfull workstation or by a minicomputer equipped with a disk for

file storage.

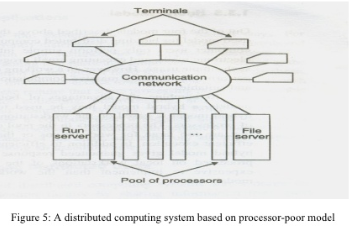
As compared to the workstation model, the workstation-server model has several advantages:

1. It is much cheaper to use a few minicomputers equipped with large fast disks that are accessed over the network than a large number of dishful workstations, with each workstation having a small, slow disk.
2. Diskless workstations are also preferred to dishful workstations from a system maintenance point of view. Software installation, backup and hardware maintenance are easier to perform with a few large disks than win many small disks scattered all over a building or campus.
3. In the workstation-server model, since all files are managed by be file servers, users have the flexibility to use any workstation and access the files in the same manner irrespective of which workstation the user is currently logged on. Note that this is not true win the workstation model, in which each workstation has its local file system, because different mechanisms are needed to access local and remote files.

In the workstation-server model, the request-response protocol described above is mainly used to access the services of the server machine. Therefore unlike the workstation model, this model does not need a process migration facility which is difficult to implement.

**Processor Pooled Model**

Processor-pool model is based on the observation that most of the time a user does not need any computing power but once m a while he or she may need a very large amount of computing power for a short time. Therefore, in the processor-pooled model the processors are pooled together to be shared by the users as needed. The pool of processors consists of a large number of microcomputer and minicomputers attached to the network. Each processor m the pool has its own memory to load and run a system program or an application program of the distributed computing system



In the pure processors model, the processors m the pool

have no terminals attached directly to them, and users

access the system from terminals that are attached to

the network via special devices. These terminals are

either small diskless workstations or graphic terminals.

A special server, called a run server, manages and

allocates the processors in the pool to different users

on a demand bases. When a user submits a Job for

computation, an appropriate number of professors

are temporarily assigned to the job by the run server.

For Example

* + If the users computation job is the compilation of a program having n segments,
  + Each of the segments can be compiled independently to produce separate releasable object files;
  + n processors from the pool can be allocated to this job to compile all the segments in parallel.
  + When the computation is completed, the processors are returned to the pool for use by other users

In the processor-pool model there is no concept of a home machine. That is, a user does not log onto a particular machine but to the system as a whole This is in contrast to other models in which each user has a home machine (e.g., a workstation or minicomputer) onto which he or she logs and runs most of his orher programs they by default.

**The Hybrid Model**

To combine the advantages of both the workstation-server and processor-pool models, a hybrid model may be used to build a distributed computing system. The hybrid model is based on the workstation-server model but with the addition of a pool of processors. The processors in the pool can be allocated dynamically for computations that are too large for workstations or that requires several computers concurrency for efficient execution. In addition to efficient execution of computation-intensive jobs, the hybrid model gives guaranteed response to interactive jobs by allowing them to be processed on local workstations of the users. However, the hybrid model is more expensive to implement than the workstation-server model or the processor-pool model.

Issues in Distributed Operating Systems

A distributed operating system is a program that manages the resources of a computer system and provides users an easy and friendly interface to operate the system. The typical characteristics of the distributed operating systems are:

* + System appears to its users as a centralized operating system, but it runs on multiple independent computers.
  + Each computer may have the same or different operating system, but not visible to the users.
  + User views the system as a virtual uniprocessor, and not a collection of distinct machines.

User does not know on what computers the job was executed, on what computers the required files are stored and how the system communicates and synchronies among different computers.

**4) Compare and contrast various Non-Token based Algorithm with example.**

**NON-TOKEN-BASED ALGORITHMS**

In non-token-based mutual exclusion algorithms, a site communicates with a set of other sites to arbitrate who should execute the CS next. Each site ‘Si’ maintains a request set ‘Ri’ that contains ids of all those sites from which it must acquire permission before entering the CS. A site can proceed to its CS only after it has acquired permission from all those sites in its request set.

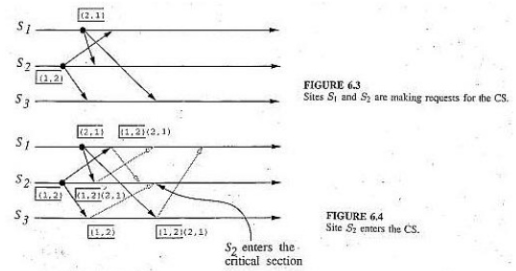
**Lamport’s Algorithm**

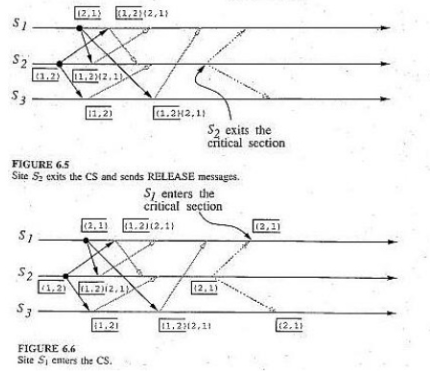
Lamport gave the following distributed mutual exclusion algorithm as an illustration of his scheme of logical clocks.

In this algorithm, every site Si, 1<i< N, maintains a request set Ri = {S1, S2,......, SN}, and a queue called request queue ‘RQi’ that contains mutual exclusion requests it has received and not yet released in order of their timestamps. The algorithm requires messages to be delivered in the FIFO order between every pair of sites.

**The Algorithm**

* **Request**
  + ***Si* sends *REQUEST*(*tsi*, *i*) to all sites in its request set *Ri* and puts the request on *request*\_*queuei***
  + **when *Sj* receives *REQUEST*(*tsi*, *i*) from *Si* it returns a timestamped *REPLY* to *Si* and places *Si*'s request on *request*\_*queuej***
* ***Si* waits to start the CS until both**
  + **[L1:] *Si* has received a message with timestamp > (*tsi*, *i*) from all other sites**
  + **[L2:] *Si*'s request is at the top of *request*\_*queuei***
* **Release**
  + ***Si* removes request from top of *request*\_*queuei* and sends time-stamped *RELEASE* message to all the sites in its request set**
  + **when *Sj* receives a *RELEASE* messages from *Si* it removes *Si*'s request from *request*\_*queuej***





The Lamport’s algorithm executes CS requests in the increasing order of timestamps.

Space-time diagrams of example to illustrate how the Lamport’s algorithm works. The algorithm’s correctness can be proved by contradiction. Illustration of Lamport’s Algorithm

**Performance**

Lamport’s algorithm requires 3(N-1) messages per CS invocation: (N-1) REQUEST, (N-1) REPLY, and (N-1) RELEASE messages. Synchronization delay in the algorithm is T, the average message delay.

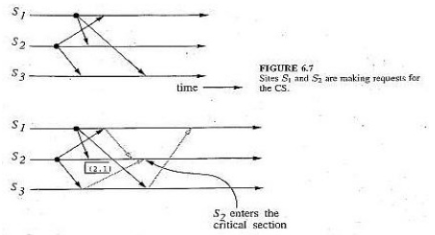
**The Ricart - Agrawala Algorithm**

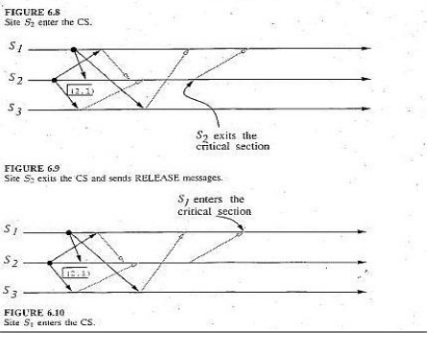
The Ricart-Agrawala algorithm is an optimization of Lamport’s algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. Again in this algorithm, every site Si, 1 < I < N, maintains request set Ri = {S1, S2, S3, ......, SN},

The Algorithm

* 1. Requesting the critical section
* When a site Si wants to enter the CS, it sends a REQUEST message to all the sites in its request set Ri.
* When a site Sj receives the REQUEST message from site Si, it sends a REPLY message to Si if site Sj is neither requesting nor executing the CS or if Sj is requesting and Si’s request’s timestamp is smaller than site Sj’s own request’s timestamp. The request is deferred otherwise.
  1. Executing the critical section Site Si enters the CS after it has received REPLY messages from all sites in its request set Ri.
  2. Releasing the critical section When site Si exits the CS, it sends REPLY messages to all sites it has deferred REPLY to.

Illustration of Ricart-Agrawala Algorithm





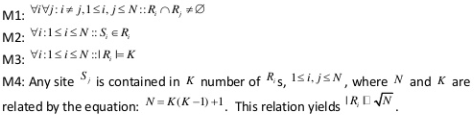
Space-time diagrams of example to illustrate how the Ricart-Agrawala algorithm works. The correctness of the algorithm can also be proved by contradiction.

**Performance:** The Ricart-Agrawala algorithm requires 2(N-1) messages per CS invocation: (N-1) REQUEST and (N-1) REPLY messages. Synchronization delay in the algorithm is also T, the average message delay, as in the Lamport’s algorithm.

**Maekawa’s Algorithm**

In Maekawa’s algorithm, a site requests permission, only from a subset of sites instead of all sites. To prevent two sites to obtain all the necessary permissions for CS, these subsets of sites must have overlaps. Specifically, the request sets are chosen so that  Each site receiving request messages serves as a mediator that ensures only one site under its watch can be given permission for CS at a time.

**Construction of Request Sets:** The request sets for sites in Maekawa’s algorithm are constructed to satisfy the following conditions:



**The Algorithm**

* + 1. Requesting the critical section.

• When a site Si wants to enter the CS, it sends a REQUEST(i) message to all the sites in its request set Ri .

• When a site Sj receives the REQUEST(i) message from site , it sends a REPLY(j) message to  provided it hasn’t sent a REPLY message to a site from the time it received the last RELEASE message. Otherwise, it queues up the REQUEST for later consideration.

* + 1. Executing the critical section.
* Site Sienters the CS after it has received REPLY messages from all sites in its request set Ri .
  + 1. Releasing the critical section.
* When site Si exits the CS, it sends RELEASE(i) messages to all sites in its request set Ri .
* When a site Sj receives the RELEASE(i) message from site Si , it sends a REPLY(j) 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 the site has not sent out any REPLY message.

**Performance:** Maekawa’s algorithm requires  messages per CS invocation:  REQUEST,  REPLY, and

 RELEASE messages. Synchronization delay in the algorithm is 2T , twice as long as in both the Lamport’s and Ricart-Agrawala algorithms. In addition, Maekawa’s algorithm is deadlock-prone, and deadlock handling will require additional messages.

**Handling Deadlocks**

Maekawa’s algorithm handles deadlocks by requiring a site failing to lock all needed sites to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock. A site suspects a deadlock whenever a higher priority request finds that a lower priority request has already locked the site (priority is based on timestamp), thus initiates message exchanges to resolve it. Messages required for deadlock handling are:

**FAILED:** a FAILED message from site Si to site Sj indicates that Si cannot grant Sj ‘s request because it has currently granted permission to a site with a higher priority request.

**INQUIRE:** an INQUIRE message from Si to Sj indicates that Si would like to find out from Sj if it has succeeded in locking all sites in its request set.

**YIELD:** a YIELD message from site Si to Sj indicates that Si is returning the permission to Sj in order to yield to a higher priority request at Si .

**Deadlock handling procedure.**

* + 1. When a REQUEST(ts, i) from site Si blocks at site Sjbecause Sj has currently granted permission to site Sk , then Sj sends a FAILED(j) message to Si if Si ’s request has lower priority. Otherwise, Sj sends an INQUIRE(j) message to site Sk .
    2. In response to an INQUIRE(j) message from site Sj , site Sk sends a YIELD(k) message to Sj , provided Sk has received a FAILED message from a site in its request set or it sent a YIELD message to any of these sites but has not yet received a new GRANT from it.
    3. In response to a YIELD(k) message from site Sk , site Sj assumes it has been released by Sk , places the request of Sk at the appropriate location in the request queue, and sends a GRANT(j) message to the top request’s site in the queue.

These extra messages may be exchanged even though there is no deadlock. The maximum number of messages required per CS invocation with deadlock handling has been determined to be 

**5) How a deadlock can be detected ? Explain distributed Deadlock detection Algorithm with example.**

A deadlock is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.

Four conditions have to be met for deadlock to be present:

* + 1. **Mutual exclusion**: A resource can be held by at most one process.
    2. **Hold and wait**: Processes that already hold resources can wait for another resource.
    3. **Non-preemption**: A resource, once granted, cannot be taken away from a process.
    4. **Circular wait**: Two or more processes are waiting for resources

geld by one of the other processes.

**Deadlock detection**

General methods for preventing or avoiding deadlocks can be difficult to find. Detecting a deadlock condition is generally easier. When a deadlock is detected, it has to be broken. This is traditionally done by killing one or more processes that contribute to the deadlock. Unfortunately, this can lead to annoyed users.

When a deadlock is detected in a system that is based on atomic transactions, it is resolved by aborting one or more transactions. But transactions have been designed to withstand being aborted.

When a transaction is aborted due to deadlock:

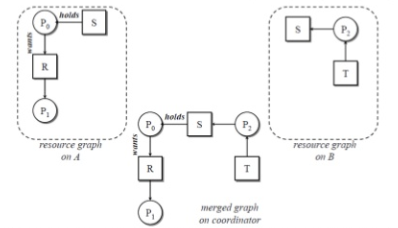
* system is restored to the state it had before the transaction began
* transaction can start again
* hopefully, the resource allocation/utilization will be different now so the transaction can succeed

Consequences of killing a process in a transactional system are less severe.

**Centralized deadlock**

The centralized algorithm attempts to imitate the non-distributed algorithm by using a centralized coordinator. Each machine is responsible for maintaining its own processes and resources. The coordinator maintains the resource utilization graph for the entire system.

Suppose machine A has a process P0 which holds resource S and wants resource R. Resource R is held by P1. This local graph is maintained on machine A. Suppose that another machine B, has a process P2, which is holding resource T and wants resource S. Both of these machines send their graphs to the coordinator, which maintains the union (overall graph). The coordinator sees no cycles. Therefore there are no deadlocks. If a cycle was found (hence a deadlock), the coordinator would have to make a decision on which machine to notify for killing a process to break the deadlock.



**Centralized deadlock detection**

We use a centralized deadlock detection algorithm and try to imitate the non‐distributed algorithm.

* Each machine maintains the resource graph for its own processes and resources.
* A centralized coordinator maintains the resource graph for the entire system.
* When the coordinator detects a cycle, it kills off one process to break the deadlock.
* In updating the coordinator’s graph, messages have to be passed.
* Method 1: Whenever an arc is added or deleted from the resource graph, a message has to be sent to the coordinator.
* Method 2: Periodically, every process can send a list of arcs added and deleted since previous update.
* Method 3: Coordinator asks for information when it needs it.

**Centralized Deadlock‐Detection Algorithms**

The Ho‐Ramamoorthy Algorithms

1. The Two‐Phase Algorithm

2. The One‐phase Algorithm

**Ho‐Ramamoorthy 2‐phase Algorithm**

* + 1. Each site maintains a status table of all processes initiated at that site: includes all resources locked & all resources being waited on.
    2. Controller requests (periodically) the status table from each site.
    3. Controller then constructs wait for graph (WFG) from these tables, searches for cycle(s).
    4. If no cycles, no deadlocks.
    5. Otherwise, (cycle exists): Request for state tables again.
    6. Construct WFG based only on common transactions in the 2 tables.
    7. If the same cycle is detected again, system is in deadlock.
    8. Later proved: cycles in 2 consecutive reports need not result in a deadlock. Hence, this algorithm detects false deadlocks.

**Ho‐Ramamoorthy 1‐phase Algorithm**

1. Each site maintains 2 status tables: resource status table and process status table.
2. Resource table: transactions that have locked or are waiting for resources.
3. Process table: resources locked by or waited on by transactions.
4. Controller periodically collects these tables from each site.
5. Constructs a WFG from transactions common to both the tables.
6. No cycle, no deadlocks.
7. A cycle means a deadlock.

**Distributed Deadlock‐Detection Algorithms**

In the distributed approach different nodes work together to detect deadlocks. No single point failure ( that is the whole system is dependent on one node if that node fails the whole system crashes) as workload is equally divided among all nodes. The speed of deadlock detection also increases.

**A Path‐Pushing Algorithm**

* 1. The site waits for deadlock‐related information from other sites
  2. The site combines the received information with its local transaction wait for (TWF) graph to build an updated TWF graph

For all cycles ‘EX ‐> T1 ‐> T2 ‐> Ex’ which contains the node ‘Ex’, the site transmits them in string form ‘Ex, T1, T2, Ex’ to all other sites where a sub‐transaction of T2 is waiting to receive a message from the sub‐transaction of T2 at that

Site

**Edge‐Chasing Algorithm**

**Chandy‐Misra‐Haas’s Algorithm:**

1. A probe(i, j, k) is used by a deadlock detection process Pi. This probe is sent by the home site of Pj to Pk.
2. This probe message is circulated via the edges of the graph. Probe returning to Pi implies deadlock detection.
3. Terms used:
   1. Pj is dependent on Pk, if a sequence of Pj, Pi1, Pi2, .., Pim, Pk exists.
   2. Pj is locally dependent on Pk, if above condition + Pj,Pk on same site.
   3. Each process maintains an array dependenti: dependenti(j) is true if Pi knows that Pj is dependent on it. (initially set to false for all i & j).

**Algorithm is as follows**

**Sending the probe:**

if

Pi is locally dependent on itself then deadlock.

else

for all Pj and Pk such that

(a) Pi is locally dependent upon Pj, and

(b) Pj is waiting on Pk, and

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

**Receiving the probe:**

if

(d) Pk is blocked, and

(e) dependentk(i) is false, and

(f) Pk has not replied to all requests of Pj,

then

begin

dependentk(i) := true;

if

k = i then Pi is deadlocked

else

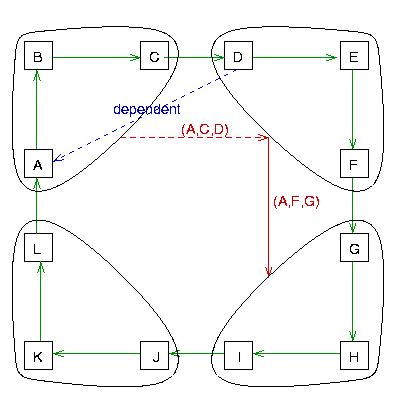
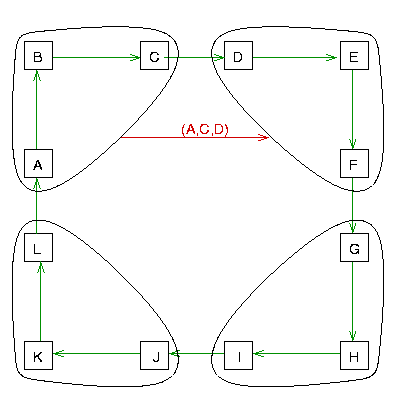
for all Pm and Pn such that

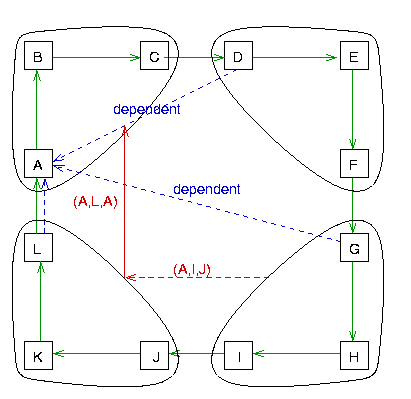
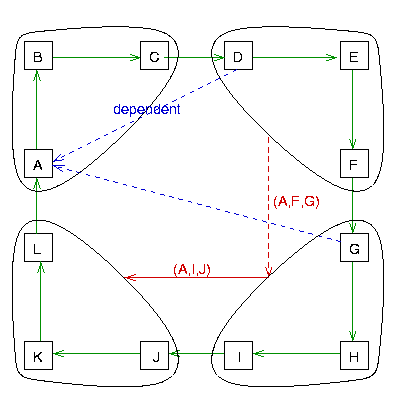
(a’) Pk is locally dependent upon Pm, and

(b’) Pm is waiting on Pn, and

(c’) Pm and Pn are on different sites, send probe(i,m,n) to the home site of Pn.

end.





**Advantages:**

* There is no need for special data structure. A probe message, which is very small and involves only 3 integers and a two dimensional boolean array *dependent* is used in the deadlock detection process.
* At each site, only a little computation is required and overhead is also low
* Unlike other deadlock detection algorithm, there is no need to construct any graph or pass nor to pass graph information to other sites in this algoirthm.
* Algorithm does not report any false deadlock (also called phantom deadlock).

**Disadvantages:**The main disadvantage of a distributed detection algorithms is that all sites may not aware of the processes involved in the deadlock this makes resolution difficult. Also, proof of correction of the algorithm is difficult.

**Hierarchical Deadlock‐Detection Algorithms**

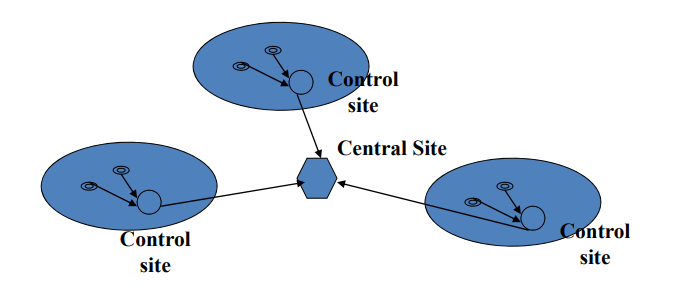
In hierarchical deadlock detection algorithm, sites are arranged in a hierarchical fashion and a site detects deadlocks involving only its descendant sites. Distributed deadlock algorithms delegate the responsibility of deadlock detection to individual sites while in hierarchical there are local detectors at each site which communicate their local wait for graphs(WFG) with one another.

**Approach:**  
Deadlocks that are local to a single site are detected at that site using their local WFG. Each site also sends its local WFG to deadlock detector at the next level. Thus, distributed deadlocks involving 2 or more sites would be detected by a deadlock detector in lowest level that has control over these sites.

In this approach, there are 2 methods to detect:

**1. Ho-Ramamoorthy algorithm:**

* Uses only two levels i.e. Master control nodes and Cluster control nodes.
* Cluster control nodes are used for detecting deadlock among their members and reporting dependencies outside their cluster to Master control node.
* The master control node is responsible for detecting inter-cluster deadlocks.
* Follows Ho-Ramamoorthy’s 1-phase algorithm. More than 1 control site organized in hierarchical manner.
* Each control site applies 1-phase algorithm to detect (intracluster) deadlocks.
* Central site collects info from control sites, applies 1-phase algorithm to detect intracluster deadlocks.



**Advantages:**

* If the hierarchy coincides with resource access pattern local to cluster of sites, this approach can provide efficient deadlock detection as compared to both centralized and distributed methods.
* It reduces the dependence on central sites thus, reducing the communication cost.

**Disadvantages:**

* If deadlocks are span over several clusters, this approach will be inefficient.
* It is more complicated to implement and would involve nontrivial modification to lock and transaction manager algorithms.