Volume 6, Issue 2, February 2016 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Different Deadlock Handling Strategies in Distributed Environment

Dr. Deepti Malhotra
Department of Computer Science & IT & Central University of Jammu,
Jammu and Kashmir, India

Abstract*—Deadlock is a highly unfavourable situation, the deadlock problem becomes further complicated if the underlying system is distributed. Deadlocks in distributed systems are similar to deadlocks in single processor systems, only worse. They are harder to avoid, prevent or even detect. They are hard to cure when tracked down because all relevant information is scattered over many machines.In deadlock situations the whole system or a part of it remains indefinitely blocked and cannot terminate its task. Therefore it is highly important to develop efficient control scheme to optimize the system performance while preventing deadlock situations. In this research paper, already existent deadlock prevention and detection schemes are discussed. The main objective of paper is to handle deadlock problem in Grid environment in order to preserve the data consistency and increase the throughput by maximizing the availability of resources and to ensure that all the resources available in the grid are effectively utilized.

Keywords— Grid Computing, Deadlock prevention, Deadlock avoidance, Deadlock detection, Resource Allocation.

I. INTRODUCTION

Deadlocks are important resource management problem in distributed systems because it can reduce the throughput by minimizing the available resources. In distributed systems, a process may request resources in any order, which may not know a priori, and a process can request a resource while holding others. If the allocation sequence of process resources is not controlled in such environments, deadlock can occur. A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set. Deadlock 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.

Deadlock refers to the coordination and concurrency problem where two or more processes are waiting indefinitely for the release of a shared resource [1]. The deadlock problem involves a circular waiting where one or more transactions are waiting for resources to become available and those resources are held by some other transactions that are in turn blocked until resources held by the first transaction are released. [2, 3, 4].Deadlock processes never terminate their executions and the resources held by them are not available to any other process.

Resource Vs. Communication deadlock.

Two types of deadlock have been discussed in the literature: resource deadlock and communication deadlock. In resource deadlocks, processes make access to resources (for example, data objects in database systems, buffers in store-and forward communication networks). A process acquires a resource before accessing it and relinquishes it after using it. A process that requires resources for execution cannot proceed until it has acquired all those resources. *A set of processes is resource-* deadlocked if each process in the set requests a resource held by another process in the set. In communication deadlocks, *messages are the resources* for which processes wait. Reception of a message takes a process out of wait - that is, unblocks it. *A set of processes is communication-* deadlocked if each process in the set is waiting for a message from another process in the set and no process in the set ever sends a message. To present the state of the art of deadlock detection in distributed systems, this article describes a series of deadlock detection techniques based on centralized, hierarchical, and distributed control organizations.

A 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 on n asynchronous processes $P_1, P_2, P_3, \ldots\ldots\ldots P_i, \ldots\ldots\ldots, P_n$ that communicate by message passing over the communication network. 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.

A process can be in two states, *running or blocked.* In the running/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.

A distributed system consists of a collection of processes which communicate with each other to achieve a common goal. These processes may run concurrently on separate physical processors which are not connected to any global memory. Local states of the processes are maintained in memories local to the processors running the processes. In absence of a shared memory, processes communicate through messages. The communication is asynchronous, and a message may take an arbitrary but finite amount of time to move from one process to another. When one process needs a resource currently being used by another process, it sends a request and waits for that resource to be released. It is assumed that every process is well-behaved in as much as once it acquires a resource, it releases it within a finite amount of time. It is fair to assume that no process has a prior knowledge of the future resource requirements of any process, including itself, in the total system. In such a resource sharing environment, deadlock is a potential danger. When a set of processes enter into such a state that each process waits for some other process in this set to release a resource, all the processes are blocked indefinitely. In a shared memory environment, a number of deadlock detection algorithms are available. However, in a distributed environment, the added complexity of these algorithms is due to the unpredictable propagation delays, and the consequent nonavailability of the global state.

In distributed systems, the state of the system can be modeled by a directed graph, called a *wait-for graph* (WFG).In a WFG nodes are processes and there is a directed edge from node P_1 to node P_2. If P_1 is blocked and is waiting for P_2 to release some resource. A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

Deadlock is an undesirable situation; some of the consequences of deadlock have been listed:

1. Throughput of the system is affected.
2. Performance of system suffers.
3. Deadlock in real time applications is not appreciable at any cost.
4. Entire or partial system is crippled.
5. Utilization of the involved resources decreases to zero.
6. Deadlock increases with deadlock persistence time.
7. Deadlock cycles do not terminate by themselves until properly detected and resolved.
8. No progress

Deadlock occurs in a resource allocation system because of the deficiency of system resources. An inappropriate execution order of the processes also causes the deadlock to occur in the resource allocation system of the distributed systems. The rest of the paper is organized as follows. Section 2 presents the background and related work. In Section3 all deadlock handling strategies are given. Section4 describes the different deadlock prevention schemes. Section5 presents the deadlock avoidance. In Section6 already existent different distributed deadlock detection approaches are discussed. Finally Section 7 concludes the paper.

II. RELATED WORK

Deadlock detection, resolution techniques and deadlock avoidance technology typically require pre empting resources, aborting processes or centralized resource management so they are inappropriate for many distributed real time systems that require the execution time of processes be predictable [5,6]. Therefore it would be more efficient if we look for an efficient deadlock prevention mechanism for distributed real time systems. The deadlock avoidance strategies make system performance to suffer [7], although deadlock detection may be effective, but it costs a lot in business transaction services [8] and detection of false deadlocks leads to wastage of resources. However deadlock resolution mechanisms are there but many real time applications are not well suited for run time deadlock resolution [9]. Although the deadlock prevention strategies are considered to be conservative and less feasible but in safety critical systems in which deadlocks may lead to serious results and [4] economic losses only deadlock prevention mechanisms can prove to be miraculous, for example in a highly automated semiconductor manufacturing system, the amount of time that wafers stay at a chamber is absolutely crucial. The occurrence of a deadlock extends the sojourn of wafers at chambers, possibly making all the wafers in such chambers scrapped. [4] As a result, it is essential to ensure that deadlock will never occur. Now a system cannot be made completely deadlock free, if resource sharing is high, but the occurrence of deadlocks can be prevented using deadlock prevention mechanisms.

In [10] describes certain theorems which propose sequence structures for processes and the resource allocation methods. It emphasizes on sequential allocation to prevent deadlocks in tool sharing systems. In [11] a service interaction model has been designed and the deadlock problem related with shared internet resources has been analyzed. It has proposed 2 solutions for deadlock prevention: a) to arrange an order for all shared web resources b) if one or more requested resource is busy transaction releases the resources it got before and restarts. Work of [6] suggests a deadlock prevention technique that involves requiring that managers in the RTC runtime system meet the AND-OR request conditions. This prevents deadlock in systems where processes can make AND-OR requests [12]. Describes a deadlock prevention mechanism which is based on atomic transaction approach to co allocate grid resources.

To detect deadlocks introduced by highly concurrent access, some researchers proposed graph-based detection algorithm [14] to detect deadlocks, and Omran Bukhres compared two different wait-for-graph (WFG) based algorithms (Central Controller and Distributed deadlock detection algorithms) for each throughput and performance evaluation [17]. Though deadlock detection may be effective, it costs a lot in business transaction services. While most researches focus on deadlock detection, for they argue that it is inappropriate to exploit prior knowledge in general purpose transaction processing to prevent deadlock, J Ezpeleta. et. al,[15] have proposed Petri net based deadlock prevention policy for flexible manufacturing systems.

On the other hand, after the deadlock is detected, one of the transactions in the cycle must be aborted. In a distributed system, inappropriate locks abortion may cause live-lock if retry mechanism is adopted. Though some researchers propose probabilistic analysis of time-out based mechanism [16] to detect global deadlock in distributed systems, the time of timeout itself or deadlock detection time should be the key factor for transaction throughput calculation, whatever timeout based or graph based detection [13].

III. DEADLOCK HANDLING STRATEGIES

Deadlock handling is 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. Next, we examine the complexity and practicality of the three deadlock-handling approaches in distributed systems.

Deadlock Prevention (statically make deadlocks structurally impossible)

Deadlock Avoidance (avoid deadlocks by allocating resources carefully)

Deadlock Detection (let deadlocks occur, detect them, and try to recover)

Deadlock prevention is commonly achieved by having a process acquire all the needed resources simultaneously before it begins executing or by pre-empting a process that holds the needed resource. This approach is highly inefficient and impractical in distributed systems.

In the *deadlock avoidance* approach to distributed systems, a resource is granted to a process if the resulting global system state is safe: Global state includes all the processes and resources of the distributed systems. Due to several problems, deadlock avoidance is impractical in distributed systems; in fact, it is not even used in single processor systems. The problem is that the banker's algorithm needs to know (in advance) how much of each resource every process will eventually need. This information is rarely, if ever, available. Hence, we will just talk about deadlock detection and deadlock prevention

Deadlock detection requires an examination of the status of process-resource interactions for the presence of cyclic wait. Deadlock detection in distributed systems seems to be the best approach to handle deadlock in distributed systems because prevention and avoidance are so difficult to implement.

IV. DEADLOCK PREVENTION SCHEMES

A method that might work is to order the resources and require processes to acquire them in strictly increasing order. This approach means that a process can never hold a high resource and ask for a low one, thus making cycles impossible. With global timing and transactions in distributed systems, two other methods are possible -- both based on the idea of assigning each transaction a global timestamp at the moment it starts. When one process is about to block waiting for a resource that another process is using, a check is made to see which has a larger times. We can then allow the wait only if the waiting process has a lower time stamp. The timestamp is always increasing if we follow any chain of waiting processes, so cycles are impossible --- we can used decreasing order if we like. It is wiser to give priority to old processes because they have run longer so the systems have larger investment on these processes. They are likely to hold more resources. A young process that is killed off will eventually age until it is the oldest one in the system, and that eliminates starvation.

Wait-die Vs. Wound-wait: As we have pointed out before, killing a transaction is relatively harmless, since by definition it can be restarted safely later.

Wait-die: If an old process wants a resource held by a young process, the old one will wait. If a young process wants a resource held by an old process, the young process will be killed.

Observation **:** The young process, after being killed, will then start up again, and be killed again. This cycle may go on many times before the old one release the resource. Once we are assuming the existence of transactions, we can do something that had previously been forbidden: take resources away from running processes. When a conflict arises, instead of killing the process making the request, we can kill the resource owner. Without transactions, killing a process might have severe consequences. With transactions, these effects will vanish magically when the transaction dies.

Wound-wait*:* (we allow preemption & ancestor worship) If an old process wants a resource held by a young process, the old one will preempt the young process – wounded and killed, restarts and wait. If a young process wants a resource held by an old process, the young process will wait.

False Deadlock: One possible way to prevent false deadlock is to use

- the Lamport's algorithm to provide global timing for the distributed systems.
- When the coordinator gets a message that leads to a suspect deadlock: It sends everybody a message saying "I just received a message with a timestamp *T* which leads to deadlock. If anyone has a message for me with an earlier timestamp, please send it immediately "When every machine has replied, positively or negatively, the coordinator will see that the deadlock has really occurred or not.

A. Two-Phase Commit Protocol

The two-phase commit protocol has been widely used as the protocol in the distributed transaction management environment. This protocol guarantees that the transaction is either successfully committed or not performed at all. The protocol work in two phases.

➢ *Two Transaction Managers request the same resource*

For example, T_A and T_B are sub-transactions of a local transaction T, and they both need to request the same resource R. Due to two phase commit protocol, R will be held by one of the two transaction managers T_{M_A} or T_{M_B} (maybe according to the first come first serve mechanism). We assume T_{M_A} for T_A gets the lock for R, and then it votes its approval to the coordinator saying that it is prepared. The coordinator now is waiting another participant $T_B's$ response. But TB requests the same resource R and must be blocked because R is held by T_A, so it must wait until T_A release its lock. In fact, if timeout period in phase one is not considered, T_A will not release its lock until it receives the coordinator's final decision. In this case, a wait-for cycle is formed and the guaranteed deadlock happens.

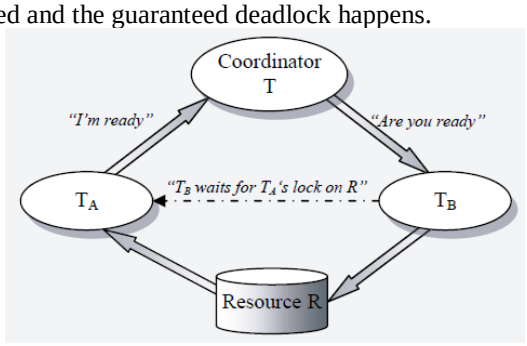

Fig1: Local Deadlock

Phase I : Prepare to Commit

- Transaction Co-ordinator(TC) is asking all the Transcation Manager(TM) of all the nested sub transaction available on different nodes of the n/w to prepare themselves ; i.e. to make lock on the resources which they needed.
- T_A request the Resource Manager $(RM)$for the resource R if it is available, then it will be allocated to transaction T_A . Then T_A will lock the resource R
- T_A gets the lock for R
- T_A votes it approval to the TC saying that it is prepared.
- TC now is waiting another participant $T_{B's}$ response. But T_B has given the request for the same resource R and must be blocked because Resource R is held by T_A, so it must wait until T_A release its lock.
- If Time out period in phase one is not considered, T_A will not release its lock until it reaches the co-ordinator's final decision.
- In this case, wait for cycle is formed and the guaranteed deadlock happens because it is the case where two or more participants request for the same resources.

Phase II

Transaction Co-ordinator(TC) sends a commit acknowledgement to all the participants (sub transaction) (i.e, T_A and T_B if all of them reach prepared state. Otherwise it tells all of them to rollback. Then all the Transaction Manager's commit or rollback as directed and return status to the Transaction Co-coordinator. As Resource R is held by T_A.So TC asks T_A and T_B to rollback.

This section describes deadlock in one nested transaction. This deadlock cannot be prevented and cannot be resolved even if detected.

➢ *More Resources in One local Transaction/ Several Resources are requested by two or more participants*

The above describes two TM s request the same resource. We can easily extend it to more resources in one local transaction. One TM is supposed to request more than one resource at one time, so if several resources are requested by two or more participants simultaneously, deadlock is also inevitable [8](un avoidable). Let's consider the distributed transaction $T_1$and T_2 , each with two sub-transactions T_{1_A}, T_{1_B} and T_{2_A}, T_{2_B}.

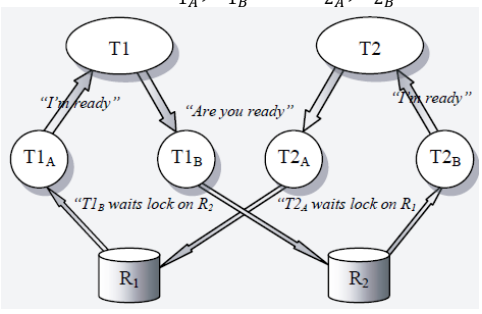

Fig2: Global Deadlock

Phase I : Prepare to Commit

- T_1 asks T_{1_A}, T_{1_B} and T_2 asks T_{2_A}, T_{2_B} to prepare their work respectively i.e. to make lock on the resources which they needed.
- T_{1_A} enters in prepare phase and request resource R_1 and set it lock on R_1.
- T_{2_B} enters in prepare phase and request resource R_2 and set it lock on R_2.
- T_{1_B} starts its work after receiving $T_1's$ instruction and begin to execute R_2 but it must wait for $R_2's$ lock.
- T_{2_A}, begin its work too and request R_1, but also needs to wait for $R_1's$ lock

In the above scenario, resource request flow can be shown as:-

$$T_{1_A} \xrightarrow{waits\ for} T_1's \text{ final decision} \rightarrow T_1 \xrightarrow{waits\ for} T_{1_B's}\ response\ of\ phase\ I \rightarrow T_{1_B} \xrightarrow{waits\ for} T_{2_B}\ to\ release\ R_2$$
$$\rightarrow T_{2_B}\ waits\ for\ T_2's \text{ final decision} \rightarrow T_2\ waits\ for\ T_{2_A}'s\ response\ of\ phase\ I$$
$$\rightarrow T_{2_A}\ waits\ for\ T_{1_A}\ to\ release\ R_1$$

The above scenario describes how deadlock is inevitable (unavoidable) when different transactions concurrently execute and have intersection of some common resources. As, we have seen that the wait-for graph cycle exists between the two transaction T_1 and T_2 ,each with two sub-transactions T_{1_A} , T_{1_B} and T_{2_A} , T_{2_B} .Unlike the usual transaction deadlocks, they often compete for several resources and each one holds a lock but waits for others. The simplest scenario contains only one resource but no transaction can proceed. This is caused by the two-phase commit protocol; for each sub-transaction has to compete for the same lock of resource in conflicting modes in phase *I*. So, it is obvious that the deadlock cannot be simply prevented by usual way [18]. On the other hand, once this deadlock is detected, we can simply choose one of the sub-transactions to abort or retry later, but it cannot solve the issue even the transaction aborts entirely and restarts again(This is why we call it guaranteed deadlock).

B. Replica Based local Deadlock Prevention

Replica based mechanism is also used to prevent local deadlocks. The basic idea is to produce a replication of the resource when more than one participants request it [8][19][20].Here the intelligent resource manager is used which is responsible for allocating resources for every applicant. Every transaction has a unique transaction id, including sub-transactions [8]. When a sub-transaction manager receives instruction from parent transaction manager/coordinator, it keeps the root transaction id passed by its parent and produces its own sub-id. So every participant knows the root id and we can distinguish if two sub-transactions belong to the same nested transaction. If two participants have the same root id, we don't care which level they are on.

In this type of mechanism, resource manager *RM* first receives $T_{1's}$ request and then T_2 for the same resource *R*. Step one, *RM* immediately caches the root transaction id from T_1, then it takes over $T_{M_{1's}}$ request and sets a lock on *R*. Soon T_2 comes. T_2 asks for *RM* to request *R* but it's already occupied by T_1. *RM* should not reject $T_{2's}$ request but checks its root transaction id first. It finds that more than one sub-transactions belong to the same local nested transaction, so it caches $T_{2's}$request and then creates a duplication of resource *R* (noted as R'). From now on, all requests from sub-transactions with the same root id should operate on resourceR'). But this time T_1will not hold its lock until phase two. In fact, *RM* will hold R's lock for root transaction and gives a new lock of R' toT_1. T_1 immediately commits its work on R' and returns prepared information to the coordinator if no exception occurs. Now *RM* takes $T_{2's}$ turn to occupy resource $R'$and work on it. After T_2 commit, the coordinator sends the second phase instruction to commit all of their works. T_1 or T_2 will pass this decision to *RM*. *RM* writes R' back to *R* and release its lock. At the end, all the participants return their final status to coordinator. On the other hand, if one of the participants cannot commit their work on replica R'), it should vote negative information to the coordinator. Then the coordinator notices all the participants to rollback their work. When *RM* receives this decision, it simply discards R').and release the lock on *R*.

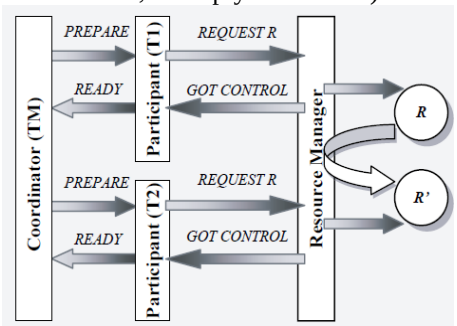

Fig3: Replica based mechanism for guaranteed deadlock [8]

C. Timestamps Based Restart Policy for Global Deadlock prevention

In this section, we describe global deadlock in distributed transaction environment. This is a more frequently discussed topic, for it is a system level problem (e.g. competing hardware devices, computing resource or database operation). Standard technology of deadlock avoidance expects sequentially resource access. For some resources, system could declare the most amount of requirements in advance. This is a pessimistic static allocation algorithm that needs to

exploit prior knowledge of transaction access patterns. If deadlock is allowed (e.g. it's rare to happen), detection and resolution are the main issues we should consider [12]. Timeout-based detection guesses that a deadlock has occurred whenever a transaction has been blocked for a long time. Wait-for graph is often used that if a circle exists in the graph, the transactions are regarded as deadlocked. But it's not easy to implement on distributed nodes. To resolve deadlock, we should choose a blocked transaction to be removed. For timeout-based detection, the blocked transaction can simply abort itself if a timeout period reaches. The best way of breaking all circles in a wait-for graph is to find a type of transactions with minimum conflicting cost. For example, we can choose the transaction that has done the least amount of work. In distributed transaction environment, business transactions are usually provided with different functional services. Each service is independent and already knows what resources it would request. That is to say, for every transaction, it is appropriate to exploit prior knowledge of related resources. Based on this premise, it has been prove that pre-assigning required resources should be a much efficient way to avoid global deadlocks.

To follow the two-phase commit protocol, we add a new phase called pre-check before every participant could prepare their work. At the first stage, the coordinator delivers all the user's requests to the participants, then these participants communicate with each resource managers to check the resources if they are available. If yes, the participants will hold them at the same time and return OK to the coordinator. It can be seen as a try-lock phase that means a transaction can go on if the resource lock is obtained, otherwise it should return false. After receiving all positive feedback from participants, the coordinator will enter in the standard two-phase commit process.

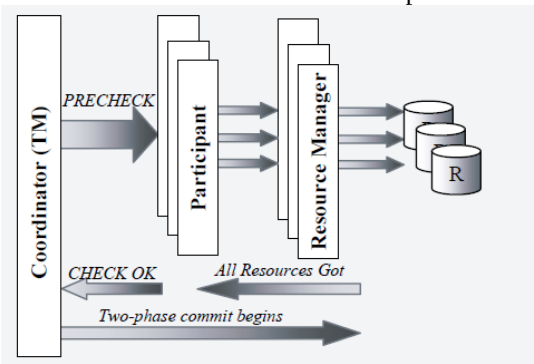

Fig4: Global Deadlock Resolution [8]

V. DEADLOCK AVOIDANCE

In deadlock avoidance strategy, a resource is granted to a process only if the resulting state is safe. (A state is safe if there is at least one execution sequence that allows all processes to run to completion.) Finally, in deadlock detection strategy, resources are granted to a process without any check. Periodically (or whenever a request for a resource has to wait) the status of resource allocation and pending requests is examined to determine if a set of processes is deadlocked. This examination is performed by a deadlock detection algorithm. If a deadlock is discovered, the system recovers from it by aborting one or more deadlocked processes. The suitability of a deadlock-handling strategy greatly depends on the application. Both deadlock prevention and deadlock avoidance are conservative, overly cautious strategies. They are preferred if deadlocks are frequent or if the occurrence of a deadlock is highly undesirable. In contrast, deadlock detection is a lazy, optimistic strategy, which grants a resource to a request if the resource is available, hoping that this will not lead to a deadlock.

For deadlock avoidance in distributed systems, a resource is granted to a process if the resulting global system state is safe (the global state includes all the processes and resources of the distributed system). The following problems make deadlock avoidance impractical in distributed systems:

(1) Because every site has to keep track of the global state of the system, huge storage capacity and extensive communication ability are necessary.

(2) The process of checking for a safe global state must be mutually exclusive. Otherwise, if several sites concurrently perform checks for a safe global state (each site for a different resource request), they may all find the state safe but the net global state may not be safe. This restriction severely limits the concurrency and throughput of the system.

(3) Due to the large numbers of processes and resources, checking for safe states is computationally expensive.

VI. DEADLOCK DETECTION APPROACHES

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

A. Detection of deadlocks

Detection of deadlocks involves addressing two issues: maintenance of the WFG and searching of the WFG for the presence of the cycles. Since, in distributed systems, a cycle 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 is maintained and the search for cycles is carried out, there are centralized, distributed and the hierarchical algorithms for deadlock detection in distributed systems [21].

A deadlock detection algorithm must satisfy the following two conditions:

- 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.
- no false deadlocks: The algorithm should not report deadlocks that do not exist *(called false deadlock)*. 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.

B. Resolution /Recovery of a detected deadlock

When a detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically.

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. There are several deadlock detection algorithms that propagate information regarding wait-for dependency 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 a detection of false deadlocks.

C. Strengths and weaknesses of :

- *Centralized algorithms.* In centralized deadlock detection algorithms, a designated site, often called the control site, has the responsibility of constructing the global state graph and searching it for cycles. The control site may maintain the global state graph all the time, or it may build it whenever deadlock detection is to be carried out by soliciting the local state graph from every site. Centralized algorithms are conceptually simple and are easy to implement. Deadlock resolution is simple in these algorithms - the control site has the complete information about the deadlock cycle, and it can optimally resolve the deadlock. However, because control is centralized at a single site, centralized deadlock detection algorithms have a single point of failure. Communication links near the control site are likely to be congested because the control site receives state graph information from all the other sites. Also, the message traffic generated by deadlock detection activity is independent of the rate of deadlock formation and the structure of deadlock cycles.
- *Distributed algorithms:* In distributed deadlock detection algorithms, the responsibility of detecting a global deadlock is shared equally among the sites. The global state graph is spread over many sites, and several sites participate in the detection of a global cycle. Unlike centralized algorithms, distributed algorithms are not vulnerable to a single point of failure, and no site is swamped with deadlock detection activity. Deadlock detection is initiated only if a waiting process is suspected to be part of a deadlock cycle. But deadlock resolution is often cumbersome in distributed deadlock detection algorithms because several sites may detect the same deadlock and may not be aware of other sites and/or processes involved in the deadlock. Distributed algorithms are difficult to design because sites may collectively report the existence of a global cycle after seeing its segments at different instants (though all the segments never existed simultaneously) due to the system's lack of globally shared memory. Also, proof of correctness is difficult for these algorithms.
- *Hierarchical algorithms:*In hierarchical deadlock detection algorithms, sites are arranged hierarchically, and a site detects deadlocks involving only its descendant sites. To efficiently detect deadlocks, hierarchical algorithms exploit access patterns local to a cluster of sites. They tend to get the best of both worlds: they have no single point of failure (as centralized algorithms have), and a site is not bogged down by deadlock detection activities that it is not concerned with (as sometimes happens in distributed algorithms). For efficiency, most deadlocks should be localized to as few clusters as possible; the objective of hierarchical algorithms will be defeated if most deadlocks span several clusters.

D. Different deadlock detection algorithms

In distributed algorithms all sites cooperate to detect a cycle in the state graph, which is distributed over several sites of the system. Deadlock detection can be initiated whenever a process is forced to wait, and it can be initiated either by the local site of the process or by the site where the process waits.

- *Goldman's algorithm.* Goldman's algorithm exchanges deadlock-related information in the form of an ordered blocked process list (OBPL), in which each process (except the last) is blocked by its successor. The last process in an OBPL may either be waiting to access a resource or be running. For example, OBPL P_1 , P_2 , P_3 , P_4 $represents$ the state graph in Figure 5. The algorithm detects a deadlock by repeatedly expanding the OBPL, appending the process that holds the resource needed by the last process in the list until either a deadlock is discovered (that is, the last process is blocked by a process in the list) or the OBPL is discarded (the last process is running). As an example, suppose in the system shown in Figure 6 process P_1 , initiates deadlock detection and sends OBPL P_1 , P_2, to process P_2 . When processP_2, receives the OBPL, it appends P_3 to the

OBPL and sends the new OBPL P_1 , P_2 , P_3 to P_3 .Likewise P_3 sends OBPL P_1 , P_2 , P_3 , P_4 to P_4 and P_4sends OBPL P_1 , P_2 , P_3 , P_4, P_5 to P_5. When P_5 receives the OBPL, it discards the OBPL because it is not blocked. Had P_5 been blocked by a P_1 , P_2 , P_3 *or* P_4,a deadlock would have been detected by P_5 .

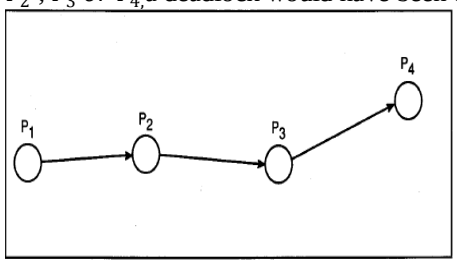

Fig 5: Example of OBPL

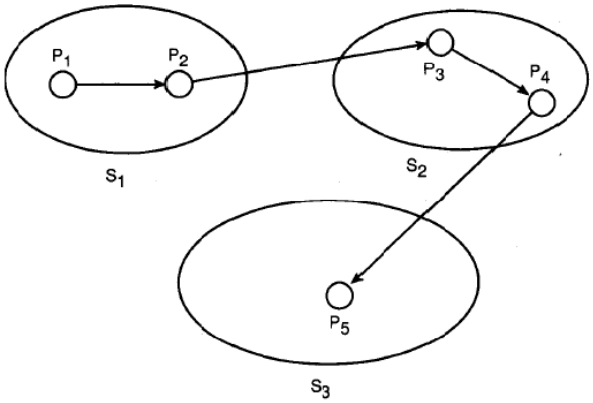

Fig 6: Example Goldman's algorithm

An advantage of Goldman's algorithm is that it does not require continuous maintenance of TWF graphs. It constructs an OBPL whenever deadlock detection is to be carried out. However, it requires that every process have at most one outstanding resource request.

- *Isloor-Marsland algorithm.* The "online" deadlock detection algorithm of Isloor and Marsland" detects deadlocks at the earliest possible instant - that is, at the time of making decisions about data allocation at the concerned site. It is based on the concept of reachable set. The reachable set of a node in the state graph is the set of all the nodes that can be reached from it. A process is deadlocked if the reachable set of the corresponding node contains the node itself. The algorithm detects deadlocks by constructing reachable sets and checking whether any node belongs to its own reachable set. To do this, every site maintains the system state graph and reachable sets for each node in the state graph; the reachable sets are continually updated whenever edges are added to or deleted from the state graph. Whenever a resource is allocated, a process is made to wait for a resource, or whenever a process releases a resource, the corresponding information is broadcast to all other sites. Therefore, if r changes per second occur in the state graph, then the algorithm requires *r (N-I)* messages per second for deadlock detection. However, the messages are short because they contain only an update to the state graph resulting from the execution of a request.
  All distributed deadlock detection algorithms have a common goal -to detect cycles that span several sites in a distributed manner - yet they differ in the ways they achieve this goal.

VII. CONCLUSIONS

Deadlocks are a fundamental problem in distributed systems. In distributed systems, a process may request resources in any order, which may not know a priori, and a process can request a resource while holding others. If the allocation sequence of process resources is not controlled in such environments, deadlock can occur. Deadlock 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. The paper reviews the literature of the different deadlock prevention as well as the detection mechanisms.

REFERENCES

[1] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B.Khailany,"*The Imagine stream processor*", Proc. International Conference of Computer Design, 2002,pg- 282–288.

[2] H.M.Deite1, *"An Introduction to Operating Systems",*Addision-Wesley Company, Second Edition, 199003-8575-6/04, IEEE.

[3] A.D.Kshemkalyani and M. Singhal,*"A One-Phase Algorithm to Detect Distributed Deadlocks in Replicated Databases"*, IEEE Transaction on Knowledge and Engineering, Vol. 11, No.6, November December, 1999.

[4] ZhiWu Li, NaiQiWu, and MengChu Zhou, "*Deadlock Control of Automated Manufacturing Systems Based on Petri Nets"*—A Literature Review‖, IEEE transactions on systems, man, and cybernetics—part c: applications and reviews, Digital Object Identifier 10.1109/TSMCC.2011.2160626, IEEE, 2011.

[5] Nisha Sharma ,Shivani,Saurabh Singh, "*Deadlock in Distributed Operating System*",International Journal of Research in Information Technolog(IJRIT),2013,pg 28-33.

[6] Victor Fay Wolfe, Susan Davidson & Insup Lee, "*Deadlock Prevention in the RTC Programming System for Distributed Real-Time Applications",* IEEE, 1993.

[7] S. Venkatesh, J. Smith, "*An evaluation of deadlockhandling strategies in semiconductor cluster tools*", IEEE Trans.Semiconductor Manufacturing, vol 18, pp. 197- 201, 2005.

[8] Lin Lou1 et.al, "*An Effective Deadlock Prevention Mechanism for Distributed Transaction Managemen*t", 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing DOI 10.1109/IMIS.2011.109 IEEE Computer society,2011.

[9] Lei Gao, et.al, "*Resource Optimization and Deadlock Prevention while Generating Streaming Architectures*",from Ordinary Programs,2011 NASA/ESA conference on adaptive hardware and systems(AHS-2011),IEEE,2011.

[10] Nagi Z. Gebraeel and Mark A. Lawley, "*Deadlock Detection, Prevention, and Avoidance for automated Tool Sharing Systems",* IEEE transactions on robotics and automation, vol. 17, no. 3, June 2001.

[11] Jieqi Ding, Han Zhu_, Huibiao Zhu and Qin Li, "*Formal Modeling and Verifications of Deadlock Prevention Solutions in Web Service Oriented System*", 2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems DOI 10.1109/ECBS.2010.48,IEEE computer society, 2010.

[12] Zhang Chuanfu Liu, "*A Deadlock Prevention Approach based on Atomic Transaction for Resource Co-allocation*", Proceedings of the First International Conference on Semantics, Knowledge, and Grid (SKG 2005), 2006, IEEE.

[13] Dotoli, M., Fanti, M.P. Iacobellis, G., "*Comparing deadlock detection and avoidance policies in automated storage and retrieval systems*", 2004 IEEE International Conference on Systems, Man and Cybernetics, vol 2, 2004,pp.1607-1612.

[14] Ahmed K. Elmagarmid., "*A survey of distributed deadlock detection algorithms*", ACM SIGMOD Record. Volume 15 Issue 3, Sept. 1986, Pages 37-45.

[15] Ezpeleta, J.; Colom, J.M.; Martinez, J., "*A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Robotics and Automation*" Volume 11, Issue 2, Apr 1995. page 173-184.

[16] Micha Hofri.,"*On timeout for global deadlock detection in decentralized database systems*", Information Processing Letters. Volume 51, Issue 6, 26 September 1994, Pages 295-302.

[17] Omran A. Bukhres,*"Performance Comparisons of Distributed Deadlock Detection Algorithms".* Proc. Int'l Conf. Data Eng., 1992.

[18] H. Wu, WN. Chin, AND J. Jaffar, "*An efficient distributed deadlock avoidance algorithm for the AND model"*, IEEE Transactions on Software Engineering, 28 (2002), pp. 18-29.

[19] Ajay D. Kshemkalyani , Mukesh Singhal, "*A One-Phase Algorithm to Detect Distributed Deadlocks in Replicated Databases, IEEE Transactions on Knowledge and Data Engineering"*, v.11 n.6, p.880- 895, November 1999.

[20] W. D. Zhu, J. Cerruti, A. A. Genta, H. Koenig, H. Schiavi, and T. Talone, "*IBM Content Manager Backup/Recovery and High Availability: Strategies, Options and Procedures*, IBM Redbook", Mar. 2004.

[21] M. Singhal, "*Deadlock detection in distributed systems"*, IEEE Computer, November, 1987, 37-48.

