Multi-cycle Deadlock Detection Algorithm for Distributed Systems

Mohammad Ariful Islam¹, Md. Serajul Islam², Shifat Jahan³

¹Programmer, ICT Wing, Bangladesh Election Commission, Dhaka, BANGLADESH
²³Assistant Programmer, National ID Registration Wing, Bangladesh Election Commission, Dhaka, BANGLADESH

ARTICLE INFO

Received: Nov 10, 2016
Accepted: Dec 15, 2016
Published: Dec 22, 2016

ABSTRACT

Deadlock handling is an important component of transaction management in a database system. Though a lot of works have already done for deadlock detection on distributed system. This paper proposes a multi-cycle deadlock detection and recovery mechanism. Our proposed algorithm has modified the probe based distributed algorithm for deadlock detection such as CMH algorithm. But CMH algorithm had some limitation. It can only detect single cycle. But some situation a node is responsible for multi-cycle deadlock detection. In this situation, it can’t detect Multi-cycle deadlock. Besides it can only detect deadlock when the initiator node involved in the cycle. Another algorithm named MC2DR has worked on CMH algorithm. But their probe message contains four fields for deadlock detection which takes more space, but our algorithm has reduced the MC2DR algorithm probe message field and increased the efficiency of this algorithm. It can detect multi-cycle and also detect which node is responsible for multi-cycle deadlock and kills it.

Key Words: Distributed system, deadlock, algorithm, deadlock algorithm

INTRODUCTION

In concurrent computing, a deadlock is a state in which each member of a group of actions is waiting for some other member to release a lock. Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization. A distributed deadlock can be defined as cyclic and inactive indefinite waiting of a set of processes for exclusive access to local or remote resources of the system. Such a deadlock state persists until a resolution action is taken. Persistence of a deadlock has two deficiencies: first, all the resources held by deadlocked processes are not available to any other process and the second, the deadlock persistence time gets added to the
response time of each process involved in the deadlock. Therefore, the problem of prompt and efficient detection and resolution is an important fundamental issue of distributed systems (Chandy et al, 1983; Sinha and Natarajan, 1985; Razzaque et al, 2007). The state of a distributed system that represents the state of process-process dependency is dynamic and is often modeled by a directed graph called Wait-for-Graph (WFG), where each node represents a process, and an arc is originated from a process waiting for another process holding that resource. A cycle in the WFG represents a deadlock. From now on, processes in the distributed system will be termed as nodes in this paper.

![Diagram](image)

**Fig. 1:** Wait-for-Graphs, some example node-node dependency scenarios

As shown in Fig. 1(a), node ‘2’ is called *successor of parent* node ‘1’ and {7, 8, 11, 12} is called a *deadlocked* set of processes. Such state graph is distributed over many sites, may form multiple dependency cycles and thereby many nodes are blocked indefinitely, like node {11, 12}. The most widely used distributed deadlock detection scheme is *edge-chasing* that uses a short message called *probe*. If a node suspects the presence of a deadlock, it independently initiates the detection algorithm, creates a *probe* message and propagates it outward to all of its *successor* nodes. Deadlock is declared when this *probe* message gets back to the initiator i.e., forming a dependency cycle {1, 2, 3, 4, 5, 6}. The key limitation of these algorithms is that they are unable to detect deadlocks whenever the initiator does not belong to the deadlock cycle. In the worst case, this may result in transmission of almost $N^2$ messages to detect a deadlock, where $N$ represents the number of blocked nodes in the WFG. Algorithms proposed in (Lee, 2002; Lee and Kim, 1995; Farajzadeh et al, 2005; Razzaque et al, 2007) overcome this problem but some of them detect phantom deadlocks and the rests can’t detect deadlocks in the case that a single node is involved in multiple deadlock cycles. Our proposed algorithm, introduces a modified *probe* message structure, a *victim* message structure and for each node a *probe* storage structure. The contributions of our algorithm includes: (i) it can detect all deadlocks reachable from the initiator of the algorithm in single execution, even though the initiator does not belong to...
any deadlock, (ii) it can detect multi-cycle deadlocks \textit{i.e.}, deadlocks where a single process is involved in many deadlock cycles, (iii) it decreases the deadlock detection algorithm initiations, phantom deadlock detections, deadlock detection duration and the number of useless messages and (iv) it provides with an efficient deadlock resolution method. The rest of the paper is organized as follows.

**RELATED WORKS**

**CMH algorithm**

![CMH algorithm diagram](image)

\textbf{Fig. 2:} Example illustrating the CMH distributed deadlock detection algorithm

The algorithm is conceptually simple and works in the following manner. When a process that requests for resources fails to get the requested resources and times out, it generates a special probe message and sends it to the processes holding the requested resources. The probe message contains the following fields.

- The identifier of the process just blocked
- The identifier of the process sending this message
- The identifier of the process to whom this message is being sent

**Probe-based distributed algorithm for deadlock detection**

On receiving the message a process checks whether it is waiting for any other resource. If not, then it must be using the requested resource and it ignores the message. Otherwise, it passes the message to the process holding the resource for which it is waiting. But, before passing the message it changes the fields of the probe message as follows:

- The first field is left unchanged.
- The second field to its own process identifier.
- The third field is changed to the identifier of the process that will be the new recipient of this message.

If the probe message returns to the original sender (the first and the third field are same), a cycle exists, and the system is in deadlock.

Let us illustrate the algorithm with the help of the simple shown in figure 2. Notice that this figure depicts suppose that process p1 gets blocked when it requests forth resources held by p3. Therefore p1 generates a probe message (p1, p1, p3) and sends it to p3 when p3 receives this message it discovers that it is itself blocked on processes p2 and p5. Therefore p3 forwards the probe message (p1, p3, p2) and (p1, p3, p5) to processes p2 and p5 respectively, when p5 receives the probe message it ignores it because it is not blocked.
any other processes. However when p2 receives the probe message, it discovers that it is itself blocked on processes p1 and p4 respectively. Since the probe returns to its original sender p1 a cycle exists and the system is deadlocked.

The key concept of CMH algorithm (Chandy et al., 1983; Chandy and Misra, 1982) is that the initiator propagates probe message in the WFG and declares a deadlock upon receiving its own probe gets back. Sinha and Natarajan (1985) proposed the use of priorities to reduce the no. of probe messages. Choudhary et al. (1989) found some weaknesses of this algorithm and Kashemkalyani and Singhal (5) proposed further modifications to this and provided with a correctness proof. Kim et al. (1997) proposed the idea of barriers to allow the deadlock to be resolved without waiting for the token to return, thereby reducing the average deadlock persistence time considerably. None of the above algorithms can detect deadlocks in which the initiator is not directly involved. Suppose in Fig. 1, node ‘1’ initiates algorithm execution, the deadlock cycle {7, 8, 12, 11, 7} can’t be detected by any of the above algorithms.

As because in all those algorithms, deadlock is declared only if the initiator ID matches with the destination ID of the probe message. Lee in (2002) proposed a probe based algorithm that exploits reply messages to carry the information required for deadlock detection. As a result, the probe message does not need to travel a long way returning to its initiator and thereby time and communication costs are reduced up to half of those of the existing algorithms. Even though this algorithm can detect deadlocks where the initiator node is not directly involved, but except the initiator, no other nodes will be able to detect deadlocks. For instance in Fig. 1(a), if node 1, 7 and 12 initiate algorithm executions one after another in order with little time intervals, then the same deadlock cycle {7, 8, 12, 11, 7} will be detected by all of them, which is a system overhead. Lee and Kim in (1995) proposed an algorithm to resolve the deadlock in single Execution even though the initiator doesn’t belong to any deadlock. This is achieved by building a tree through the propagation of the probes and having each tree node collects information on dependency relationship (route string) among its sub tree nodes to find deadlocks based upon the information. But, we found several drawbacks and in capabilities of this algorithm. First, all deadlocks Reachable from the initiator may not be resolved by a single execution of the algorithm since a deadlock may consist of only tree and cross edges in the constructed tree. Second, deadlock detection algorithm works correctly for single execution of the algorithm, but it would detect phantom deadlocks in case of multiple executions. To prove the above statement, let we consider in Fig. 1(b), node ‘e’ initiates algorithm at sometime later than node ‘a’ and in addition to dependency edges shown in the figure, there are two other edges, one from ‘e’ to ‘c’ and another from ‘c’ to ‘b’. Node ‘b’ forwarded probe message initiated by ‘a’ and then as the steps of algorithm (Lee and Kim, 1995) phantom deadlock will be detected if it receives probe message initiated by ‘e’ before receiving one from ‘d’. This is happened due to appending unique bits for each successor (0, 1, 2, ..., m) to its own path string. Our algorithm resolves this problem by appending system wide unique ID of individual nodes. MC2DR: Multi-cycle Deadlock Detection and Recovery Algorithm and The algorithm (Razzaque et al., 2007) proposed by the same authors, criticized Lee and Kim in (1995) for not giving any deadlock resolution method and proposed priority based victim detection. This may lead to starvation for low priority nodes. Farajzadeh et. al. in (2005) considered simultaneous execution of many instances of the algorithm but what happens if a single process is involved in multiple deadlock cycles was not illustrated. Simulation has not also been carried out. Hence, their algorithm is so weak that even in simple example scenarios it can’t detect deadlocks. Suppose in Fig. 1(c),
node ‘3’ stores the initiator ID (1) and route string (00) of the probe message initiated by node ‘1’, forwards the message with necessary modification to node ‘4’ and ‘6’, and then receives another probe message initiated by node ‘7’, at this stage according to their algorithm node ‘3’ replaces the stored route string with new one (0). Due to this incorrect replacement, node ‘3’ will not be able to detect deadlock cycle {3, 4, 5, 3} although it does exist. It is not necessary to unfold that such incorrect replacement of existing route string might also cause the probe message infinitely moving around the cycle and increase the number of algorithm initiations as well as message passing.

MC2DR: Multi-cycle Deadlock Detection and Recovery Algorithm for Distributed Systems

<table>
<thead>
<tr>
<th>InitID</th>
<th>VictimID</th>
<th>DepCnt</th>
<th>RouteString</th>
</tr>
</thead>
</table>

Fig. 3(a): Probe Message

<table>
<thead>
<tr>
<th>InitID</th>
<th>VictimID</th>
</tr>
</thead>
</table>

Fig. 3(b): Victim Message

A node can be in any of the two states at any time instant: active and blocked. If the requested lock is not available i.e., it is being used by some other node then the requesting node will enter the blocked state until the resource is obtained. Deadlock occurs when a set of nodes wait for each other for an indefinite period to obtain their intended resources. The probe message used for deadlock detection in MC2DR consists of four fields as shown in Fig. 3(a). The first field InitID contains the identity of the initiator of the algorithm. VictimID is the identity of the node to be victimized upon detection of the deadlock and DepCnt of a node represents the number of successors for which it is waiting for resources. The fourth field, RouteString, contains the node IDs visited by probe message in order. At each node, there will be a probe message storage structure, named ProbeStorage, same as that of probe message for temporary storage of probes. At most one probe message is stored in ProbeStorage at a particular time. MC2DR is history independent and upon detection of a deadlock, respective probe message is erased from storage. The node that detects the deadlock sends a victim message to the node found to be victimized for deadlock resolution. This message will also be used for deleting probes from respective storage entries. This short message contains just first two fields of probe message as shown in Fig. 3(b).

**Strategies for algorithm initiation.** A node initiates the deadlock detection algorithm execution if it waits for one or more resources for a predefined timeout period, T0 and its probe storage is empty. But if T0 is shorter, then many nodes may be aborted unnecessarily, and if it is longer, then deadlocks will persist for a long time. Choosing an appropriate value of T0 is the most critical issue as because it does depend on several system environment factors such as process mix, resource request and release patterns, resource holding time and the average number of locks held (locked) by nodes. As the above dependency factors change dynamically, the value of T0 is also set dynamically in our algorithm. If the value of T0 is decreased (or increased) at a node for each increase.

**Probe message forwarding policy.** On reception of probe message, a node first checks the emptiness of its ProbeStorage. If it is found to be empty (i.e., till now no probe message is forwarded by this node), then it compares its own DepCnt value with probe’s DepCnt value. If this node’s DepCnt is higher, then probe’s VictimID and DepCnt values are
updated with this node’s ID and DepCnt values respectively; otherwise the values are kept intact. Before forwarding the probe message to all successors of this node, probe’s RouteString field is updated by appending this node’s ID at last of existing string (i.e., concatenate operation). One copy of updated probe message is saved in ProbeStorage of this node. For example, in Fig. 3(a), node ‘0’ has initiated execution and send probe message (0, 0, 1, “0”) to its successor node 1. As node 1’s ProbeStorage is empty and DepCnt value is 2, it has updated the probe message, stored the modified probe (0, 1, 2, “01”) in ProbeStorage and forwarded to its successors 2 and 4. Nodes 2, 3, 4, 5 and 6 have updated only the RouteString field of the probe message and forwarded to their successors.

Fig. 4(a): Example WFGs in our network model with edges labeled by probe messages

**Deadlock detection.** If the ProbeStorage is nonempty, the node first goes for checking whether the stored route string is a prefix of the received probe’s route string. If it is, deadlock is detected and otherwise the probe message is discarded. Probe message is also discarded by a node that has just detected a deadlock. So, MC2DR can detect deadlock cycle at any node right at the moment the traveled path of probe message makes a dependency cycle. Node ‘1’ in Fig. 4(a) has eventually got back its forwarded probe and detected one of the two deadlock cycles {1, 2, 3, 1} and {1, 4, 5, 6, 1}. If the probe message for deadlock cycle {1, 2, 3, 1} is received first then that from node ‘6’ is discarded or vice-versa. Again, if node ‘4’ would be the successor of node ‘6’ then two deadlock cycles {1, 2, 3, 1} and {4, 5, 6, 4} would be detected (by a single probe) by node 1 and 4 respectively, even though none of them is the initiator.

**Strategies for deadlock resolution.** A deadlock is resolved by aborting at least one node involved in the deadlock and granting the released resources to other nodes. When a
deadlock is detected, the speed of its resolution depends on how much information about it is available, which in turn depends on how much information is passed around during the deadlock detection phase. We opine that rather than victimizing initiator node or node with the lowest priority; it is better to victimize a node which is more likely to be responsible for multiple deadlocks. To make the above notion a success, MC2DR selects the node with highest $\text{DepCnt}$ value as victim and the deadlock detector node sends a victim message to all successors. If the detector node is not the initiator, it also sends the victim message to all simply blocked (node that is blocked but not a member of deadlock cycle) nodes. On reception of this message, the victim node first forwards it to all of its successors and then releases all locks held by it and kills itself, other nodes delete deadlock detection information from their ProbeStorage memories. Node ‘1’ in Fig. 4(a) has killed itself as because it has the highest $\text{DepCnt}$ value amongst the members in any of the cycles. Node ‘1’ is not the initiator, so it has also sent the victim message to simply blocked node ‘0’. Node ‘3’ and node ‘6’ stop further propagation of victim message.

**PROPOSED ALGORITHM**

**Informal Description of the Algorithm**

Our proposed algorithm has three fields VicID, NumSucc and RouteString where

1. VicID is the identity of the node that has just blocked,
2. NumSucc is the number of successors of a node for which it is waiting for resources and
3. The third field RouteString contains the node IDs visited by probe message in order.

<table>
<thead>
<tr>
<th>VicID</th>
<th>NumSucc</th>
<th>RouteString</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1,0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1,2</td>
<td>0,1</td>
</tr>
<tr>
<td>4</td>
<td>1,2</td>
<td>1,0</td>
</tr>
<tr>
<td>6</td>
<td>1,2</td>
<td>1,0</td>
</tr>
</tbody>
</table>

Fig. 4(b): Example WFGs in our network model with edges labeled by probe messages

**Probe message forwarding policy:** On reception of probe message, a node first checks whether it is waiting for any other resource. If not, then it must be using the requested resource and it ignores the message. Otherwise, it passes the message to the process holding the resource for which it is waiting but, before passing the message it changes the fields of the probe message as follows:

- if it has more successor than sending

node— probe’s VicID and NumSucc values are updated with this node’s ID and NumSucc values respectively; The first field VicID will be its own process identifier. The second field NumSucc will be its number of successors. Otherwise, the values are kept intact before forwarding the probe message to all successors of this node and probe’s RouteString field is updated by appending this node’s ID at last of existing string.
For example, in Fig. 4(b), node ‘0’ has initiated execution and send probe message (0, 1, “0”) to its successor node 1. As node 1's ProbeStorage is empty and NumSucc value is 2, it has updated the probe message, stored the modified probe (1, 2, “01”) in ProbeStorage and forwarded to its successors 2 and 4. Nodes 2, 3, 4, 5 and 6 have updated only the RouteString field of the probe message and forwarded to their successors.

**Deadlock detection:** If the Probe Storage is nonempty, the node first goes for Checking whether the stored route string is a prefix of the received probe’s route String. If it is, deadlock is detected and otherwise the probe message is discarded. Probe message is also discarded by a node that has just detected a deadlock. Node ‘1’ in Fig. 4(b) has eventually got back its forwarded probe and detected one of the two deadlock cycles {1, 2, 3, 1} and {1, 4, 5, 6, 1}. If the probe message for deadlock cycle {1, 2, 3, 1} is received first then that from node ‘6’ is discarded or vice-versa. Again, if node’4’ would be the successor of node ‘6’ then two deadlock cycles {1, 2, 3, 1} and {4, 5, 6, 4} would be detected (by a single probe) by node 1 and 4 respectively, even though none of them is the initiator.

**CONCLUSION**

Even though the deadlock persistence duration detection of all deadlocks reachable from the initiator including multicycle deadlocks. Algorithm initiation is increased highly in some rarely occurred exceptional conditions; it is more reliable and robust as because it does detect real deadlocks on a single execution of the algorithm and ensures policy and deadlock resolution mechanism makes it more efficient.

**REFERENCES**


