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

Md. Abdur Razzaque, Md. Mamun-Or-Rashid, and Choong Seon Hong

Department of Computer Engineering, Kyung Hee University
1, Seocheon, Giheung, Yongin, Gyeonggi, Korea, 449-701
m_a_razzaque@yahoo.com, mamun@networking.khu.ac.kr, cshong@khu.ac.kr

Abstract. Even though there have been many research works on distributed deadlock detection and recovery mechanisms, the multi-cycle deadlock problems are not extensively studied yet. This paper proposes a multi-cycle deadlock detection and recovery mechanism, named as MC2DR. Most existing algorithms use edge-chasing technique for deadlock detection where a special message called *probe* is propagated from an initiator process and echoes are sent back to it that carries on necessary information for deadlock detection. These algorithms either can't detect deadlocks in which the initiator is indirectly involved or a single process is involved in multiple deadlock cycles. Some of them often detect phantom deadlocks also. MC2DR defines new structures for *probe* and *victim* messages, allows any node to detect deadlock dynamically, which overcomes the aforementioned problems and increases the deadlock resolution efficiency. The simulation results show that our algorithm outperforms the existing probe based algorithms.

1 Introduction

A distributed deadlock can be defined as cyclic and inactive indefinite waiting of a set of processes for exclusive access to local and/or remote resources of the system. Such a deadlock state persists until a deadlock resolution action is taken. Persistence of a deadlock has two major 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 of a deadlock is an important fundamental issue of distributed systems [1, 3, 10]. [1] 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 [13, 14]. A cycle in the WFG represents a deadlock. From now on, processes in the distributed system will be termed as nodes in this paper.

[1] This research was supported by the MIC, Korea, under the ITRC support program supervised by the IITA, Grant no - (IITA-2006 - (C1090-0602-0002)).

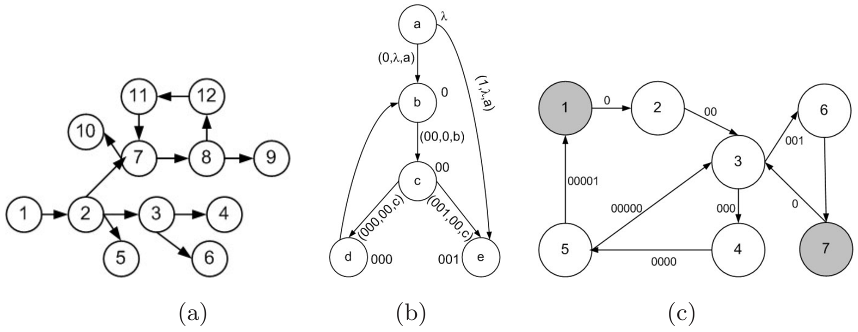


Fig. 1. Wait-for-Graphs, some example node-node dependency scenarios

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 [12, 13].

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 is 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 [7, 8, 9, 10, 11] 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, MC2DR, introduces a modified *probe* message structure, a *victim* message structure and for each node a *probe* storage structure. The contributions of MC2DR 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 *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. A thorough study and critics of state-of-the-art probe based algorithms are presented in section 2. Section 3 introduces the network and computation models, section 4 and 5 describe the proposed algorithm and its correctness proof respectively. Simulation and performance comparisons are presented in section 6 and section 7 concludes the paper.

2 Related Works

The key concept of CMH algorithm [1, 2] is that the initiator propagates *probe* message in the WFG and declares a deadlock upon receiving its own *probe* gets back. Sinha and Natarajan [3] proposed the use of priorities to reduce the no. of *probe* messages. Choudhary et. al. [4] found some weaknesses of this algorithm and Kashemkalyani and Singhal [5] proposed further modifications to this and provided with a correctness proof. Kim Y.M. et. al. [6] 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.

S. Lee in [7] 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 back 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.

S. Lee and J. L. Kim in [8] proposed an algorithm to resolve 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 subtree nodes to find deadlocks based upon the information. But, we found several drawbacks and incapabilities 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 [8] 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.

The algorithm [9] proposed by the same authors, criticized [8] for not giving any deadlock resolution method and proposed priority based victim detection. This may lead to starvation for low priority nodes. N. Farajzadeh et. al. in [10, 11] 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.

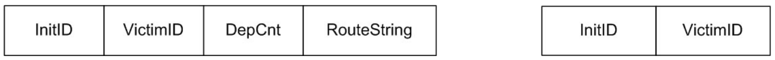
3 Network and Computation Model

3.1 Network Model

We assume that each data object in our distributed system is given a unique lock that can't be shared by more than one node. A node can make request for locks residing either at local or remote sites. A request for a lock is processed by the lock manager to determine whether the lock can be granted. If the requested lock is free, it is granted immediately; otherwise, the lock manager sends a reject message to the requesting node and inserts the requesting node ID into the waiting list for the lock. A reject message carries the identifier of the node which is currently holding the resource. Upon receiving a reject message for any of the locks requested, the node remains *blocked* until the lock is granted, and inserts the lock holder's ID into the successor list.

It is assumed that there might be one or more nodes running on each site as we are concerned with both distributed and local deadlocks. There is no shared memory in the system, nodes communicate with each other by message passing. Each node is uniquely identified by its {site id:process id} pair. But for simplicity of explanation of the proposed algorithm, we have assigned unique integer numbers (0, 1, 2, 3,...,m) to all nodes as shown in Fig. 3.

Another important characteristic of our network model is that the underlying channel is FIFO, *i.e.*, messages are arrived at the destination nodes in the order in which they were sent from the source nodes without any loss or duplication. Message propagation delay is arbitrary but finite. MC2DR is proposed for multi-resource model where a single process may make multiple lock requests at a time. In this model, the condition for a *blocked* node to get *unblocked* is expressed as a predicate involving the requested resources. In the WFG, there will be no self-loop *i.e.*, no node make requests for resources held by itself.



(a) Probe Message (b) Victim Message

Fig. 2. Structure of Probe and Victim Messages

3.2 Computation Model

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. 2(a). The first field *InitID* contains the identity of 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. 2(b).

4 Proposed Algorithm

4.1 Informal Description of the Algorithm

We have found in our study that a correct and efficient deadlock detection algorithm needs to consider a number of parameters and corresponding strategies.

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, T_0 and its *probe* storage is empty. But if T_0 is shorter, then many nodes may be aborted unnecessarily, and if it is longer, then deadlocks will persist for a long time. Choosing appropriate value of T_0 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 T_0 is also set dynamically in our algorithm. If the value of T_0 is decreased (or increased) at a node for each increase

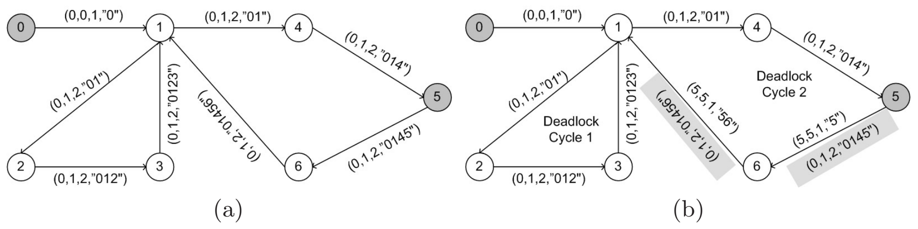


Fig. 3. Example WFGs in our network model with edges labeled by probe messages

(or decrease) of *DepCnt* (defined in section 3.2) our simulation results show that average deadlock detection duration and the average number of algorithm executions are decreased. Dynamic updating of T_0 value also increases the probability that the node involves in multiple deadlock cycles would initiate the algorithm execution.

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.

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. 3(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 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 *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. 3(a) has killed itself as because it has the highest *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.

One exceptional condition. Even though the probability of appearing the following exceptional condition in our algorithm is very low, we have to block it for the sake of algorithm's correctness. One exception of the above mechanism in *probe* message discard policy is that only if any node Q is the initiator of another *probe* message then rather than discarding, Q will keep the message in a buffer space and waits for Q's probe to return back. If Q's probe gets back to it or Q receives a *victim* message within average deadlock detection period T_d (computed as in [9]), then the buffered *probe* is discarded, otherwise it is forwarded to Q's all successors. The last one is a worst case situation in MC2DR where node Q defers from detecting any deadlock. Let we consider in Fig. 3(b), immediately after node '0' has initiated execution, node '5' starts another one and node '1' receives second *probe* from node '6' after it has forwarded the first one, so the second *probe* is discarded. Node '5' has blocked first *probe* from further forwarding as it is the initiator of second *probe* and waited for T_d. In this scenario, the chance is very high that within T_d, node '5' receives *victim* message sent by node '1' (on detection of deadlock cycle 1) and discards first *probe* message, deletes second *probe* from it's *ProbeStorage* and forwards *victim* message to its successor node '6'; otherwise, node '5' forwards first *probe* as described in previous part and keeps itself away from detecting any deadlock.

4.2 Deadlock Detection and Resolution Algorithm

For a particular node *i*, pseudo code for deadlock detection and recovery algorithm is presented in Fig. 4.

```
Algorithm_Initiation() {
    int W; //waiting time for a particular resource
    probe p; allocate memory for p;
    if (W > To && ProbeStorage == NULL) {
        p = Create_Probe(i); Send_Probe(i, p);}
}

probe Create_Probe(node i) {
    p.InitID = i.ID;
    p.VictimID = i.ID; p.DepCnt = i.DepCnt;
    p.RouteString = i.ID; return (p);
}

Send_Probe(node i, probe p){
    int j = i.DeptCnt;
    while (j){ //sends probe to all successors, j
        send (j, p); j--; }
}

Receive_Probe(probe p){
    if (ProbeStorage == NULL){
        if( p.DepCnt < i.DepCnt) { p.VictimID = i.ID; p.DepCnt = i.DepCnt;}
        p.RouteSting = p.RouteString + i.ID;
        Send_Probe(i,p);  }
    else if( i.RouteString is prefix of p.RouteString){
        Deadlock is detected.
        //send victim message to all successors and simply blocked nodes
        Send_Victim(j, p.VictimID);  }
    else if (i is the initiator of another probe)
        Exception_Handling(p);
    else { Discard (p);} //probe message is discarded
}

Receive_Victim(int VictimID){
    //forward victim message to all successors
    Send_Victim(j, VictimID);
    if(VictimID == i.ID){  // this node is vicitimized
        Release (All locks held by this node);
        Kill (this node); }
    else {Erase Probe message from ProbeStorage;}
}

Exception_Handling(probe p){
    int Td; //avg. deadlock detection period
    put p in a buffer space;
    wait for Td and check for i's receiving probe
    if(i's probe is received){ Discard (p);}
    else { p.RouteSting = p.RouteString + i.ID;
            Store(p); //Store p in ProbeStorage
            Send_Probe(i, p); }
}
```

Fig. 4. Pseudo code of MC2DR

5 Correctness Proof

Theorem 1. *If a deadlock is detected, the corresponding nodes are really in deadlocked state. Phantom deadlocks are not detected.*

Proof. Let's prove it using proof by contradiction. A set of nodes could be detected as in a phantom deadlock, when the detection algorithm misinterprets the existence of a deadlock cycle. This type of misinterpretation can be taken place in MC2DR only and if only at least any two nodes have the same ID. In the case, a false deadlock is detected even though the traveling path of *probe* message does not make a cycle. But this contradicts with our network model, described in section 3.1.

Theorem 2. *A single deadlock cycle would never be detected by more than one node.*

Proof. Again, we use proof by contradiction. A single deadlock cycle could be detected by multiple nodes if and only if any detection algorithm allows multiple probes to be forwarded by a single node. But, MC2DR defers it by storing other probes into node's *ProbeStorage*. Only in exceptional condition case (described in section 4.1), it is allowed but at the same time the forwarder node is kept away from detecting any deadlock. Hence, there is no chance of multiple detections of a deadlock. This is a distinctive contribution of MC2DR.

Theorem 3. *Multiple deadlocks can be detected by a single probe message.*

Proof. Let's prove it by using proof by contradiction. Detection of multiple deadlock cycles by a single *probe* is prohibited whenever a detection algorithm does not allow any node other than the initiator to detect deadlock cycles. It is further restricted by priority based probing, where the higher priority nodes discard the *probe* message initiated by low priority nodes. As described in section 4.1 under the heading "Deadlock Detection", MC2DR defers both the above methods and here the intermediary nodes forward the *probe* message towards multiple directions, which enables MC2DR to detect multiple real deadlocks exist in the system.

6 Simulation and Performance Comparison

We have run the simulation program using fixed sites (20) connected with underlying network speed of 100Mbps, but with varying multiprogramming level (MPL), ranges from 10 to 40. The interarrival times for lock requests and the service time for each lock are exponentially distributed for each node. Only write operations to data objects are considered. To increase the degree of lock conflicts, a relatively small size database in comparison with the transaction size has been chosen. Experiments have been carried out in both the light and heavily loaded environments. We have found that deadlocks increases linearly with the

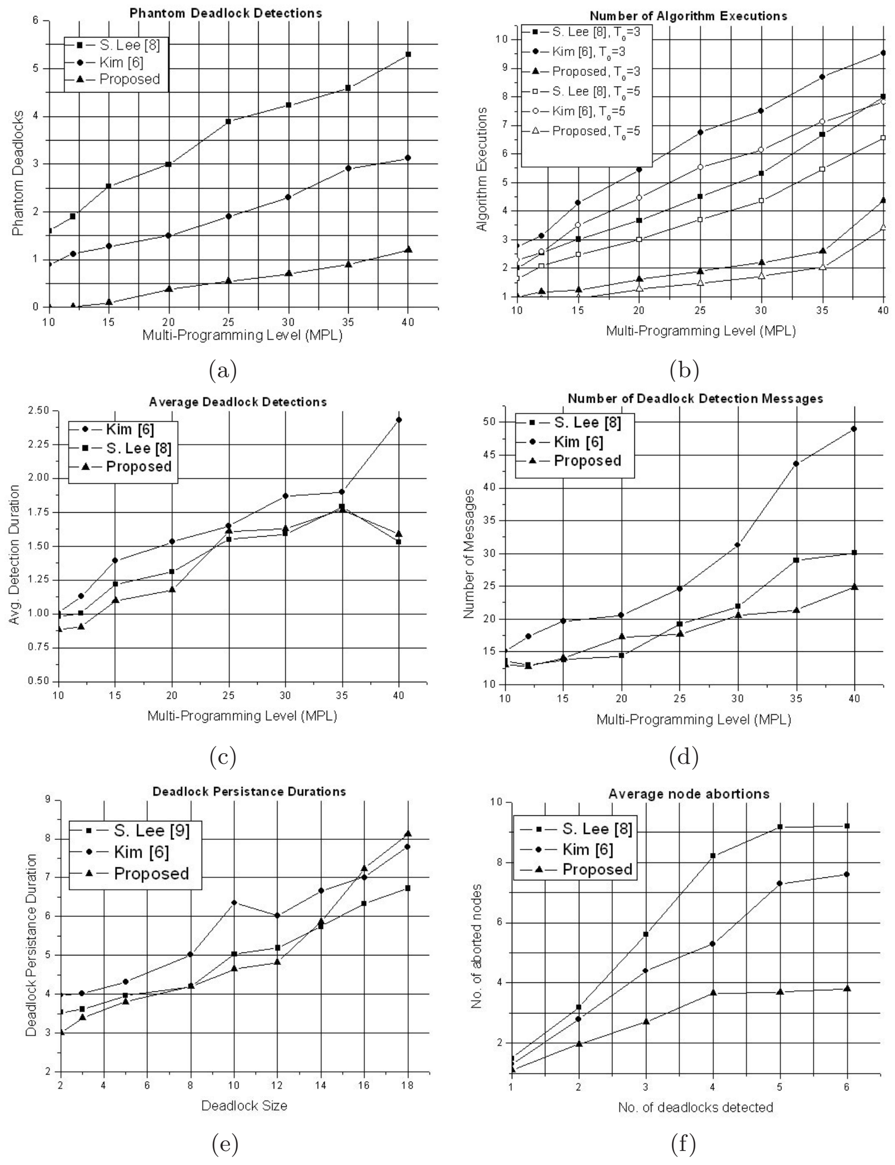


Fig. 5. Simulation results showing performance comparisons

degree of multiprogramming and exponentially with average lock holding time. We have compared the performance of MC2DR with that of Y.M. Kim et. al.'s algorithm [6] and S. Lee's algorithm [8].

As shown in Fig. 5(a), the number of phantom deadlocks detected by MC2DR is 5 times less than Lee's algorithm [8] and almost 3 times less than Kim's algorithm [6]. This is because phantom deadlocks can only be detected by MC2DR in the case of unusual excessive large size of deadlocks, not for any misinterpretation of nonexistence deadlock cycles. Fig. 5(b) shows the mean number of algorithm initiations with varying timeout periods and multiprogramming levels, MC2DR requires about 56% less number of initiations than [6] and about 45% less than that of [8]. This result shows congruence with the theoretical expectation as because MC2DR dynamically controls algorithm initiations (see section 4.1).

Average deadlock detection duration resulted from Kim's algorithm [6] and our algorithm is almost same for higher MPL values (>25) and is slightly less than that from Lee's algorithm [8], as shown in Fig. 5(c). For low to medium MPL values (<20) MC2DR takes 30% to 50% less time than Kim's algorithm [6]. It is observed that the deadlock detection duration increases with MPL until the number of nodes reaches to 30 for most graphs and then become almost flat. The reason behind this could be the increase of *simply blocked* nodes with MPL and the increased chance of algorithm initiations by nodes having higher *DepCnt* values. As shown in Fig. 5(d), Kim's algorithm passes 2 times more messages than MC2DR and almost 1.5 times than Lee's algorithm [8] for higher MPL values (>33). This is because in some cases Kim's algorithm requires multiple executions for detecting a single deadlock. For lower to medium MPL values (10-30), it is observed that proposed algorithm and [8] need almost same number of messages to detect deadlocks.

Fig. 5(e) indicates that the mean deadlock persistence duration is nonlinear for all the algorithms. In case of exceptional conditions arisen in MC2DR, the persistence time may be much longer. Graphs of Fig. 5(f) shows that the number of aborted nodes is very less in MC2DR as compared to [6] (almost 50% decreased) and [8] (almost 65% decreased). This is happened as because in MC2DR, node having highest *DepCnt* value is aborted and it is more likely that abortion of single node might untie more than one deadlock cycles. It's a key contribution of MC2DR.

7 Conclusion

Even though the deadlock persistence duration of MC2DR is increased highly in some rarely occurred exceptional conditions, it is more reliable and robust as because it does detect real deadlocks on single execution of the algorithm and ensures detection of all deadlocks reachable from the initiator including multi-cycle deadlocks. Algorithm initiation policy and deadlock resolution mechanism of MC2DR make it more efficient.

References

1. Chandy, K.M., Misra, J., Haas, L.M.: Distributed Deadlock Detection. ACM Transaction on Computer Systems. 144–156 (1983)
2. Chandy, K.M., Misra, J.: A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, Ottawa, Canada, pp. 157–164. ACM Press, New York (1982)

3. Sinha, M.K., Natarajan, N.: A Priority Based Distributed Deadlock Detection Algorithm. IEEE Trans. Software Engg. 11(1), 67–80 (1985)
4. Choudhary, A.N., Kohler, W.H., Stankovic, J.A., Towsley, D.: A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution. IEEE Transactions on Software Engg. 15(1), 10–17 (1989)
5. Kashemkalyani, A.D., Singhal, M.: Invariant Based Verification of a Distributed Deadlock Detection Algorithm. IEEE Transactions on Software Engineering 17(8), 789–799 (1991)
6. Kim, Y.M., Lai, T.W., Soundarajan, N.: Efficient Distributed Deadlock Detection and Resolution Using Probes, Tokens, and Barriers. In: Proc. Int'l Conf. on Parallel and Distributed Systems, pp. 584–591 (1997)
7. Lee, S.: Fast Detection and Resolution of Generalized Distributed Deadlocks. In: EUROMICRO-PDP 2002 (2002)
8. Lee, S., Kim, J.L.: An Efficient Distributed Deadlock Detection Algorithm. In: Proc. 15th IEEE Int'l Conf. Distributed Computing Systems, pp. 169–178 (1995)