Message-Optimal Algorithm for Detection and Resolution of Generalized Deadlocks in Distributed Systems

Selvaraj Srinivasan and R. Rajaram
Department of Information Technology
Thiagarajar College of Engineering
Madurai, 625015, India
E-mail: ssnit@tce.edu, rrajaram@tce.edu

Keywords: distributed deadlock, generalized model, deadlock detection, wait-for graph, deadlock resolution

Received: October 15, 2010

In this paper, we present a new algorithm to detect and resolve generalized deadlocks in distributed systems. The algorithm constructs a distributed spanning tree by diffusing probes along the edges of the Wait-For Graph (WFG) and collects a reply that carries the dependency information of processes to determine a deadlock. Unlike the previous algorithms, it performs reduction whenever it receives a reply from an active process. Moreover it isolates termination detection from deadlock detection, and terminates the execution once it detects a deadlock. It has a worst-case time complexity of d+2 and message complexity of e+2n; where n is the number of nodes, e is the number of edges and d is the diameter of the WFG. Correctness proof and performance analysis for the algorithm are also provided. Furthermore, it minimizes the message length and message overhead associated with deadlock resolution as compared with the existing algorithms.

Povzetek: Članek preučuje problem smrtnega objema v porazdeljenih sistemih.

1 Introduction

In a distributed computing environment, if a process needs a resource on the remote site for its computation, it sends a request message to the desired site. If the resource is available, it will be granted to the requesting process immediately; otherwise, the requesting process waits indefinitely until its request is granted. This will lead to a deadlock in distributed systems where a set of processes wait indefinitely for each other to satisfy their requests. Since deadlock reduces the resource availability and throughput, it should be detected and resolved promptly. However, deadlock is very difficult to detect as well as resolve in distributed systems due to the presence of multiple sites. In general, the interdependency among the distributed processes is modeled by a directed graph known as the Wait-For Graph (WFG) [1,2]; where each node represent a process and an edge from a node 'i' to node 'j' indicates that process 'i' is requested a resource from process 'j' and process 'j' is not granted a resource to process 'i'. A deadlock is defined differently depending upon the underlying resource request model such as Single-Resource model, AND model, OR model and P out-of Q model [1,7]. In the Single Resource and AND model, a process needs all requested resources to continue its execution. Hence, the presence of cycle in the WFG implies a deadlock. In the OR model, a process proceeds the execution only if any of the requested resource is granted. Therefore, the presence of knot is necessary to determine OR deadlock. In the P out-of Q model, a process makes requests for Q resources and remains blocked until it is granted any P resource. Since AND and OR model are the special case of P out-of Q model, it is also referred as generalized request model. A generalized deadlock corresponds to a deadlock in the generalized request model. The generalized request model is quite common in many domains such as resource management in distributed operating systems, communicating sequential processes and quorum consensus algorithms in distributed databases [11,12,16]. A cycle in the WFG is necessary but not sufficient condition whereas a knot is sufficient but not necessary condition for a generalized deadlock. Since detection of generalized deadlock requires the detection of a complex topology in the WFG, only few generalized deadlock detection and resolution algorithms [4,5,7,8,10,12,15,16] have been proposed in the literature. Most of them have used the diffusion computing technique [1] in which a distributed computation is initiated by a single node and joined by other nodes only after receiving a message. The generalized deadlock detection algorithms are grouped into two categories namely centralized and distributed algorithms based on the existence of the WFG. In the centralized algorithms, the initiator maintains the entire information to determine a deadlock whereas in the distributed algorithm the information is spread across multiple sites.

1.1 Literature survey

In general, the distributed algorithms [4,5,7,10,12] have used 'record and reduce' principle to detect the generalized deadlocks. According to the technique, the algorithm records the consistent snapshot of distributed

WFG and performs reduction later to determine a deadlock. The algorithm proposed by Bracha and Toueg [4] consists of two phases. In the first phase, the initiator records a snapshot by propagating the probes along the edges of the WFG. In the second phase, the algorithm simulates the granting of resources to determine a deadlock. The second phase is nested within the first phase. It exchanges 4e messages in 4d time units, where e is the number of edges and d is the diameter of the WFG. By following the approach in [4], Wang.et.al [5] developed another algorithm in which an explicit termination technique is used to detect the end of the first phase. The second phase begins only after the first phase is finished. Although it reduces the time complexity of [4] into 3d+1, it needs 6e messages to detect a deadlock. Moreover, both [4] and [5] have failed to resolve deadlocks. The algorithm in [10] records as well as reduces the WFG simultaneously to determine a deadlock. It records a consistent snapshot of distributed WFG in the outward sweep and reduces in the inward sweep in a single phase. It uses 4e-2n+4l messages in 2d time units to find out whether the initiator is deadlocked, where n is the number of nodes and l is the number of leaf nodes in the WFG. However it deals with the complications introduced because the reduction of node in the inward sweep can begin much before the state of all WFG edges incident at that node have been recorded in the outward sweep. And it needs O(e) messages to resolve deadlocks. The algorithm in [11,12] uses lazy evaluation technique by which the reduction of a process in a snapshot is delayed until the initiator detects the termination. Although it minimizes the message complexity into 2e as with the previous algorithms, it uses variable sized messages with the length of O(e) . Since the initiator knows the resource requirement of all deadlocked processes, it minimizes the message overhead associated with deadlock resolution unlike in [10]. The algorithm in [19] achieves the time and message complexity of [12] with fixed sized messages. Unlike [12], it performs the reduction before the initiator terminates the execution of the algorithm. In general, distributed algorithms require two or more rounds of message transfer along the entire edges of the WFG. Hence, they need at least 2d time units to detect deadlock in the worst case.

In the centralized algorithms [8,15,16], the initiator maintains the Local Wait-For Graph (LWFG) to detect a deadlock. The initiator of the algorithm in [8] collects a reply from each process in its reachable set exactly once. Based on the information in replies, it incrementally constructs the WFG locally to determine a deadlock. It needs only 2n messages with the length of O(n) to find out deadlock. However, it has a time complexity of 2d like the distributed algorithms. In addition, it may abort nodes that are not deadlocked and needs O(n) messages to resolve deadlocks. The algorithm in [15] constructs the LWFG by using the ancestor-descendent relationship between the processes in replies. It uses less than 2e messages in 2d time units to detect a deadlock. It reduces the message length into O(e-n+m) where m indicates the number of nodes that are not associated with any non-tree edges in the spanning tree induced by the algorithm. However, it needs additional technique to assign a unique path string to each node in the WFG and to interpret the path strings for constructing LWFG at the initiator. In contrast to [8], it resolves all deadlocks reachable from the initiator. The initiator of the algorithm in [16] collects the dependency information of all nodes to determine the deadlocked processes. It has almost half the time complexity of previous algorithms to detect a deadlock. It needs less than 2e messages in d+2 time units for detecting all deadlocked processes. However it needs additional technique to optimize the message length at each node and requires messages with the length of O(d) in the worst case. Hashemzadeh proposed an algorithm [17,18] based on history based edge chasing technique in which the initiator declares a deadlock once it finds its existence in the message. However, it significantly minimizes the message overhead associated with the executions of concurrent instances and deadlock resolution. We do not consider the algorithms based on edge chasing techniques [17,18] in this paper.

We propose a new centralized algorithm to detect and resolve distributed deadlock in generalized model. Our algorithm improves the message complexity and message size of previous algorithms. The initiator of the algorithm constructs the distributed spanning tree (DST) by propagating probes (CALL messages) along the edges of the WFG. Once a process receives the probe, it sends a reply that carries its dependency information directly to the initiator. However, the initiator performs reduction immediately after receiving a reply from an active process and receives a reply at most twice from each node in its reachable set unlike the earlier algorithms. At the end of termination, it declares all the nodes whose resource requirements are not met as deadlocked. We have formally proved the correctness of the proposed algorithm. It has a worst-case time complexity of d+2 time units and message complexity of less than e+2n, where d is the diameter, n is the number of nodes and e is the number of edges in the WFG. Further, it has a data traffic complexity of O(n) in the worst case. Since it selects a victim without using additional messages, it considerably simplifies deadlock resolution.

Although the proposed algorithm have some similarities with [16], it differs from Lee's algorithm [16] and pervious centralized algorithms [8,15] in the following aspects:

1. The algorithm performs reduction whenever it receives a reply form an active node whereas Lee's algorithm [15,16] performs reduction only after it detects the termination.
2. The algorithm in [15,16] uses an explicit mechanism to reduce the message length, whereas the proposed algorithm does not require any additional techniques.
3. The initiator of the proposed algorithm builds a directed spanning tree of the WFG, whereas the initiator of Chen [8] algorithm does not consider any structural property of the WFG.

4. Unlike in [16], the initiator of the proposed algorithm receives a reply from all nodes in its reachable set at most twice.

The rest of this paper is divided into five main sections. In Section 2, we describe the key definitions and assumptions about the system model and the problem definition. In Section 3, we present the algorithm along with an example. In Section 4, we prove the correctness of the algorithm. In Section 5, we analyze the performance of the algorithm and compare it with that of previous algorithms. Finally, we conclude the paper in section 6.

2 System Model

Although we follow the computation model in [4, 8, 10, 12, 16], we describe it for completeness of this paper. The system has 'n' processes and each one has a unique identity. There is a logical channel between any two processes. The processes can communicate only by message passing. The message delays are arbitrary but finite. The messages are delivered to the destination in the same order as the sender sends them. The messages are neither lost nor duplicated and the entire system is fault-free. The messages are grouped into namely computation and control messages in the system. The computation messages including REQUEST, REPLY, CANCEL and ACK are generated as a result of application's execution. However, the control messages including CALL, REPORT and WEIGHT, which will be discussed in the section 3, are generated by the execution of the deadlock detection algorithm. Both computation and control messages are time stamped based on the requesting process's logical lock [3]. Thus the time stamp of ACK or REPLY should be matched exactly with the corresponding REQUEST message.

A process state is active or blocked at any instant. When a process 'i' makes a generalized request and blocks, the unblocking condition of its request is denoted as a function F_i. For example, $F_i=A\wedge(B\vee C)$ denotes that process 'i' requires a resource from process A and a resource from either process B or C. Function F_i is evaluated in the following manner: substitutes true for a node id in F_i if it has received a REPLY, indicating granting of that request from that node; otherwise, substitutes false for it. Then, evaluate the function F_i. A process unblocks when a sufficient number and combination of its requests to make F_i true are granted.

An active process can send both computation and control messages whereas the blocked process can send either control messages or ACK. When process 'i' blocks on p_i out-of q_i requests, it sends REQUEST message to q_i processes in OUT_i. Therefore, OUT_i gives the domain of F_i. Upon receiving a REQUEST message from 'i', process 'j' records <i, t_block_i> in IN_j and sends an ACK message to the sender of the message. If process 'j' is active, it sends a REPLY message to process 'i' and subsequently removes <i, t_block_i> from IN_j. Once a node is unblocked, it withdraws the remaining requests it had sent earlier but not yet granted.

Each process 'i' maintains the following variables to keep track of its state in the WFG. The initial value of each variable is given within parenthesis.

$parent_i$: the process identifier from which 'i' has received the first probe (NULL)

IN_i : the set of processes which are directly blocked on 'i' (φ)

OUT_i : the set of processes for which 'i' is waiting (φ)

F_i : the condition for unblocking a process 'i'

We denote the set of processes in IN_i as predecessors and the set of processes in OUT_i as successors of process 'i'. A blocked process cannot withdraw any one of its requests spontaneously. And it could not abort any requests abnormally. These two assumptions are essential to ensure that the algorithm records a consistent snapshot. We use the terms process and node interchangeably throughout this paper.

2.1 Problem statement

A generalized deadlock exists in the system if the requesting conditions of one or more processes can never be satisfied. The formal description of deadlock is provided as in [16].

Definition 1: Let evaluate(F_i) be a recursive operation evaluated based on the following:

1. evaluate (i) = evaluate (F_i),
2. evaluate (F_i) = true, for active node 'i',
3. evaluate ($P\vee Q$) = evaluate (P) $\vee$ evaluate (Q)
4. evaluate ($P\wedge Q$) = evaluate (P) $\wedge$ evaluate (Q)

where P and Q are nonempty AND/OR expressions of node identifiers. A generalized deadlock exists in the system if and only if the following topology exists in the WFG.

Definition 2: A generalized deadlock is a sub graph (D,K) of WFG (V,E) where

$\forall i\in D(\neq\varphi)$, evaluate ($F_i$) = false,

No message for computation is under transmission between any nodes in D

Therefore, all processes in D are blocked forever and the resource requirement of processes that do not belong to D can be satisfied at any instant. It should be necessary to abort a node in D to resolve a deadlock.

A distributed deadlock detection algorithm should satisfy the following two correctness conditions:

Liveness: If a deadlock exists, the algorithm will detect it within finite time.

Safety: The algorithm does not report any false deadlock.

3 The Proposed Algorithm

In this section, we present the basic idea behind the execution of single instance our deadlock detection algorithm. Then, we present the algorithm formally and provide an example.

3.1 The description

We assume that the initiator 'i' initiates the deadlock detection algorithm. It includes the unblocking function (F_i) in the set UC_{init} and sends CALL message to each one of its successor 'j' in out_i. If node 'j' receives the first CALL message, it becomes the child of the sender and sends REPORT message that carries the unblocking function (F_j) to the initiator directly. Further, it propagates the CALL message to its own successors. However, if a node that has already joined the execution of current instance receives the second and subsequent CALL message, it does send a reply immediately. Those nodes send a WEIGHT message to the initiator only after receiving CALL messages from all its predecessors. Hence, it minimizes the message overhead to detect a deadlock.

Whenever the initiator receives a REPORT message from a blocked node 'i', it includes a tuple (i, F_i, num_pred_i) in the set UC_{init}. At the same time, if it receives a REPORT message from an active node 'i', it includes 'i' in the set A_{init} and attempts to evaluate all unblocking functions in the set UC_{init}. It performs the evaluation in the following manner: Select a tuple (i, F_i, num_pred_i) from the set UC_{init} and check if the node identifiers in the set A_{init} are sufficient to make F_i as true. If it happens, it includes 'i' in the set A_{init} and removes the corresponding tuple from the set UC_{init}. This process is repeated continuously until there is no more unblocking function in the set UC_{init} can be simplified as true. If the algorithm unblocks all nodes in the set UC_{init} during evaluation, it terminates the execution immediately; otherwise, it continues the execution until it detects the termination based on weight distribution technique. Once the algorithm terminates the execution, it declares all the nodes that have not been reduced in the set UC_{init} as deadlocked.

The algorithm detects the termination based on the weight distribution method like in [10,17]. According to the method, the initiator distributes a weight of one to its successors through CALL messages. When a node receives the first CALL message, it distributes the weight in the message among its successors. However, it accumulates the weight in all subsequent CALL messages until it receives the CALL messages from all its predecessors. It then returns the weight to the initiator through a WEIGHT message. It is accomplished as follows. Each node 'i' has a variable 'num_pred_i' that counts its predecessors. Whenever it receives a CALL message, it decreases the count by one. Hence, the num_pred_i at a node becomes zero signifying that it receives the CALL message from all its predecessors. Whenever the initiator receives a WEIGHT message, it sum ups the weights. The algorithm terminates when the weight at the initiator becomes one.

In a dynamic environment, the algorithm may report a false deadlock due to the presence of phantom edges. Let us consider a phantom edge from node 'i' to node 'j'. This implies that when node 'j' receives a CALL message from node 'i', node 'j' has sent a REPLY to node 'i'. In the proposed algorithm, it is resolved by as follows. Whenever node 'j' receives the CALL message from node 'i', it checks whether node 'i' is in IN_j. If $i \notin IN_j$, it sends an ALERT message to the initiator. Upon receiving the ALERT message, the initiator evaluates f_i by substituting j as true. If node 'i' unblocks during the evaluation, it is included in the set A_{init} and initiates the evaluation of other nodes in the set UC_{init}.

3.2 Formal specification

A formal description of the proposed algorithm executed at node 'i' is presented below. The initial value is given inside the parenthesis.

Data Structure of a node 'i'

$parent_i$: node id (NULL); /* a node from which a CALL has been first received */

$weight_i$: float (0); /* the weight value of 'i' */

in_i : set of nodes (IN_i); /*the set of predecessors of 'i' */

out_i : set of nodes (OUT_i); /*the set of successors of 'i' */

f_i : AND-OR Expression (F_i); /*the condition for 'i' to be active */

num_pred_i: integer ($|IN_i|$); /* the number of predecessors of 'i' */

Additional Data Structures at initiator

UC_{init} → a set of unblocking functions which contains tuples of the form (i, f_i, num_pred_i) where f_i denotes the unblocking condition of a node 'i' (φ).

A_{init} → a set of active nodes (φ)

$weight_{init}$ → the accumulated weight value (0)

$victim_{init}$ → the node identifier to be aborted to resolve the deadlock (φ).

Message Formats

CALL(initiator, sender, weight): Sent by node 'sender' carrying the identifier of the initiator and the weight value for the receiver of this message.

REPORT(sender, f_{sender}, num_pred_{sender}): Sent by node 'sender' as a response to first CALL message carrying the unblocking condition and its number of predecessors.

WEIGHT(sender, $weight_{sender}$):Sent by node 'sender' after receiving CALL messages from all its predecessors.

ALERT (sender, $weight_{sender}$): Sent by node 'sender' after receiving CALL messages through a phantom edge.

I. When a node 'i' initiates the algorithm

$initiator_i := i$;

$parent_i := i$;

$UC_{init} := UC_{init} \cup \{(i, f_i, num\_pred_i)\}$;

send CALL(initiator, i, $1 / |out_i|$) to each $j \in out_i$

II. Upon receipt of CALL(initiator, j, $weight_j$) from j

begin

num_pred$_i$ --;
if (parent$_i$ = NULL ∧ j∈in$_i$) **then**
/* Step II.1 */
parent$_i$:=j;
initiator$_i$:= initiator;
send REPORT(i, f$_i$, num_pred$_i$) to initiator$_i$
if (|out$_i$| >0) **then** /* Step II.1.1 */
send CALL(initiator$_i$,i,weight$_j$/|out$_i$|) to each k∈out$_i$
else if (parent$_i$ ≠ NULL ∧ j∈in$_i$) **then** /* Step II.2 */
if (i=initiator) **then** /* Step II.2.1 */
weight$_{init}$= weight$_{init}$ + weight$_j$;
else if (num_pred$_i$ = 0) **then** /* Step II.2.2*/
send WEIGHT(i, weight$_i$) to initiator$_i$
else
weight$_i$:= weight$_i$ + weight$_j$;
else if (j∉in$_i$) **then** /* Step II.3*/
send ALERT(i, weight$_j$) to initiator$_i$
end

III. Upon receipt of REPORT(i, f$_i$,num_pred$_i$) from i :
begin
if (f$_i$ = φ) **then**
A$_{init}$:= A$_{init}$ ∪ {i};
evaluation();
else
UC$_{init}$:= UC$_{init}$ ∪{(i, f$_i$, num_pred$_i$)};
end

IV. Upon receipt of WEIGHT(i, weight$_i$) from i :
begin
weight$_{init}$:= weight$_{init}$ + weight$_i$;
if (weight$_{init}$ =1 ∧ UC$_{init}$!= φ**)** **then**
resolution(); // Declare a Deadlock
end

V. procedure evaluation()
begin
for each i ∈ UC$_{init}$ **do**
begin
if (evalvate(i.f$_i$) = true) **then**
A$_{init}$:=A$_{init}$ ∪ {i};
UC$_{init}$:= UC$_{init}$ – { i ,f$_i$, num_pred$_i$}
if (UC$_{init}$ = φ) **then**
No deadlock; exit;
else
evaluation();
end for
end

VI. procedure resolution()
begin
count:=0;
repeat
for each i ∈ UC$_{init}$ **do**
begin
if (i.num_pred ≥ count) **then**
count:= i.num_pred;
victim$_{init}$:= i.id;
end for
send ABORT to victim$_{init}$;
UC$_{init}$:=UC$_{init}$–{(victim$_{init}$,f$_{victim}$,num_pred$_{victim}$)};
A$_{init}$:=A$_{init}$ ∪ {victim$_{init}$};
evaluation() ;
until (UC$_{init}$ = φ **)**
end

VII. Upon receipt of ALERT(i, j, weight$_i$) from i :
begin
for each k ∈ UC$_{init}$ **do**
if (k.id = i**) then**
k. f$_i$:= k. f$_i$ | $_{j = true}$;
// Substitutes j by true
if (evaluate (k. f$_i$) = true) **then**
UC$_{init}$:= UC$_{init}$ - {k};
A$_{init}$: = A$_{init}$ ∪ {k.id};
evaluation();
weight$_{init}$:= weight$_{init}$ + weight$_i$;
if (weight$_{init}$ =1∧ UC$_{init}$!= φ**) then**
resolution(); // Declare a Deadlock
end

3.3 Example execution

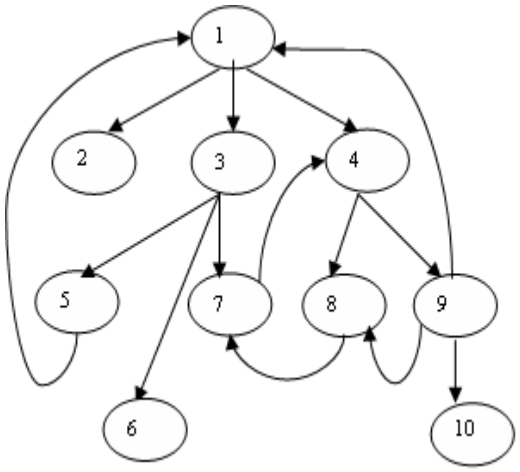

Figure 1: The Wait-For Graph

We illustrate the idea behind our algorithm with the help of an example. Figure 1 shows a distributed WFG that spans 10 nodes labelled from 1 to 10. All the nodes except 2, 6 and 10 are blocked initially. The unblocking conditions of these nodes are as follows: $F_1=(2\wedge3)\vee4$, $F_3=(5\wedge6)\vee7$, $F_4=8\wedge9$, $F_5=1$, $F_7=4$, $F_8=7$ and $F_9=(8\wedge10)\vee1$)

Let us consider node 1 initiates the algorithm and the messages are propagated in such a manner to induce a Breath First Search(BFS) Spanning Tree of the WFG. Figure 2 shows the Directed Spanning Tree, where tree

and nontree edges are indicated by solid and dashed lines respectively.

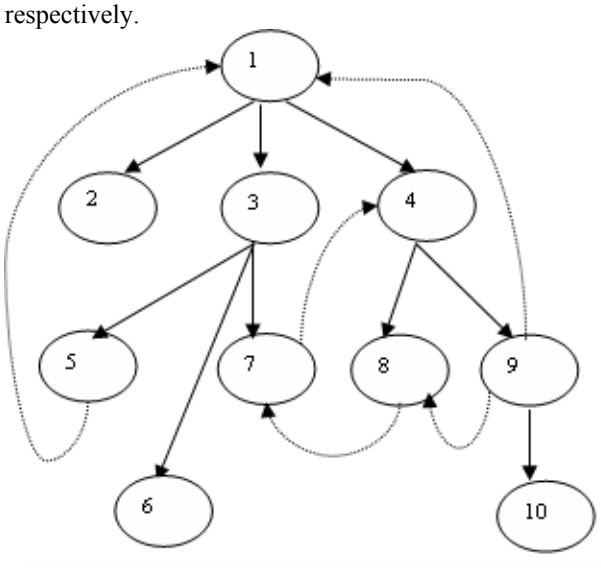

Figure 2: The Distributed Spanning Tree

1. When node 1 initiates the algorithm, it sends CALL(1,1,1/3) to nodes 2,3 and 4 respectively.
2. When node 2 receives the CALL from 1, it sends REPORT(2,ϕ,1) and WEIGHT(2,1/3) to 1.
3. When node 3 receives the CALL from 1, it sends REPORT(3,(5∧6)∨7,2) to 1 and CALL (1,3,1/9) to nodes 5,6 and 7 respectively.
4. When node 4 receives the CALL from 1, it sends REPORT(4,8∧9,2) to 1 and CALL(1.4,1/6) to nodes 8 and 9 respectively.
5. When node 5 receives the CALL from 3, it sends REPORT(5,1,1) and CALL (1,5, 1/9) to 1
6. When node 6 receives the CALL from 3, it sends REPORT(6,ϕ,1) and WEIGHT(6,1/9) to 1.
7. When node 7 receives the CALL from 1, it sends REPORT(7, 4,2) to 1 and CALL(1,7,1/9) to 4.
8. When node 8 receives the CALL from 4, it sends REPORT(8, 7,2) to 1 and CALL(1,8,1/6) to 7.
9. When node 9 receives the CALL from 4, it sends REPORT(9, (8∧10)∨1,1) to 1 and CALL(1,9,1/18) to nodes 1,8 an 10 respectively.
10. When node 1 receives the CALL from 5 through a back edge, it updates $weight_{init}$
11. When node 4 receives the CALL from 7, it sends WEIGHT (4, 1/9) to 1.
12. When node 7 receives the CALL from 8, it sends WEIGHT (7, 1/6) to 1.
13. When the initiator 1 receives the CALL from 9 through a back edge, it updates $weight_{init}$.
14. When node 8 receives the CALL from 9, it sends WEIGHT (8, 1/18) to 1.
15. When node 10 receives the CALL from 9, it sends REPORT(10,ϕ,1) and WEIGHT(10,1/18) to 1.

Figure 3 shows the flow of control messages across the WFG.

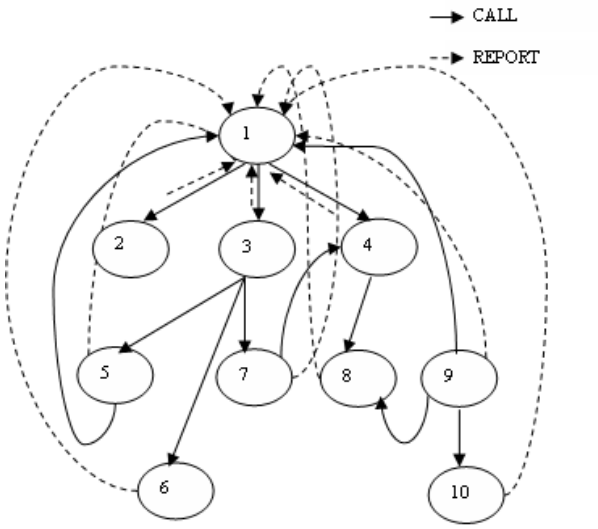

Figure 3: The Message Flow

Whenever the initiator receives the REPORT from nodes 2, 6 and 10, it simplifies the unblocking functions in the set UC_{init}. Finally, it declares the nodes 1,3,4,5,7,8 and 9 as deadlocked nodes.

3.4 Properties of the Algorithm

In this section, we prove the correctness of our algorithm by using several observations (observations 1 - 9) and lemmas (Lemmas 1-4) about the properties of the algorithm.

Observation 1: When the initiator diffuses the CALL messages, it is eventually received by all nodes in its reachable set.

Observation 2: The diffusion of CALL message induces a distributed spanning tree of the WFG.

Observation 3: Whenever a node receives the first CALL message, it propagates the message to each one of its successor.

Lemma 1: If node 'i' receives the CALL message, the unblocking function F_i is sent to the initiator.

Proof: From observation 1, each node that is reachable from the initiator receives the CALL message. Upon receiving the first CALL message, node 'i' sends its unblocking function F_i to the initiator through REPORT message after the execution of Step II.1. Thus, the lemma is proved.

Observation 4: Once a num_pred_i has been recorded in node 'i', it does not change during the execution of the algorithm.

Observation 5: Whenever a node 'i' receives the CALL message through any one of its incoming edge, it decrements num_pred_i by one.

Lemma 2: If a node 'i' sends a WEIGHT message to the initiator then it must have received CALL messages from all its predecessors.

Proof: By observation 5, upon receiving the CALL message from each node $j \in in_i$, node 'i' decreases num_pred_i by one. At the time node 'i' receives the CALL message, if num_pred_i is decremented to 0 then its weight is sent to the initiator through a WEIGHT message by Step II.2.2. Hence, the lemma holds.

Observation 6: Whenever an initiator receives the REPORT message from an active node, it evaluates the unblocking functions in the set UC_{init}.

Definition 3: The initiator reduces a node 'i' iff it has sufficient active nodes in the set A_{init} to simplify F_i as true.

Observation 7: Node 'i' can belong to the set A_{init} only if any one of the following holds.

The initiator receives a REPORT message from node 'i' that contains F_i as true

At the time the initiator evaluates F_i as true during reduction, node 'i' is added to the set A_{init}.

Definition 5: If the initiator is reduced during the evaluation, the algorithm stops the execution.

Observation 8: The weight in a CALL and WEIGHT message is always in transit until they reach the initiator and added to $weight_{init}$.

Definition 3: The algorithm is said to be terminated when $weight_{init} = 1$

Observation 9: When the algorithm terminates the execution, all nodes that is reachable from the initiator either in the set UC_{init} or A_{init}.

Lemma 3: If a deadlock exists in the system, the algorithm will detect it in finite time.

Proof : Assume that a deadlock D exists in the system. The initiator declares a deadlock only if a set $UC_{init} \neq \phi$ after the execution of Step V. Thus, it is sufficient to prove that the initiator has all information about the nodes and their associated edges in D. Let us consider node 'i' in D. It implies that node 'i' has sent F_i to the initiator through a REPORT message by lemma 1. Since node 'i' is blocked forever, the initiator evaluates F_i as false during the execution of Step V at the end of termination. Thus, all nodes in D exist in the set UC_{init}. Let us now assume a edge e = (i,j) and $e \in D$. By lemma 1, node 'i' sends this information to the initiator only after sending a CALL message to node 'j'. Before sending the CALL message, node 'i' must send a REQUEST message to node 'j'. If node 'i' has received a REPLY message from node 'j', an edge e has not been included in F_i. Since both nodes 'i' and 'j' are in D, edge e can not reduced during the execution of Step V in the algorithm. Therefore, UC_{init} contains a deadlock D after the algorithm has terminated. Consequently, the initiator sends an ABORT message to a victim until to resolve a deadlock by Step VI. Thus, the lemma holds.

Lemma 4: If a deadlock is declared, the deadlock exists in the system

Proof: The proposed algorithm reports a deadlock only when $UC_{init}=\phi$. Assume a contrary that the algorithm does not detect a deadlock D. So, it is sufficient to prove that $UC_{init} = \phi$ after the execution of Step V. This reflects the fact that a deadlock D exists in the system and the nodes of D in the set UC_{init} are reduced during the execution of Step V. Let node 'i' be one of such nodes that unblocks first in the set UC_{init}. According to the definition of deadlock, node 'i' is removed from the set UC_{init} only if the unblocking function F_i is simplified as true. It can be possible only when node 'i' had received at least one REPLY from its successors in D at some time T_i. By observation 7, it should happen only before it sends F_i to the initiator. Let node 'j' be one such successor in D that unblocks node 'i'. And node 'j' sends a REPLY to node 'i' only if it has received the REPLY from some of its successors in D before T_i. That is in contradiction with the assumption that node 'i' is the first node that unblocks in the set UC_{init}. Thus is proved.

Theorem 1: The initiator of the algorithm terminates the execution in finite time.

Proof: By step I, the initiator distributes the weight of one to all nodes that are reachable from it through CALL messages. The messages are neither lost nor duplicated according to our network assumptions. From observation 5, for each node 'i' that is reachable from the initiator sends a WEIGHT message that carries the weight value to the initiator after the execution of step II.4. The initiator executes the Step II.2.1 or IV upon receiving the CALL or WEIGHT messages and stops the execution once its weight becomes one. Since the messages transmission takes finite time, the initiator terminates the execution in finite time.

Theorem 2: The algorithm records a consistent snapshot at the initiator.

Proof: Let S be the last snapshot computed by the algorithm, and it contains a edge (p,q). This implies that this dependency relation was included in F_p which has sent by p to the initiator through a REPORT message. Before sending a REPORT message, node 'p' sends a CALL message to all its successors, including node 'q' during the execution of Step I or II. This is so because node 'p' had sent a REQUEST message to node 'q' and $p \in in_q$. This reflects the fact that the edge from p to q indeed exists in the WFG at the time of execution. Hence, the theorem holds.

Theorem 3: The algorithm detects a deadlock if and only if it exists in the system

Proof: Follows from Lemmas 3 and 4.

3.5 Concurrent executions

Since several nodes may block simultaneously, each one of them invokes the deadlock detection algorithm independently. If this happens, a node can be involved in the execution of more than one instance and several initiators may report the same deadlock. In such situations, each instance may select different victims even though a single victim is sufficient to resolve a deadlock. Nevertheless, few instances of the algorithm might be engaged in false deadlock resolution. The various issues associated with the concurrent execution of the algorithm are addressed in [8,11,12,16]. Since the method in [8] needs more messages and prone to useless aborts, we follow the priority based technique in [11,12,16] to handle concurrent executions. According to the method, the algorithm assigns a unique priority to each instance based on its identifier, which comprises the initiator's identifier and the block time / sequence numbers. Since the control messages of every instance carries this label, each instance can be distinguished from others.. When a node involves in the execution of multiple instances, it will support the execution of only high priority instance and suspends the execution of low priority instances.

3.6 Deadlock resolution

The initiator selects a victim that unblocks as many as deadlocked nodes in the set UC_{init} to resolve a deadlock. Then, it sends an ABORT message to the victim directly. It includes the victim into A_{init} and removes the corresponding tuple from the set UC_{init}. It then evaluates the unblocking functions of all nodes in the set UC_{init} and removes the nodes whose unblocking function is simplified as true. If a victim is insufficient to make UC_{init} as empty, it selects another victim. This process continues until UC_{init} is empty. Upon receiving the ABORT message, a node aborts its execution and releases all resources it had acquired earlier. An aborted process restarts its execution as in [11,13]. Thus the proposed algorithm simplifies the deadlock resolution by minimizing the messages and the nodes to be aborted.

4 Performance Analysis

We discuss the performance of the proposed algorithm with respect to time, message and data traffic complexities. The message complexity is the total number of messages exchanged by the algorithm. The time complexity of the algorithm is the time required by the initiator to detect a deadlock. The data traffic complexity defines the total length of data transmitted by the algorithm. The measurements are based on the assumption that the message transmission between any two nodes takes one time unit. We assume that n is the number of nodes, e is the number of edges and d is the diameter of the WFG.

Theorem 4: The algorithm terminates the execution in d+2 time units.

Proof: Whenever a node initiates the algorithm, it sends CALL message to its successors which in turn propagates the message to its own successors. Therefore, the CALL message must travel to the farthest node reachable from the initiator. Let $d_{max} \leq d$ be the maximum diameter of the WFG. Then, the latest time the leaf node of spanning tree receives the CALL message is $d_{max}+1$. Since the leaf node sends a WEIGHT message to the initiator directly, the algorithm will receive all replies at most $d_{max}+2$ time units. Thus the time complexity of the proposed algorithm is d+2 in worst-case.

Theorem 5: The algorithm detects a deadlock using e+2n messages.

Proof: To compute the message complexity, we consider separately each message type.

CALL messages are sent once over any edge of the WFG. Thus, at most e messages are sent totally.

REPORT messages are sent to the imitator over a communication channel directly. Since there is no more than 'n' node, the total number of REPORT messages is bounded by n.

WEIGHT messages are sent to the initiator once by the leaf nodes of spanning tree and thus no more than n-1 of such messages can be sent.

From above, we can conclude that the message complexity at worst case is O(e+2n) messages.

Let us consider the message length of proposed algorithm. Since CALL and WEIGHT messages are fixed sized, we now analyse the length of REPORT message. A REPORT message delivers the unblocking function of a node to the initiator. In the generalized model, the unblocking function of a node 'i' is a AND-OR expression that involves $|out_i|$ node identifiers. In the best case, the unblocking condition can be true and F_i is ϕ. In the worst case, F_i comprises the set of $|out_i|$ node identifiers.

For computational complexity at the initiator, we need to determine computational complexity of two procedures namely evaluation and resolution. The evaluation procedure is executed whenever an initiator receives the REPORT message from an active node. In the worst case, when all nodes are deadlocked, the unblocking functions of all n nodes are in the set UC_{init} and $A_{init}=\phi$. At the time, the algorithm declares the deadlock without evaluating the unblocking conditions and invokes the procedure resolution. In the procedure resolution, a victim is selected, inserted into the set A_{init} and removed from the set UC_{init}. Therefore, the number of processes in the set UC_{init} is reduced at least by one at each execution of resolution. Hence, in the worst case the computational complexity at the initiator is O(n) steps. In contrast, the algorithm in [16] requires $O(n^2)$ steps and the algorithm in [12] needs $O(t^2)$ steps, where t is the number of nodes in the induced spanning tree by those algorithms. However, in the best case ($UC_{init}=\phi$), the local complexity of this algorithm is O(1)

Algorithms	Delay	Number Of Messages	Message Size	Resolution
Barcha et al [4]	4*d*	4*e*	O(1)	no Scheme
Wang et.al [5]	3*d*+1	6*e*	O(1)	no Scheme
Kshemkalyani et.al [10]	2*d*	4*e*-2*n*+2*l*	O(1)	e messages
Kshemkalyani et.al [12]	2*d*	2*e*	O(*e*)	1 message
Brzezinski et.al [7]	4*n*	½ n^2	O(*n*)	no Scheme
Chen et .al [8]	2*d*	2*n*	O(*n*)	3n messages
Soojung Lee [16]	*d*+2	<2*e*	O(*d*)	1 message
Our algorithm	*d*+2	*e*+2*n*	O(*n*)	1 message

Table 1: Performance Comparison

Table 1 compares performance of different generalized deadlock detection algorithms. The message length of O(n) indicates that it consists of all node identifiers in the algorithms [7,8]. And the message used in the algorithms [12,15,16] and the proposed algorithm carries the unblocking functions to the initiator. However, the message length of these algorithms is differed due to the following reason. In the algorithm [16] the unblocking functions of nodes are merged as well as distributed during propagation of probes outward from the initiator whereas in [17], the unblocking function of each node is merged during the propagation of replies backwards to the initiator. As a result, the number of unblocking function in a reply grows as the message goes up in the spanning tree induced by the algorithm [12]. Similarly, if a node has exactly one successor, the number of unblocking conditions in a reply message is at most n-1 in the worst case in [16]. In contrast to [12, 16], the proposed algorithm sends an unblocking function of a node to the initiator disrespect the presence of deadlock and the number of successors of nodes in the WFG. In this conjuncture, the message length of proposed algorithm is a constant.

5 Conclusion

We presented a new algorithm to detect and resolve generalized deadlocks in distributed systems. The initiator of the algorithm collects the unblocking functions of all nodes in its reachable set exactly once. Then it arbitrarily simplifies the unblocking conditions depends on the reply from an active to determine deadlock. We proved the correctness of the algorithm. It has a time complexity of d+2 time units and worst case message complexity of e+2n messages hops delay to detect a deadlock. In addition, it finds out all nodes that are in deadlock with the initiator only if the initiator is deadlocked unlike the earlier algorithms. The performance of the proposed algorithm is better or comparable with the existing algorithms in terms of time, message and data traffic complexities. Furthermore, it simplifies the deadlock resolution by minimizing the additional round of messages. The proposed algorithm is applicable to detect deadlocks in different domains of distributed systems design such as resource management in distributed operating systems, store and forward communication networks, communicating processes and replicated databases.

References

[1] E.Knapp. (1987), Deadlock Detection in Distributed Database Systems, *ACM Computing Surveys*, Vol.19, No. 4 pp.303-327.

[2] M,Singhal.(1989), Deadlock detection in distributed systems. *IEEE Computer*, Vol.22, pp. 37–48.

[3] L.Lamport. (1978), Time, Clocks, and the ordering of events in a distributed systems, *ACM Communications*, vol 21, pp. 558-565.

[4] G.Bracha, and S.Toueg. (1987), A distributed algorithm for generalized deadlock detection, *Distributed Computing*, Vol.2, pp.127–138.

[5] J.Wang, S.Huang, and N.Chen.(1990), A distributed algorithm for detecting generalized deadlocks, Tech. Rep., Dept. of Computer Science, National Tsing-Hua Univ.

[6] W.K.Ng, and C.V.Ravishankar.(1994), On-Line Detection and Resolution of Communication Deadlocks, *Proc. 27th Ann. Hawaii Int'l Conf. System Science*, pp.524-533.

[7] J.Brzezinski, J.M.Helary, M.Raynal, and M.Singhal. (1995), Deadlock Models and a General Algorithm for Distributed Deadlock Detection, *J. Parallel and Distributed Computing*, Vol.31, pp.112-125.

[8] S.Chen, Y.Deng, P.C.Attie, and W.Sun.(1996), Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs, *Proc. Int'l Conf. Distributed Computing Systems*, pp.613–619.

[9] M.Roesler, and W.A.Burkhard.(1989), Resolution of Deadlocks in Object-Oriented Distributed Systems, *IEEE Trans. Computers*, Vol. 38, No. 8, pp.1212-1224.

[10] A.D.Kshemkalyani, and M.Singhal. (1989), Efficient detection and resolution of generalized distributed deadlocks, *IEEE Transactions on Software Engineering*, Vol.20, pp. 43–54.

[11] A.D.Kshemkalyani, and M.Singhal.(1997), Distributed detection of generalized deadlocks. *Proc. 17th Int'l Conf. Distributed Computing Systems*, pp.553–560.

[12] A.D.Kshemkalyani, and M.Singhal. (1999), A One-Phase Algorithm to Detect Distributed Deadlocks in Replicated Databases, *IEEE Trans. Knowledge and Data Eng.*, vol. 11, No. 6, pp. 880-895.

[13] S. Lee. and J.L. Kim.(1995), An Efficient Distributed Deadlock Detection Algorithm, *Proc. of the 15th Int. Conference on Distributed Computing System*, pp.169–178.

[14] S.Lee, and J.L.Kim.(2001), Performance Analysis of Distributed Deadlock Detection Algorithms, *IEEE Trans. Knowledge and Data Eng.*, vol. 13, no. 4,pp. 623-636 .

[15] S.Lee.(2001), Efficient Generalized Deadlock Detection and Resolution in Distributed Systems, *Proc. 21st Int. Conference on Distributed Computing Systems*, pp. 47-54.

[16] S.Lee(2004), Fast, Centralized Detection and Resolution of Distributed Deadlocks in the Generalized Model, *IEEE Trans. on Software Engineering*, Vol. 30, No.9,pp.561-573.

[17] Nacer Farajzadeh, Mehdi Hashemzadeh, Morteza Mousakhani, Abolfazl T. Haghighat. (2005), An Efficient Generalized Deadlock Detection and Resolution Algorithm in Distributed Systems, *Proc of the Fifth Int.Conference on Computer and Information Technology*

[18] Hashemzadeh, M. Farajzadeh, N. Haghighat, A.T. (2006)., Optimal detection and resolution of distributed deadlocks in the generalized model, *Proc of th 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*

[19] Srinivasan. S., Rajan Vidya, Rajaram Ramasamy. (2009), An Optimal, Distributed Deadlock Detection and Resolution Algorithm for Generalized Model in Distributed Systems, *CCIS* Vol.40, pp. 70–80.