Distrib Parallel Databases
DOI 10.1007/s10619-011-7078-7

A decentralized deadlock detection and resolution algorithm for generalized model in distributed systems

Selvaraj Srinivasan · R. Rajaram

Abstract We propose a new distributed algorithm for detecting generalized deadlocks in distributed systems. It records the consistent snapshot of distributed Wait-For Graph (WFG) through propagating the probe messages along the edges of WFG. It then reduces the snapshot by eliminating the unblocked processes to determine the set of deadlocked processes. However, the reducibility of each blocked process is arbitrarily delayed until a node collects the replies in response to all probes, unlike the earlier algorithms. We also prove the correctness of the proposed algorithm. It has a worst-case time complexity of $2d$ time units and the message complexity of $2e$, where d is the diameter and e is the number of edges of the WFG. The significant improvement of proposed algorithm over other algorithms is that it reduces the data traffic complexity into constant by using fixed sized messages. Furthermore, it minimizes additional messages to resolve deadlocks.

Keywords Distributed systems · Generalized deadlocks · Wait-For graph · Deadlock detection · Deadlock resolution

1 Introduction

Deadlock is an important resource management problem in distributed systems, since it reduces the throughput by minimizing the available resources. In general, deadlock is defined as a system state in which every process in a set is waiting indefinitely for other processes in the same set. The interdependency among the distributed processes is commonly represented by a directed graph known as Wait-For Graph (WFG) [7].

S. Srinivasan (✉) · R. Rajaram
Department of Information Technology, Thiagarajar College of Engineering, Madurai, India
e-mail: ssnit@tce.edu

R. Rajaram
e-mail: rrajaram@tce.edu

Published online: 19 January 2011

In the WFG, each node represents a process and an arc represents dependency relation between the processes. Based on the underlying resource request model, the distributed deadlock detection algorithms are classified into single Resource model, OR model, AND model and P out-of Q model and so on [5]. In the AND model, a process requires all requested resources to proceed the execution, whereas in the OR model, a process requires only few resources among the requested resources to carry out the execution. However in the P out-of Q model, every process requires "P" resources among "Q" to precede the execution. Since AND and OR model are the special case of P out-of Q model, they are also called as the generalized model. A deadlock associated with the P out-of Q model is referred as the generalized deadlocks. The generalized deadlock occurs in many areas such as resource allocation in distributed operating systems, communicating processes and quorum consensus algorithm for distributed databases [11, 15]. The AND and OR deadlock are detected by examining the existence of cycles and knot in the global WFG respectively. But the presence of cycle or knot is insufficient to determine a deadlock in the generalized model.

Very few algorithms have been proposed to detect and resolve generalized deadlocks in the literature. Most of the existing algorithms have used diffusion computation technique [1–3], in which a special process called "initiator" sends one or more messages to its descendants. If a process receives a message, it sends a reply. When every process is idle and waiting for other processes, the initiator terminates the execution and determines a deadlock based on the replies. The generalized deadlock detection algorithms are grouped into two major categories namely centralized and distributed algorithms based on the existence of WFG. In the centralized algorithm [13, 16, 17], the initiator constructs the WFG based on the information in the replies. However, the WFG constitutes multiple sites in distributed algorithms [6, 9, 11, 12, 15]. We do not consider centralized algorithms in this paper.

1.1 Proposed work

We present a new decentralized algorithm for detecting generalized deadlock in distributed systems. The initiator of the proposed algorithm builds the Distributed Spanning Tree (DST) of WFG through propagating probe (CALL) messages along the outgoing edges of WFG in the forward phase. As the replies (REPORT) are sent backwards to the initiator in the backward phase, the algorithm determines the reducibility of a blocked node. Thus the reducibility of a blocked node is arbitrarily delayed until it receives a reply in response to all probes (CALL messages). An unblocked process initiates the reduction of distributed snapshot by eliminating all reducible nodes during the backward phase. Finally, the processes that have not been reduced in the snapshot are declared as deadlocked processes. We formally prove the correctness of proposed algorithm. It has a worst case time complexity of $2d$ time units and the message complexity of $2e$, where d is the diameter and e is the number of edges of the WFG. However, it reduces the data traffic complexity into constant by using fixed sized messages as compared to the existing distributed algorithms.

1.2 Differences between the proposed algorithm and earlier algorithms

In the proposed algorithm, the replies are not carrying the unblocking conditions and the reducibility of a node is not delayed until the termination of algorithm as in [15]. Furthermore, it does not construct WFG partially at the initiator to find out a victim. Unlike in [11], the termination of the algorithm is not based on weight distribution technique and it does not require additional rounds of messages to resolve deadlocks.

This paper is organized as follows. Section 2 describes the related work. Section 3 describes the underlying computational model and the problem definition. Section 4 presents the proposed algorithm and its correctness proof. Section 5 does the performance analysis and compares it with the earlier algorithms. Section 6 concludes.

2 Related works

The distributed algorithms 'record and reduce' the global WFG in a single or two phases. In the two phase algorithms [6, 9, 12], the WFG is recorded through propagating the probes to all processes in initiator's reachable set in the first phase. It reduces the unblocking processes in the second phase to find out a deadlock. However, these two phases are overlapped in the single phase algorithms [11, 15]. In [6], the initiator propagates the probe messages to record the WFG in the first phase and collects the replies from the unblocked processes in the second phase. The existence of deadlock is implied when the replies are insufficient to unblock the initiator. Here, the second phase is nested within the first phase. It exchanges $4e$ messages in $4d$ time units to find out whether the initiator is in deadlock or not. The algorithm in [9] uses an effective termination technique to detect the end of the first phase before initiating the second phase, unlike in [4]. However, it uses $6e$ messages within $3d + 1$ time units to detect a deadlock. The algorithm in [12] arranges the processes in the distributed snapshot into logical ring, and circulates the probes (token) among them. If the processes do not change their states in two consecutive rounds, it declares a deadlock. It uses weak termination technique as compared to the earlier algorithms. It detects a deadlock using $\frac{1}{2}n^2$ messages in $4n$ time units, where n is the number of nodes in the WFG. Unlike in [6, 9, 12], the algorithm in [11] records as well as reduces the WFG simultaneously to find out the presence of deadlock. The initiator records all the processes in its reachable set in outward sweep, and eliminates the process that grants the resources in the inward sweep. It detects the generalized deadlock through $4e - 2n + 4l$ messages within $2d$ time units. The algorithm in [15] uses lazy evaluation technique to delay the reduction of any unblocked processes until the initiator terminates the execution. It detects deadlock using $2e$ messages in $2d$ time units. In addition, the initiator retains the resource requirement of blocked processes to resolve the deadlock. Thus, [15] avoids additional e messages.

3 System model and problem definition

The system consists of n processes, where each has unique identity. The processes are communicating through a logical communication channel by message passing. There

is no shared memory in the system. The messages are delivered at the destination in the same order as sent by the sender, with arbitrary but finite delay. The messages are neither lost nor duplicated and the entire system is fault-free. The events in the system are classified into internal and external events, and they are time stamped using logical clock [1]. They are further classified into computation events and control events. The computation event triggers the computational messages such as REQUEST, REPLY, CANCEL and ACK due to the execution of applications. Whereas, the control event generates the control messages including CALL and REPORT as a result of the execution of deadlock detection algorithm.

In a generalized model, a process resource requirement is expressed as a predicate involving the requested resources using logical AND OR operators. For example, a process resource requirement $A \wedge (B \vee C)$ implies that it requires a resource from A and a resource from either B or C. In this algorithm, the generalized resource requirement of a blocked process i is represented as a function F_i. Once a blocked process i receives the replies from its descendants, it simplifies the function as follows. It substitutes *true* if a process grants a resource and *false* if it denies it. When a process i collects sufficient replies to make F_i as *true*, it immediately gets unblocked. Each process i maintain its local state using the following data structure. The initial values are given within the brackets.

t_block_i	the logical time at which i was last blocked(0)
IN_i	the set of tuples $<k, t\_block_k>$ where k is a process that is waiting for i and t_block_k is the logical time at which k has sent its request to process i (ϕ)
OUT_i	set of processes for which process i is waiting since the last t_block_i (ϕ)
F_i	the unblocking condition of process i

Hence, Domain(F_i) $\subseteq OUT_i$. Each process $j \in OUT_i$ is referred as the successor of i and each process $j \in IN_i$ is referred as the predecessor of i. We use the '*process*' and '*node*' interchangeably throughout this paper.

When a process i blocks on p_i out-of q_i requests, it sends a REQUEST message to q_i processes. If it receives p_i REPLY messages, it immediately sends $q_i - p_i$ CANCEL messages to withdraw its request and become active. Otherwise, a process is in *blocked* state. Therefore, a process state is *active* or *blocked* at any instant. An *active* process can send both communication and control messages while the *blocked* process can send either control messages or ACK. A *blocked* process could not request additional resources and unblock abnormally. These two assumptions are essential to record consistent snapshot of distributed WFG. Whenever a process i receives the REQUEST message from a *blocked* process $j \in OUT_i$, it records the request along with t_block_i in IN_i. A blocked process j then sends ACK message immediately to acknowledge the arrival of REQUEST message. Both communication and control messages are properly time stamped according to [1] in order to synchronize with corresponding messages.

The distributed snapshot comprises the local states of all processes in initiator's reachable set. It is formally defined as follows in [11].

Definition 1 A snapshot of process i is a collection of local states of all processes reachable from it. A consistent snapshot records both the sending and receiving of

REQUEST, REPLY and CANCEL message. If the receipt of the message exists in the snapshot, the corresponding sending event must be recorded.

Definition 2 A node i is reachable from j iff there exists a directed path from i to j in the WFG.

3.1 Problem statement

A generalized deadlock exists in the system iff the topology, which is given in Definition 3, exists in the WFG.

Definition 3 A generalized deadlock is a sub graph (D, K) of WFG (V, E) in which

1. (i) $\forall i \in D, evaluate(F_i) = false \wedge D \neq \phi$
2. (ii) $\forall i \in V - D, evaluate(F_i) = true$
3. (iii) $\forall i \in D, \forall j \in OUT_i$, no REPLY message is in underlying communication channel from j to i.

Where, $evaluate(F_i) = evaluate(F_i/_{\forall j \in OUTi, j \leftarrow evaluate(Fj)})$ and $evaluate(F_i) = true$ for an *active* node. Hence, each process in a set D is *blocked* permanently.

3.2 Correctness criteria

The correctness of any deadlock detection algorithm depends on the following two criteria.
Liveness: The algorithm detects the deadlock within a finite time after its formation in the underlying system.
Safety: The algorithm reports the deadlock iff it actually exists in the system.

4 The proposed algorithm

When a node i blocks on a p_i out-of q_i requests, it initiates the deadlock detection algorithm. The initiator of the algorithm records the consistent snapshot of distributed WFG by propagating the CALL messages along the outgoing edges of WFG. When the replies are propagated backwards to the initiator, the algorithm reduces the snapshot to determine a deadlock. The proposed algorithm follows the method in [10, 11, 15] to handle the concurrent executions of the algorithm. According to the method, the algorithm assigns a priority to each instance based on the initiator's identifier and the time at which it was blocked. It supports the execution of higher priority and suspends the execution of lower priority instances in the conflicting nodes. Hence, each initiator maintains its own snapshot to detect a deadlock. For simplicity, we focus only on single instance execution of our algorithm.

4.1 An overview of the algorithm

When a node i initiates the deadlock detection algorithm, it builds a distributed spanning tree of distributed WFG by propagating CALL messages along the edges in the

WFG. To describe in detail, the initiator i sends CALL message to each one of its successors. When a blocked node receives the first CALL message, it becomes the child of the sender and propagates CALL messages to its own successors and so on. Thus, an edge through which each node receives the first CALL message induces a distributed spanning tree in the WFG.

An *unblocked* node initiates the reduction of distributed snapshot by sending REPORT message to the sender of CALL message immediately. But, a *blocked* node delays to send a reply in response to CALL message until the arrival of REPORT messages from all of its successors. When a *blocked* node receives all REPORT messages, it attempts to simplify the unblocking condition to determine its *state*. However, if a node sends the CALL message to any one of its own predecessor through a back edge, it delays to simplify its unblocking condition until it receives the REPORT from other successors. In such situations, it substitutes *false* for the node ids that does not send REPORT message during the simplification. After simplifying the unblocking condition, it sends its *state* to all predecessors through REPORT messages. This process continues until the initiator determines its own *state*. If the unblocking condition of initiator is not simplified as *true* upon receiving the REPORT messages from its successors, the algorithm declares a deadlock. And the nodes that have not been reduced are declared as deadlocked.

In this algorithm, the deadlock detection as well as termination is incorporated into a single process as in [14, 15]. Whenever a blocked node receives the REPORT message in response to all CALL messages, it sends REPORT message to its predecessors. This process continues until the initiator receives the REPORT message from its own successors. The algorithm terminates once the initiator receives all REPORT messages in response to CALL messages.

4.2 An explanation of the algorithm

When node i wants to find out whether it is deadlocked, it sends a CALL(i, i) message to all its successors (out_i). The first parameter of the CALL message is id of the node that propagates the message and the second parameter is the id of the initiator. When node j receives the CALL message from node i, it performs one of the following actions.

1. If it is the first CALL message and node j is *blocked*, it sets its $father_j$ to i and sends the CALL(j, *initiator*) message to all the nodes in out_j.
2. If it has already received a CALL message (i.e., $father_j \neq$ udef), it includes the id of i in the set in_j. It also reduces m_j by one ($m_j = 0$ implies that the node j receives the CALL message from all its successors).
3. If node j is *active,* it sends REPORT(*initiator*, j, *true*, ϕ, ϕ) to node i. The first parameter of the REPORT message is the id of the initiator. The second and third parameter represents the id and the *state* of the node that sends the message respectively. The fourth parameter is the id of the node that would be a victim in case of deadlock and fifth parameter is the number of predecessors of a node *victim*. Since node j is *active*, it cannot be a victim. Therefore, the fourth and fifth parameter value is set as ϕ in the message.

4. If node j receives the CALL message through a phantom edge (i.e., i$\notin IN_j$), it sends a REPORT(i, j, *true*, ϕ, ϕ) message immediately to node i.

Whenever node i receives the REPORT message, it reduces n_i by one. Once it receives the REPORT message from all its successors (i.e., $n_i = 0$), it evaluates its unblocking condition (f_i). If f_i is simplified as *true*, it sends the REPORT message to its predecessors without changing the *victim* and $|in_{victim}|$ in the message. Otherwise, it updates the fourth and fifth parameter of the message by comparing number of its predecessors ($|in_i|$) with $|in_{victim}|$. If $|in_i| \geq |in_{victim}|$, it sets node i as victim and sends REPORT(*initiator*, i, *false*, i, $|in_i|$) to its predecessors. Else, it sends REPORT (*initiator*, i, *false*, *victim*, $|in_{victim}|$) to its predecessors.

In some cases, node i is waiting to receive the REPORT message from its own predecessor j in response to its CALL message (i.e., when $j \in in_i \wedge j \in out_i$) for determining its *state*. In such cases, node i cannot send the REPORT message to its predecessors including the node j. This problem is resolved as follows. When node i receives the REPORT message, it reduces n_i by one. In addition, it counts the number of nodes that act as both predecessor and successor (*loop*). Therefore, it attempts to simplify its unblocking condition (f_i) at the time it has received n_i-*loop* REPORT messages. It then sends REPORT message to its predecessors. This will ensure that any node that is reachable from the initiator does not wait indefinitely to determine its *state*.

After receiving the REPORT message from all its immediate successors, the initiator evaluates its unblocking condition. If the unblocking condition of the initiator is not simplified as *true*, the algorithm declares deadlock. In that case, the initiator sends ABORT message to a node *victim* directly to resolve it.

4.3 Formal specification

The formal description of the proposed algorithm is presented below. The initial values are given inside the parenthesis.

4.4 An example

We now illustrate the algorithm with the help of an example shown in Fig. 1 [17]. Let us consider the distributed WFG that spans six nodes labeled 1 to 6. Assume that node 1 initiates the deadlock detection algorithm and the messages are propagated in such a way to induce a BFS distributed spanning tree. All the nodes except 6 are blocked. The unblocking conditions of all blocked nodes are given as follows: $F_1 = 2 \wedge 3$, $F_2 = (4 \wedge 5) \vee 6$, $F_3 = 5$, $F_4 = 5 \vee 6$ and $F_5 = 3 \wedge 6$. Figure 2 shows the distributed spanning tree induced by the algorithm.

Node 1 sends CALL(1, 1) message to nodes 2 and 3 respectively. When node 2 receives the CALL from 1, it propagates the CALL(2, 1) message to 4, 5 and 6 respectively. Similarly, node 3 sends CALL(3, 1) message to 5. A node 4 sends CALL(4, 1) message to its successors 5 and 6, whereas node 5 sends CALL(5, 1) message to its own successors 3 and 6 in response to CALL message from node 2. When node 6 receives the CALL message from 2, 4 and 5, it sends REPORT(1, 6, *true*, ϕ, ϕ) in response to CALL message. When node 5 receives the REPORT message from

Data Structures of a node *i*

t_block_i:	integer $\leftarrow 0$	/* time at when i was blocked */
$initiator_i$:	integer $\leftarrow 0$	/* initiator of the current instance */
in_i:	set of tuples $< j, t\_block_j > \leftarrow \phi$	/* nodes waiting for i^*/
out_i:	set of integers $\leftarrow OUT_i$	/* nodes for which i is waiting for */
f_i:	AND-OR Expression	/* Condition for i to unblock */
$father_i$:	integer $\leftarrow 0$	/* parent of i^*/
m_i:	integer $\leftarrow 0$	/* number of predecessors of i^*/
n_i:	integer $\leftarrow \|OUT_i\|$	/* number of successors of i^*/

Messages Formats

CALL (*sender*, *initiator*)

REPORT (*initiator*, *sender*, *state*, *victim*, $|in_{victim}|$)

I. When a node *i* initiates the algorithm //Step I

$initiator_i := i$;

$father_i := i$;

$m_i := |IN_i|$;

send CALL(i, i) to each process $j \in OUT_i$;

II. On receiving CALL(j, *initiator*$_j$) by node *i*

if ($father_i = udef \wedge |OUT_i| > 0 \wedge j \in IN_i$) **then** /* Step II.1 tree edge */

$father_i := j$;

$initiator_i := initiator_j$;

$out_i := OUT_i$;

$in_i := in_i \cup \{j\}$

$f_i := F_i$;

$n_i := |OUT_i|$;

$m_i := |IN_i| - 1$;

send CALL(i, $initiator_i$) to each process $j \in out_i$;

if ($father_i = def \wedge |OUT_i| > 0 \wedge j \in IN_i$) **then** /* Step II.2 non tree edge */

$in_i := in_i \cup \{j\}$

$m_i := m_i - 1$;

if ($m_i > 0$) **then**

$loop := |IN_i \cap OUT_i|$; // Number of nodes that act as both predecessor & successors

if (($loop > 0$) $\wedge$(($n_i - loop$) $= 0$)) **then**

send REPORT($initiator_i, i, false, i, |in_i|$) to *j;*

if ($|OUT_i| = 0 \wedge j \in IN_i$) **then**

/* Step II.3 When unblocked node i receives the CALL */

send REPORT($initiator_j, i, true, \phi, \phi$) to j;

if ($j \notin IN_i$) **then** /* Case II.4 phantom edge */

send REPORT($initiator_j, i, true, \phi, \phi$) to j;

III. On receiving REPORT ($initiator_j$, j, $state$, $victim$, in_{victim}) by node i
if ($state = true$) **then**
  /*Step III.1 When node i receives the REPORT from an active node j^*/
  $out_i := out_i - \{j\}$;
endif
$n_i := n_i - 1$;
if ($n_i = 0$) **then**
  $evaluate(initiator_i, i, victim, in_{victim})$;
else
  $loop := |IN_i \cap OUT_i|$;
  if $((m_i = 0) \wedge ((n_i - loop) = 0))$ **then** /* Step III.2 Resolving Loops */
    $evaluate(initiator_i, i, victim, in_{victim})$;
  endif
endif
IV. procedure $evaluate(initiator_i, i, victim, in_{victim})$
begin
  if $((i \neq initiator_i) \wedge (evaluate(f_i) = true))$ **then** /*Step IV.1 node i is reducible */
    $state := true$;
    send REPORT$(initiator_i, i, true, victim, in_{victim})$ to $j \in in_i$;
  else /* Step IV.2 node i is not reducible */
  if $(|in_i| \geq |in_{victim}|)$ **then** /*Selection of a victim */
    send REPORT$(initiator_i, false, i, |in_i|)$ to $j \in in_i$;
    else
    send REPORT$(initiator_i, false, victim, in_{victim})$ to $j \in in_i$;
    endif
endif
if $((i = initiator_i) \wedge (evaluate(f_i) = true))$ **then**
    No deadlock; *exit*; /* Step IV.3 Check the state of initiator */
else
  Declare Deadlock; send ABORT$(initiator_i)$ to *victim*;
endif
end procedure
V. On receiving the ABORT message by node i
send the CANCEL message to withdraw its REQUEST message to $j \in out_i$;
send the REPLY to $j \in in_i$;
abort i; // the process 'i' is terminated

6, it sends REPORT$(1, 5, false, 5, 2)$ message to 2, 3 and 4 respectively. A node 4 sends REPORT$(1, 4, true, 5, 2)$ to 2. Node 3 sends REPORT$(1, 3, false, 3, 2)$ to 1. Upon receiving the REPORT messages from 4, 5 and 6, node 2 sends REPORT$(1, 2, true, 5, 2)$message to 1. Since the REPORT message from 2 and 3 are

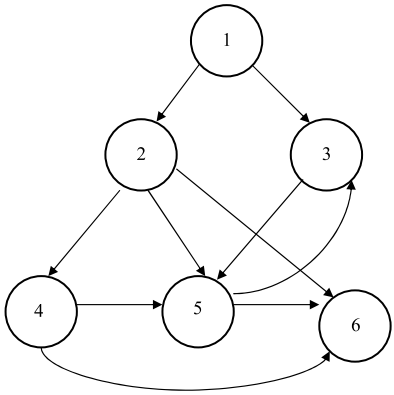


Fig. 1 The Wait-For graph

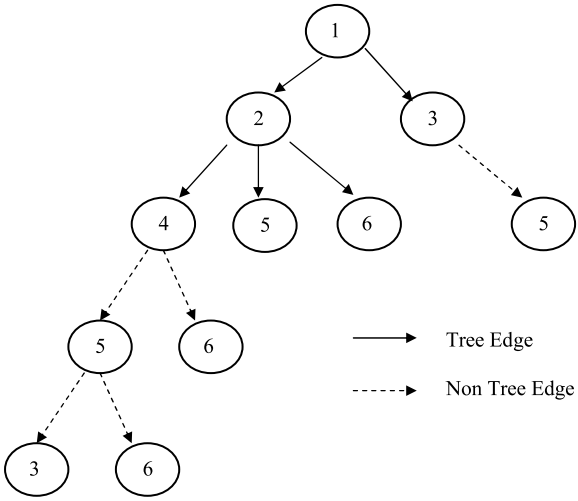


Fig. 2 The distributed spanning tree

insufficient to simplify the unblocking condition of node 1 into *true*, the algorithm declares a deadlock. It then selects node 3 as victim to resolve a deadlock.

4.5 Deadlock resolution

The *victim* is selected based on the number of predecessors of deadlocked nodes in the WFG. To be more specific, a process that unblocks maximum number of processes in the underlying is chosen as a *victim*. It is achieved in the following manner. The number of predecessors of each node is sent to its predecessors through the REPORT message. Upon receiving the REPORT message, the value in the message is compared with its own number of predecessors. It then sends the identity of a node that has maximum number of predecessors through REPORT messages. This process continues until the initiator collects the replies from all its successors. When the initiator detects a deadlock, it sends abort signal to the victim directly [8].

4.6 Correctness proofs

Since we do not consider the execution of multiple instance of the algorithm, the proofs to prove the correctness of single instance execution is presented using the following theorems.

Theorem 1 *The algorithm records the complete and consistent snapshot of distributed WFG.*

Proof The completeness property ensures that the initiator of the algorithm sends CALL messages to all the nodes in its reachable set. To prove, Let us consider a node i, which initiates the algorithm. It diffuses the CALL messages to each one of its successor $j \in OUT_i$ according to the Step I. Since $<i, t\_block_i> \in in_j$, a node j receives the CALL message from process i. If a node j is blocked, it forwards the CALL message to its own successors by Step II.1. If an unblocked node j receives the CALL message, it sends the REPORT message to i by Step II.3. Since the messages are never lost according to our network assumption, all the nodes reachable from the initiator i receives the CALL message. Thus, the snapshot is complete.

We now show that the algorithm records consistent snapshot as follows. Let us consider the contrary that an edge (i, j) does not exist in the WFG. Such an edge exists in the snapshot iff node j receives the CALL message from process i. It is possible only if $j \in OUT_i$ and $i \in in_j$. If an unblocked node j receives the CALL message from node i, it sends REPORT message to node i by Step II.3.As a result, the algorithm reduces the edge which is subsequently removed from the snapshot. However, if a blocked node j receives the CALL message, the edge (i, j) is included in the snapshot by Step II.2. Therefore, an edge(i, j) exist in the snapshot iff it exists in the WFG. Thus, the contradiction is disproved. □

Theorem 2 *The algorithm terminates within a finite time.*

Proof By Step I, the initiator i sends CALL message to each one of its successor. If a blocked node j receives the CALL message, it forwards CALL message to its own successors by Step II.1. However, if an unblocked node receives the CALL message, it sends a reply to the sender of message immediately by Step II.3. Whenever a node receives the REPORT message, it reduces the required number of replies by one according to the Step III. Once a blocked node collects the replies in response to all CALL messages, it sends REPORT message to its predecessors by Step IV. This process continues until the initiator receives the REPORT message from all its successors. The initiator of the algorithm terminates the execution once it receives the REPORT messages in response to all CALL messages. Thus, the theorem holds. □

Theorem 3 *If a deadlock exists in the system, the algorithm detects it within finite time.*

Proof Assume that a deadlock D exists in the system. Let us consider a contrary that the algorithm does not detect a deadlock D in the underlying system. Hence,

there is a node $i \in D$, where $evaluate(f_i) = true$ exists in the snapshot. Since a node i'is a member of deadlock D, it will be blocked forever according to the definition of deadlock. However, if no such deadlock D exists in the system according to our assumption, f_i is evaluated as *true* during the simplification. Therefore, $evaluate(f_i) = true$ and $i \notin D$, which contradicts our assumption. If $evaluate(f_i) = true$, then f_i consists of only those nodes that are not deadlocked in the system. Let us consider a node $j \in Domain(f_i)$ and whose state determines the state of node i. Since $evaluate(f_i) = true, evaluate(f_j)$ is also *true*. Similar to node i, there is at least one node $j' \in Domain(f_j)$ and whose state determines the state of node j'. By induction, there must exist a node $n + 1 \in D$ such that $n \in Domain(f_{n+1})$ and $evaluate(f_n) = true$. Since active nodes have *true* unblocking conditions by Step IV, the algorithm does not $evaluate(f_i)$ as *true* during the simplification. So the algorithm does not report $i \in D$ which contradicts our assumption. Hence, the theorem is proved. □

Theorem 4 *The algorithm does not report any false deadlock.*

Proof Let us prove this theorem using a contrary that the algorithm reports a deadlock that does not exist in the system. It implies that there are some edges recorded in the snapshot that have not existed in the system. Let us consider an edge (i, j) is one among them. The edge exists in the snapshot iff node i sends CALL message to node j by Step I or II.1. By Step I, the initiator sends CALL message to its successor j. If it is executed the later step, a blocked node i sends CALL message to each one of its successor including node j. If a node j sends REPLY to node i, this edge disappears from the WFG. Therefore, it does not exist in the snapshot. Let us consider that (i, j) is a tree-edge of distributed spanning tree induced by the algorithm. Upon receiving the CALL message from node i, node j executes either step II.1 or II.3. If node j executes Step II.3, it sends REPORT to node i which subsequently reduces the edge (i, j). Hence, it will not exist in the snapshot. If node j is blocked in the WFG, it executes the Step II.1 and eventually sends CALL message to its successor. Since an edge does not in the WFG, node i must be reduced during the simplification. Therefore, it should not exist in the snapshot. Let us now consider (i, j) is a non-tree edge. Since the edge disappears from the WFG, node j must send REPLY to node i at some time, say t. If node j receives the CALL message before t, it sends REPORT message to node i once it determine its state by Step III. Since node j is not deadlocked, the edge (i, j) is removed from the snapshot once it sends REPORT message to node i by Step III. If node j receives the CALL from node i before t, it sends REPORT message to node i by Step II.4. Hence it does not exist in the WFG. Hence the algorithm records an edge iff it exists in the WFG. Thus the contradiction is disproved, and the theorem is proved. □

5 Performance analysis

We compare the performance of the proposed algorithm with the existing algorithms in terms of time, message, data traffic and space complexity. The time and message

Table 1 Performance Comparison of Distributed Deadlock Detection algorithm in the worst Case Here, S represents the total number of processes in the System, n represents the number of nodes, e represents the number of edges and d represents the diameter of WFG (V, E)

Algorithm	Delay	Messages	Message length
Bracha and Toueg [6]	$4d$	$4e$	$O(l)$
Wang et al. [9]	$3d+1$	$6e$	$O(l)$
Brzezinski [12]	S^2	S^2	$O(n)$
Kshemkalyani et al. [11]	$2d+2$	$4e-2n+2l$	$O(l)$
Kshemkalyani et al. [15]	$2d$	$2e$	$O(e)$
Proposed Algorithm	$2d$	$2e$	$O(l)$

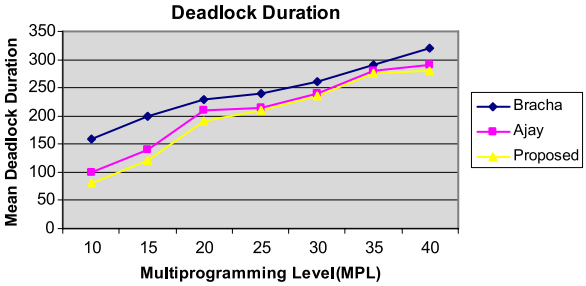
complexity are measured based on the assumption that the transmission of logical messages between any two processes takes one time unit. Let n be the number of processes, e be the number of edges and d be the diameter of the WFG. In this algorithm, the CALL messages are sent along the outgoing edges of WFG in the forward phase and the REPORT message are sent backwards to the initiator in the backward phase. Hence the message complexity of this algorithm is $2e$. Since the initiator determines a deadlock once it has received the REPORT messages from all its successors, the worst-case time complexity of our algorithm is $2d$. The space complexity in the worst-case will not exceed $O(n^2)$. However, it minimizes the data traffic complexity into a constant by using fixed sized messages. The Table 1 compares the performance of proposed algorithms with other algorithms.

We have compared the performance of proposed algorithm with that of Bracha's algorithm [6] and Ajay's algorithm [15]. In the simulation, programs are event driven and written in JAVA. As in [17], the events are classified into process arrival/departure, message receipts for communication and message receipt of computational messages. The simulator maintains all data structures including Wait-For Graph to support the execution of each algorithm. The system has 'L' resources and each one of them is associated with an exclusive lock. The resources are evenly distributed over sites. Upon entering into the system, a process spends T_{pre} time units before making process size (PS) resource requests. It requests a resource either local or remote based on the probability P_l. If a process needs a remote resource, it takes T_m time units to send a REQUEST message. The Lock Manager determines whether a lock can be granted. If the resource is idle, lock Manager allocates the resource to the process; otherwise, if the resource is already held by another process, the resulting process sends a REQUEST message, and includes the process identifier that holds a lock into the successor list. Once a process acquires PS locks, it becomes *active* and continues its execution for T_{exec} time for each acquired resources. At the end of the execution, it releases all the acquired resources. Termination of a process brings a new process in the site immediately. Hence, the system has fixed number of processes at any instant. At the same time, a process blocks continuously until all the requested locks are granted. When a lock is released, the lock manager selects one of its successors as a new holder of the lock. It then sends a UPDATE message to inform all other processes that are waiting for a resource. Upon receiving a UPDATE message, a process modifies its successor list by adding the current holder instead

Table 2 System parameters

Parameter	Description	Mean value
L	Total Number of Locks	300
PS	Process Size	1–10
T_{pre}	Execution time of a process before making PS resource requests	60
T_{exec}	Execution time of a process to utilize each acquired resource	30
P_l	Probability of a process to make request for a resource residing at local site	0.1
T_m	Transmission time of a message	20
T_{dlmsg}	Time require to execute a routine corresponding to a deadlock detection message	1.5

Fig. 3 Deadlock duration



of previous holder. A deadlock detection algorithm is executed at a site upon receiving a computational message from other sites or upon initiation of the algorithm. We assume that, the system takes T_{dlmsg} time for a process to execute a deadlock detection module regardless of the message type and the algorithm. We have assessed the performance of all algorithms under the same initial conditions. Table 2 presents the system parameter values used by the simulation. To increase the degree of lock conflicts, a relatively small number of resources in comparison with the process size have been chosen.

We have used the following metrics for quantitative comparison: Deadlock Duration which is the time taken by the algorithm to detect a deadlock after it happens. Message traffic is the number of messages needed to detect deadlocks. Message length is the size of deadlock detection messages. Deadlock Resolution time is the time when a deadlock occurs until it is resolved by the algorithm. Experiments have been carried out in both the light and heavily loaded environments by varying the multiprogramming level ranging from 10 to 40. Each simulation result is the mean value obtained after running the programs 20 times for the same set of input parameter. Each simulation was run for 100000 time units.

Figure 3 shows the deadlock duration plotted as a function of the multiprogramming level of the system. As shown in the figure, mean deadlock detection duration resulting from proposed algorithm is less than that from Bracha's algorithm [6]. It is observed that deadlock duration of Ajay's algorithm [15] and our algorithm is almost same for higher MPL values, which is consistent with complexity comparison presented in Table 1. It is also observed that the deadlock detection duration increases

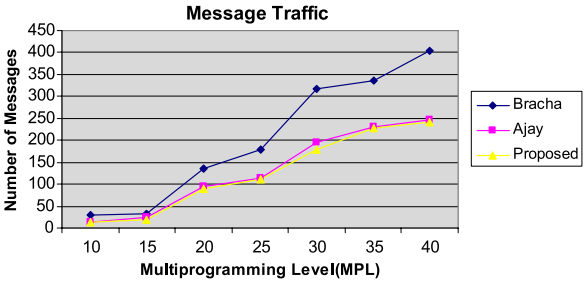


Fig. 4 Message traffic

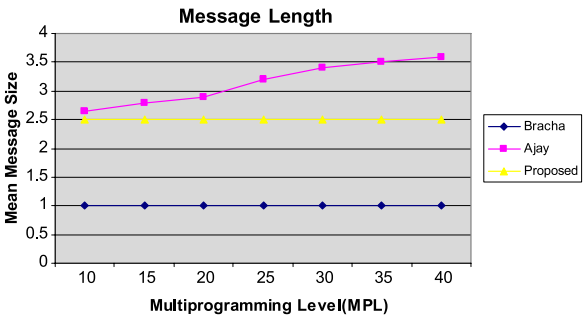


Fig. 5 Message length

with MPL until the number of processes reaches 30, and then tapers to flat. The reason behind this is due to the increase of simply blocked nodes with MPL.

Figure 4 shows the mean number of deadlock detection messages generated per algorithm execution with varying multiprogramming levels. As shown in figure, Bracha's algorithm [6] passes 1.5 times more messages than proposed algorithm for higher MPL values. It is observed that proposed algorithm and [15] need almost same number of messages to detect deadlocks according to the congruence with the theoretical expectation.

Figure 5 shows the mean length of deadlock detection messages in terms of number of node identifiers for each algorithm. It is observed that, as the MPL is increased in the system, the message length of Ajay's algorithm [15] is also increased. However, the message length of proposed algorithm and [6] is a constant. If the system is in deadlock, the algorithm in [6] aborts the initiator which may not resolve the detected deadlock. On the contrary, the algorithm in [15] selects a victim by invoking additional procedure like centralized algorithms. Since the initiator of proposed algorithm identifies an appropriate victim without invoking any additional procedure, the deadlock resolution time is very less in proposed algorithm as compared to [15]. It is observed that a deadlocked process having highest predecessor is aborted and it is more likely that abortion of single process might resolve a deadlock. It's another key contribution of proposed algorithm.

6 Conclusion

We have presented a new distributed deadlock detection and resolution algorithm in generalized model and proved its correctness. In this proposed algorithm, the probes

are propagated along the edges of WFG in the forward phase and the replies are sent backwards to the initiator in the backward phase. The reducibility of a blocked node is decided once it has received the REPORT messages from all its descendants unlike the earlier algorithms. If the initiator is not reduced at the end of termination, the algorithm declares a deadlock. It is shown that the message complexity of $2e$ and time complexity of $2d$ is equal or better than the existing algorithms. The notable improvement of this algorithm is that it significantly reduces the message length without using any explicit techniques. The data traffic complexity of proposed algorithm is optimum as compared to any decentralized algorithms. Since the initiator identifies the appropriate victim during the propagation of replies, it significantly minimizes the message overhead associated with deadlock resolution. Our simulation result reveals that the proposed algorithm performs better than the existing decentralized algorithms in terms of message length and deadlock resolution.

References

1. Lamport, L.: Time, clocks, and the ordering of events in a distributed systems. Commun. ACM **21**, 558–565 (1978)
2. Dijkstra, E.W., Scholten, C.S.: Termination detection for diffusing computations. Inf. Process. Lett. **11**(1), 104 (1980)
3. Chandy, K.M., Misra, J., Hass, L.M.: Distributed deadlock detection. ACM Trans. Comput. Surv. **1**(2) (1983)
4. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Trans. Comput. Syst. **3**(1), 63–75 (1985)
5. Knapp, E.: Deadlock detection in distributed database system. ACM Comput. Surv. **19**(4), 303–327 (1987)
6. Bracha, G., Toueg, S.: A distributed algorithm for generalized deadlock detection. Distrib. Comput. **2**, 127–138 (1987)
7. Singhal, M.: Deadlock detection in distributed systems. IEEE Comput. **22**, 37–48 (1989)
8. Roesler, M., Burhard, W.A.: Resolution of deadlocks in object-oriented distributed systems. IEEE Trans. Comput. **38**(8), 1212–1224 (1989)
9. Wang, J., Huang, S., Chen, N.: A distributed algorithm for detecting generalized deadlocks, Tech. Rep., Dept. of Computer Science, National Tsing-Hua Univ. (1990)
10. Kshemkalyani, A.D.: Characterization and correctness of distributed deadlock detection and resolution. Ph.D. dissertation, Ohio State Univ. (1991)
11. Kshemkalyani, A.D., Singhal, M.: Efficient detection and resolution of generalized distributed deadlocks. IEEE Trans. Softw. Eng. **20**(1), 43–54 (1994)
12. Brzezinski, J., Helary, J.M., Raynal, M., Singhal, M.: Deadlock models and a general algorithm for distributed deadlock detection. J. Parallel Distrib. Comput. **31**(2), 112–125 (1995)
13. Chen, S., Deng, Y., Attie, P., Sun, W.: Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs. In: Int'l. Conf. Distributed Computing Systems, pp. 613–619 (1996)
14. Boukerche, A., Tropper, C.: A distributed graph algorithm for the detection of local cycles and knots. IEEE Trans. Parallel Distrib. Syst. **9**(8), 748–757 (1998)
15. Kshemkalyani, A.D., Singhal, M.: A one-phase algorithm to detect distributed deadlocks in replicated databases. IEEE Trans. Knowl. Data Eng. **11**(6), 880–895 (1999)
16. Lee, S., Kim, J.L.: Performance analysis of distributed deadlock detection algorithms. IEEE Trans. Knowl. Data Eng. **13**(4), 623–236 (2001)
17. Lee, S.: Fast, centralized detection and resolution of distributed deadlocks in the generalized model. IEEE Trans. Softw. Eng. **30**(9), 561–573 (2004)