Optimal Deadlock Detection in Distributed Systems Based on Locally Constructed Wait-for Graphs [*]

Shigang Chen [†], Yi Deng, Paul C. Attie and Wei Sun [‡]

Abstract

We present a new algorithm for detecting generalized deadlocks in distributed systems. Our algorithm incrementally constructs and reduces a wait-for graph (WFG) at an initiator process. This WFG is then searched for deadlock. The proposed algorithm has two primary advantages: First, it avoids sending messages along the edges of the global wait-for graph (WFG), thereby achieving a worst-case message complexity of $2n$, where n is the number of processes in the WFG. Since information must be obtained from every process reachable from the initiator, this is optimal to within a constant factor. All the existing algorithms for the same problem construct a distributed snapshot of the WFG. As this involves sending messages along the edges of the WFG, the best available message complexity among these algorithms is $4e - 2n + 2l$, which is $O(n^2)$ in the worst case, where e and l are the number of edges and leaves in the WFG, respectively. Second, since the information about a detected deadlock is readily available at the initiator process, rather than distributed among different processes, it significantly simplifies the task of deadlock resolution, and helps to reduce system overhead associated with the resolution. The time complexity of our algorithm is also better than or equal to the existing algorithms.

Keywords - Deadlock detection and resolution, wait-for graph, generalized-deadlock, algorithms, distributed systems.

1 Introduction

A deadlock is a system state in which every process in some group requests resources from other processes in the group, and then waits indefinitely for these requests to be satisfied. Because distributed systems are vulnerable to deadlocks, the problems of deadlock detection and resolution have long been considered important problems in such systems.

Existing deadlock detection algorithms can be classified in terms of their underlying resource request models [8], such as AND [2, 4, 5, 13, 14, 15], OR [2, 7, 11, 12], AND-OR [6], and p-out-of-q

[*]This work was supported in part by Rome Laboratory, Air Force under Contract No. F30602-93-C-0247, by NASA under Grant No. NAGW-4080, and by NSF under Grant No. CDA-9313624.

[†]Department of Computer Science, University of Illinois at Urbana Champaign, Urbana, IL 61801

[‡]School of Computer Science, Florida International University - the State University of Florida at Miami, University Park, Miami, FL 33199, {deng,attie,weisun}@fiu.edu

[1, 9, 16] models. In an AND model, all requested resources are required. In an OR model, any one of a number of requested resources is sufficient. Finally, in the p-out-of-q model, requests are issued for q resources, and the issuing process remains blocked until any p of these are acquired. A distributed deadlock based on the p-out-of-q model is called a *generalized* deadlock. If $p = q$ (resp. $p = 1$), then the AND (resp. OR model) is seen to be a special case of the p-out-of-q model.

To detect deadlocks in a distributed system, the global state of the system is commonly modeled by a logical structure called the *wait-for graph* (WFG). A WFG is a directed graph, in which a vertex represents a process, and an edge (i, j) indicates that process i has requested a resource from process j, and j has not granted the request. As indicated in [9], detecting generalized deadlocks in distributed systems is a difficult problem, because it requires detection of a complex topology in the global WFG. Among the distributed deadlock detection algorithms in the literature, only [1, 9, 16] address this problem. All three algorithms use a distributed snapshot-based approach. The basic idea behind these algorithms can be briefly described as follows: They have either two distinct phases [1, 16], or one phase consisting of two overlapped sweeps [9], of message transmission. In the first phase (or the outward sweep), they record a snapshot of the WFG, which is distributed among all the processes in the system. In the second phase (or the inward sweep), they reduce the distributed WFG by simulating the unblocking of those processes whose requests can be granted.

The proposed algorithm is based on a new approach that differs from the above. A process initiates the algorithm when it blocks on a resource request. Instead of recording a distributed snapshot, the algorithm incrementally constructs an "image" of the WFG, which is stored locally at the initiator process. The algorithm is composed of a sequence of stages. In the first stage, the initiator process i sends an inquiry (called FORWARD) message to each process j which it is waiting for, and then each j reports its state information to i via a BACKWARD message; in the second stage, i sends a FORWARD message to each process k which some j which was involved in the first stage is waiting for, and then each k reports its state information to i via a BACKWARD message. This process continues *stage-by-stage* in a similar manner. At each stage, the new processes are those processes that the processes of the previous stage are waiting for. At the end of each stage, the WFG is locally (at process i) updated (based on the new information from the received BACKWARDs), reduced, and checked for the existence of a deadlock.

This new algorithm may be seen as a "hybrid" algorithm, with both centralized and distributed aspects. We argue that it combines the simplicity and efficiency (with respect to the total number of messages generated) of a centralized algorithm, and the flexibility and robustness of a distributed algorithm. A single instance of our algorithm is executed in an essentially centralized fashion, with the initiator being responsible for collecting system state information, constructing a WFG, and reducing it to detect a deadlock. Consequently, in addition to a simpler structure, each instance of our algorithm only requires $2n$ messages in the worst case, compared to the best available message complexity [9] of $4e - 2n + 2l$ among the existing algorithms [1, 9, 16], where n, e, and l are the number of nodes, edges, and leaf nodes in the WFG, respectively. The latter complexity is $O(n^2)$ in the worst case. Since information must be obtained from every process involved in a deadlock, the message complexity of our algorithm is optimal to within a constant factor. Also, worst case time complexity of our algorithm is better than or equal to the existing algorithms. (See Section 5 for detailed analysis.)

From the global view of the system, however, our algorithm is a distributed algorithm in nature. There is no *designated* central controller in our system, as in a typical centralized algorithm, to control the detection of deadlocks in the system. All processes in our algorithm play the same symmetric role. Any process may initiate deadlock detection as necessary, and all instances of deadlock detection are executed independently and concurrently, as in other distributed deadlock detection algorithms. Furthermore, any distributed deadlock detection algorithm will typically have many instances active at any one time. Thus the centralized aspect of our algorithm does not lead to a communication bottleneck around the initiator(s).

Another primary advantage of our approach is its support to deadlock resolution. Because the WFG is constructed at the initiator, deadlock resolution is simplified because the global state information required is locally available at the initiator process rather than distributed among all the involved processes. Thus the choice of which process(es) to abort in order to break the deadlock can be made locally by the initiator, rather than requiring another round of communication. In addition, the availability of the global information makes it possible to construct optimal or near optimal deadlock resolution strategies, e.g. to minimize the number of processes needed to be aborted in order to break the deadlock. Based on the proposed deadlock detection algorithm, we have developed a simple and efficient deadlock resolution algorithm, which requires only a slight increase in message complexity. [3] Unlike [1, 9, 16], the proposed algorithm does not require any storage whose size is pre-determined by the size of the system. Hence, it is suitable for use in an environment where processes are created and terminated dynamically.

The rest of the paper is organized as follows. In Section 2, the model of computation is defined. In Section 3, the deadlock detection algorithm is provided. Section 4 provides a proof of correctness, and section 5 analyzes the complexity of the algorithm. We conclude the paper in Section 6.

2 Model of computation

A distributed system is composed of n processes, each of which has a system-wide unique identity. Each pair of processes is connected by a logical channel [9]. There is no shared memory in the system. Processes communicate by message passing, and message delays on a channel are arbitrary but finite. A destination process receives messages in the same order as they are sent by a source process. Messages are neither lost nor duplicated, and are transmitted error-free. There are two types of messages. *Computation messages* are generated by the underlying computation of processes in the system, including REQUEST, REPLY, CANCEL and ACK messages; *control messages* are generated by the execution of the deadlock detection algorithm, including FORWARD and BACKWARD messages which will be discussed in Section 3.

The following data structure is used at a process $i, i = 1...n$, to keep track of its state. We assume that the logical time at each process is maintained as specified in [10].

t_i:	the current logical time at i,
t_block_i:	the logical time at which i last blocked,
out_i:	the set of processes for which i is waiting,
in_i:	the set of tuples $< k, t\_block_k >$, where k is a process waiting for i and t_block_k is the logical time at which k sent its request to i,
p_i:	the number of replies required for i to unblock.

Each process is either *active* or *blocked*. An active process can send both computation and control messages. A blocked process, however, can only send control messages or ACK messages, i.e., its underlying computation is suspended. A process i becomes blocked after it sends a p_i-out-of-q_i request (via REQUEST messages) to q_i other processes. It records these processes in out_i. When a process j receives a REQUEST message from process i, it records $< i, t\_block_i >$ in in_j and immediately sends an ACK back to process i to acknowledge the receipt of the request [1]. A

[1]Notice that this ACK can be the same acknowledgement used by the underlying network to guarantee reliable communication channels. There is no need to send a separate message to deliver such information.

REPLY message denotes the granting of a request. When j sends a REPLY to i, $< i, t\_block_i >$ is removed from in_j. The process i becomes unblocked (goes from blocked to active state) *only* when any p_i out of the q_i requests are granted, namely, i receives REPLY messages from at least p_i out of the q_i processes. When i unblocks, it sends CANCEL messages to withdraw the remaining $(q_i - p_i)$ requests it had sent.

Each REQUEST, REPLY, or ACK message is timestamped with the requester's logical clock value [10] at which it blocked, so that an ACK or a REPLY can be matched with its corresponding request. ACK or REPLY messages with unmatched timestamps are discarded.

When j is in out_i, we say process i is *waiting* for process j or there is a *wait-for* edge from i to j. The set of processes in out_i is called the *dependent set* of process i. If $p_i \neq 0$, then process i is blocking on a p_i-out-of-$|out_i|$ request. When i receives a REPLY message from $j (\in out_i)$, j is removed from out_i and p_i is decreased by one. $(|out_i| - p_i)$ remains a constant. When p_i is decreased to zero, i unblocks.

A process is deadlocked when it belongs to a generalized deadlock set which is defined as follows.

Definition 1 A *generalized deadlock set* (deadlock set in short) is a set S of processes in the system which satisfies the following conditions:
(1) $\forall i \in S$, i blocks on a p_i-out-of-q_i request,
(2) $\forall i \in S$, $\exists C_i \subseteq out_i$, $C_i \subseteq S \bigwedge |C_i| \geq q_i - p_i + 1$, and
(3) $\forall i \in S$, $\forall j \in C_i$, no REPLY message is under transmission from j to i.

We use a data structure called WFG to model a distributed system.

Definition 2 A wait-for graph WFG $< N, E >$ is a directed graph, where a vertex in N models a process, and an edge in E from vertex i to vertex j indicates that i blocks and waits for j to grant some resource. Every vertex i has two pieces of information: $i.p$ and $i.q$, which indicates that i blocks on a $i.p$-out-of-$i.q$ request, where $i.q$ is the outdegree of i.

A process j is said to be *reachable* from process i iff there is a directed path in the WFG from i to j.

Definition 3 A *generalized tie* (*tie* in short) is a sub-graph $< N_t, E_t >$ of a WFG, where N_t is a nonempty set of vertices, and E_t is the set of wait-for edges between the vertices in N_t, such that each $i \in N_t$ has at least $i.q - i.p + 1$ outgoing edges in E_t.

3 An Efficient Distributed Deadlock Detection Algorithm

Whenever a process i blocks on a p_i-out-of-q_i request, it initiates an instance of deadlock detection (i is called the initiator of the instance). Every instance of the algorithm is treated independently from others. The control messages of a particular instance of the algorithm are identified by timestamps which consist of the initiator's identity and the logical time at which the initiator blocked. Control messages belonging to different instances of the algorithm have different timestamps, and thus can be distinguished. We will thus focus our attention on a single instance of deadlock detection in the ensuing discussion. The following precondition needs to be satisfied for the algorithm to be invoked:

Precondition for invocation of the algorithm:
The *precondition* for i to invoke the algorithm is that i is blocked on a p_i-out-of-q_i request and has received an ACK message from every process j in its dependent set.

This is to ensure that $< i, t\_block_i >$ has already been recorded in in_j. The precondition is necessary to guarantee that the proposed algorithm will detect every deadlock in the system. The initiator i will then incrementally construct a data structure called WFG_i, which will be used for deadlock detection.

3.1 Outline of the Algorithm

The proposed algorithm consists of a series of similar stages. In the first stage, the initiator i sends a FORWARD message to every process $j \in out_i$. When j receives the FORWARD, it sends the current values of t_block_j, out_j, in_j, and p_j via a BACKWARD message back to i. When i receives the BACKWARD, it expands WFG_i by inserting a vertex j and edges associated with j into WFG_i. After i receives all BACKWARD messages sent by the processes in out_i, the algorithm enters the second stage. At the second stage, the same process is repeated between i and the processes in out_j for every $j \in out_i$. This process continues *stage-by-stage* in a similar manner. At the end of each stage, a reduction is performed to WFG_i to remove those edges that will not belong to any tie and those vertices that are unreachable from the initiator; deadlock detection is then attempted by searching for a tie in the reduced WFG_i. The algorithm terminates in one of the following three cases: (a) A tie is found in WFG_i, (b) all outgoing edges of vertex i are removed during the reduction, and (c) all processes reachable from i have been inserted into WFG_i and no tie is found in WFG_i. A deadlock is detected only in case (a). WFG_i is deleted and the memory space is released in case (b) or (c).

3.2 Definitions and conventions

We first define the data structure used to store WFG_i. (Note: The data structure described here plays a different role from the one defined in Section 2.) A vertex j in WFG_i is represented by a tuple $< j.t, j.out, j.in, j.p >$.

The rule for constructing WFG_i is as follows: When the initiator i receives a BACKWARD message (t_block_j, out_j, in_j, p_j) from a process j, it inserts a new vertex j into WFG_i, with the following assignments:

$j.t := t\_block_j$, $j.out := out_j$, $j.in := in_j$, and $j.p := p_j$.

$|out_j|$ and $j.p$ specify that process j blocks on a $j.p$-out-of-$|out_j|$ request when $j.p > 0$ (Definition 2).

Definition 4 (1) An *edge* (j, k) belongs to WFG_i if j and k are two vertices in WFG_i and $k \in j.out$. (2) There are two types of edges in WFG_i: An edge (j, k) is a *matched edge* if $< j, j.t > \in k.in$; otherwise, it is an *unmatched edge.*

(j, k) is a matched edge in WFG_i if and only if the wait-for relation recorded at vertex j and the waited-by relation recorded at vertex k refer to the same request. All unmatched edges must be removed from WFG_i. The rules for removing an unmatched edge (j, k) are as follows: (1) Remove k from $j.out$, (2) decrease $j.p$ by one, and (3) set $j.out$ to empty if $j.p$ is decreased to zero (i.e., remove all outgoing edges of j from WFG_i).

Definition 5 A matched edge (j, k) in WFG_i is a *reducible edge* if $k.out = \emptyset$.

A reducible edge $(j,\ k)$ means the request from j to k is grantable. Therefore, all reducible edges should also be removed from WFG_i because they will not contribute to a deadlock. The rules for removing a reducible edge (j, k) are as follows: (1) Remove k from $j.out$, (2) decrease $j.p$ by one, (3) set $j.out$ to empty if $j.p$ is decreased to zero (which will cause the edges incident to j to become reducible), and (4) remove the new reducible edges created in (3) recursively.

Definition 6 A vertex j $(j \neq i)$ in WFG_i is an *invalid* vertex if there is no directed path from i to j in WFG_i.

We remove invalid vertices from WFG_i and put them in $Pool_i$ which is a data structure consisting of invalid vertices removed from WFG_i. Initially, $Pool_i$ is empty.

3.3 The algorithm

A more formal description of the algorithm is presented below. The initialization of the proposed algorithm is composed of three steps: (1) WFG_i is created at the initiator i, containing only one vertex i, where $i.t := t\_block_i$, $i.out := out_i$, $i.in := in_i$, and $i.p := p_i$; (2) $New_i := out_i$; and (3) $Pool_i := \emptyset$. New_i is the set of new vertices which will be inserted into WFG_i at the next stage.

Each stage of the algorithm can be divided into the following five steps:

1. For every $j \in New_i$, if it is in $Pool_i$, then remove it from $Pool_i$ and insert it into WFG_i; otherwise, send a FORWARD message to process j.
2. When i receives a BACKWARD message from j, a new vertex j is inserted into WFG_i. (The edges associated with j are inserted automatically.) After i receives all BACKWARD messages sent by the processes in New_i, go to the next step.
3. (a) Remove all unmatched edges from WFG_i.

   (b) Remove all reducible edges from WFG_i.

   (c) Remove all invalid vertices from WFG_i and New_i.

4. $New_i = (\sum_{j \in New_i} j.out) - WFG_i.N$, where $WFG_i.N$ is the set of vertices in WFG_i.

5. One of the following actions is taken:

   (a) If there is a tie in WFG_i, a deadlock is detected and the algorithm terminates.

   (b) If $i.p = 0$, the algorithm terminates, with no deadlock detected.

   (c) If $New_i = \emptyset$ and there is no tie in WFG_i, the algorithm terminates, with no deadlock detected.

   (d) Otherwise, go to the next stage.

When a process j receives a FORWARD message from the initiator i, it responds simply by sending a BACKWARD message $(t\_block_j, in_j, out_j, p_j)$ to i.

4 Correctness of the Algorithm

It suffices to prove correctness by considering only a single instance of the algorithm. Without loss of generality, assume that i is the initiator. Due to space limitations, only a proof sketch is presented here. The detailed formal proof is provided in [3].

We first introduce the notations and conventions used in the proof. For the purpose of the correctness proof only, we introduce T as the global physical (or real) time of the system, in contrast to the logical time t used in the algorithm. It should be noted that our algorithm does not depend on a global (physical time) clock.

- T_j: the physical time at which process j sends a BACKWARD message to the initiator i.
- R_j: the last p_j-out-of-q_j request issued by j before T_j.

Lemma 1 If a matched edge (j, k) is added to WFG_i during the execution of the algorithm, then no REPLY message for R_j is sent from k to j before T_k.

Proof: We establish the contrapositive of the lemma. Let $t\_block_j(R_j)$ denote the logical time at which j blocked immediately before issuing the request R_j. Since R_j is the last request issued by j

before T_j, the variable $j.t$ will be set to $t\_block_j(R_j)$ upon receipt by i of the BACKWARD message sent by j (by construction of the algorithm). Now suppose k sends a reply to R_j before time T_k. Upon sending this reply, k removes $< j, t\_block_j(R_j) >$ from in_k. There are now two cases.

Case 1: k does not receive another request from j before T_k.
Then, the in_k field of the BACKWARD message that k sends to j will not contain a tuple of the form $< j, t >$ (for some value t). Upon receipt by i, $k.in$ will be assigned this value of in_k. Since $k.in$ and $j.t$ are assigned to exactly once during the construction of WFG_i, it follows, by definition 4, that WFG_i will never contain a matched edge (j, k).

Case 2: k receives another request from j before T_k.
Call this request R'_j. Then, the in_k field of the BACKWARD message that k sends to j will contain the tuple $< j, t\_block_j(R'_j) >$, where $t\_block_j(R'_j)$ denotes the logical time at which j blocked immediately before issuing the request R'_j, and no other tuples with j as the first element. Upon receipt by i, $k.in$ will be assigned this value of in_k. Since the receipt by j of $k's$ reply to R_j occurs between the first time j blocked (immediately before issuing R_j) and the second time j blocked (immediately before issuing R'_j), we have $t\_block_j(R_j) < t\_block_j(R'_j)$, by Lamport's clock condition [10]. Since $j.t$ is set to $t\_block_j(R_j)$ (see above), and $t\_block_j(R_j) \neq t\_block_j(R'_j)$, we have, again by definition 4, that WFG_i will never contain a matched edge (j, k).

In both cases we have established the contrapositive, and so the lemma is established. □

Theorem 1 No false deadlock is detected by the proposed algorithm. That is, every tie in the WFG constructed by the algorithm corresponds to a deadlock set in the system.

Proof: The proposed algorithm detects a deadlock only when there is a tie in WFG_i. We prove by contradiction that, if a tie G is found in WFG_i, then every process involved in G never unblocks. Without loss of generality, assume that j is the first process in G which unblocks (on R_j). Let N_j be the set of vertices in G that j has edges outstanding at. Every vertex in N_j represents a process in the system for which j is waiting at time T_j. There are at least $(q_j - p_j + 1)$ vertices in N_j because G is a tie. Hence, before process j unblocks on R_j, at least one process in N_j must send a REPLY message to it. Suppose k is such a process. By Lemma 1, process k sends the REPLY to process j after T_k. But process k is blocked on R_k at T_k because $k \in G$. Before process k sends the REPLY, k has to unblock on R_k first. That is in contradiction with the assumption that j is the first process in G to unblock. □

Theorem 2 Every deadlock in the system will be detected by the proposed algorithm. That is, there will be a tie in the WFG constructed by some instance of the algorithm for every deadlock set in the system.

Proof: Let G be a tie in the underlying system. For all $j \in G$, j blocks on a p_j-out-of-q_j request. Let i be the last process in G to initiate an instance of deadlock detection. We prove by contradiction that i detects a deadlock. Suppose i does not detect a tie in WFG_i. Let WFG_i^f be the final WFG_i before the deadlock detection initiated by i terminates.

Consider an arbitrary vertex j of G. Since j is a member of a tie G, j will be blocked forever (after the physical time at which G was formed, and in the absence of deadlock resolution). This is easily seen by considering the first member j' of G to unblock, and noting that (by definition 3), there is at least one member j'' of G which must reply to j''s outstanding request before j' can unblock. Since j'' is itself blocked, j'' must unblock before replying, which contradicts the fact that j' is the first member of G to unblock.

Now if i is the last process in G to initiate deadlock detection, then G must already be formed when i initiates this instance, by virtue of our precondition (given on page 6) for invoking the deadlock detection algorithm. Since every process in G is blocked forever from this moment (in physical time) onwards, all state variables (i.e., t_i, t_block_i, out_i, in_i) of all processes in G will remain unchanged. From this fact and the construction of our algorithm (in particular, none of the edges in G will be eliminated by step 3 of the algorithm) we see that:

> If an arbitrary process j of G is a process of WFG_i^f, then so is k, for every $k \in G$ such that j waits for k. Furthermore, (j, k) is a matched edge in WFG_i^f. (*)

Suppose $G^f = < N^f, E^f >$ is a sub-graph of WFG_i^f, where N^f is the set of vertices in both WFG_i^f and G, and E^f is the set of edges between vertices in N^f. $N^f \neq \emptyset$ because $i \in N^f$. For all $j \in N^f$, since j is a process in the tie G, there are at least $(q_j - p_j + 1)$ processes in G for which j is waiting. By (*) above, each of these processes is a vertex in G^f, and the wait-for edges from process j to these processes are also in G^f. That is, j has at least $(q_j - p_j + 1)$ edges outstanding at other vertices in G^f. Hence, G^f is a tie, which contradicts the assumption that there is no tie in WFG_i. Hence the theorem holds. $\square$

5 Performance Analysis

In this section, we consider the time, message, data-traffic and space complexities of the proposed algorithm. In calculating the message complexity, we only consider logical message transfers. Based on the type of underlying communication network, a logical message may result in the transfer of a number of physical messages, which is not an issue here. Assume that message delay on a logical channel (one hop) is 1 unit of time. It is important to notice that a WFG is a logical structure. An edge in the WFG does not correspond to a physical communication channel. Therefore, whether a message is sent along an edge of the WFG has no affect on the cost of transmitting the message. For simplicity, we consider, in the following analysis, that a message transmisstion between any pair of processes in the system has the same cost.

The *message complexity* is the number of messages transmitted; the *time complexity* is the time it takes for the initiator to detect a deadlock; the *data-traffic complexity* is the total length of data transmitted by the algorithm. For simplicity, consider the length of a control message in [1, 9, 16] as 1 unit. We now analyze the proposed algorithm. Consider a system depicted by WFG_s with n processes, e wait-for edges and a diameter of d, and suppose every process is reachable from the initiator i. Each process in WFG_s receives at most one FORWARD message and sends at most one BACKWARD message in a single instance of the algorithm. The number of FORWARD (or BACKWARD) messages is not greater than n. Hence, the worst-case message complexity is $2n$. The proposed algorithm is composed of a series of stages. Let $H(j)$ denote the stage number when j sends a BACKWARD to the initiator. $H(j)$ equals to the length of the shortest path from i to j in WFG_s. Thus, $H(j) \leq d$, which implies that the number of stages of the algorithm can not be greater than d. Each stage takes two (hops). Hence, the worst-case time complexity of the proposed algorithm is $2d$. The worst-case data-traffic complexity of the proposed algorithm is $e + 2n$, which is still better (by a constant factor) than the best result $(4e - 2n + 2l)$ of the other three algorithms. The detailed analysis for the data-traffic complexity is provided in [3].

Looking at Table 1, we see that the proposed algorithm is comparable to the algorithms given

Criterion	Bracha-Toueg [1]	Wang et al. [16]	Kshem.-Singhal [9]	Our Algorithm
Number of Messages Sent	$4e$	$6e$	$4e - 2n + 2l$	$2n$
Delay	$4d$	$3d + 1$	$2d$	$2d$
Total Size of Data Sent	$4e$	$6e$	$4e - 2n + 2l$	$e + 2n$

Table 1: Performance comparison between our and existing algorithms. Given a WFG, n = number of processes, l = number of leaf processes, e = number of edges, d = diameter.

in [1, 9, 16] in terms of the time complexity and the data-traffic complexity. In terms of message complexity however, the message complexity of the proposed algorithm is $O(n)$; the message complexities of the algorithms in [1, 9, 16] are all $O(e)(= O(n^2)$ in the worst case). This is the first deadlock detection algorithm to our knowledge with message complexity linear in the number of processes of the system, i.e., optimal to within a constant factor. Although the data-traffic complexity of our algorithm is comparable to the other algorithms, we remark that the reduced message complexity of our algorithm reduces the overhead associated with message creation, transmission, and receipt, which are significant in a practical sense, considering the very small-sized messages used in [1, 9, 16]. For example, because a message in [1, 9, 16] contains only several integers, the message header may be much larger than the message body.

The worst-case time, message, and data-traffic complexities are often unreached in our algorithm. The algorithm may terminate (due to the detection of a tie or due to the reduction of the initiator) before every process reachable from the initiator gets involved in the deadlock detection. Therefore, the average message, time, and data-traffic complexities are expected to be better than the worst-case complexities. Unlike our algorithm, the algorithms in [1, 9, 16] require the involvement of every process reachable from the initiator for every instance of the algorithm, and thus require corresponding message transfers.

In algorithms [1, 9], the memory space needed by each process to store the snapshots for different instances of deadlock detection is $O(n^2)$. Therefore, the total space required is $O(n^3)$. In our algorithm, no snapshot needs to be stored at any process. The initiator requires at most $O(n^2)$ space to store the constructed WFG. In the worst case, when all the n processes initiate deadlock detections simultaneously, $O(n^3)$ space in all is needed. Thus our algorithm is comparable to the other algorithms in its space complexity. Unlike [1, 9, 16], our algorithm does not require any array whose size is determined by n, and hence is suitable for use in an environment where processes are created and terminated dynamically.

6 Conclusion and Discussion

We have presented a new algorithm for detecting generalized-deadlock in distributed systems, and proved its correctness. Our approach differs from the existing algorithms in that, instead of using distributed snapshot, our algorithm is based on the idea of locally constructing a wait-for graph at an initiator. Instead of using the diffusion computation to propagate control messages level-by-level from the initiator to all reachable processes, our algorithm lets the initiator communicate directly with those processes. Under this new approach, our algorithm combines the simplicity and efficiency of centralized algorithms, and the flexibility and robustness of distributed ones. It is shown that the proposed algorithm requires only $2n$ messages in the worst case, compared to the best message complexity, $4e - 2n + 2l$, of the existing algorithms (which is $O(n^2)$ in the worst case), where n, e and l are the number of vertices, edges and leaves of the WFG, respectively. Its time complexity is better than or equal to those algorithms.

The algorithm is the first distributed algorithm which detects every deadlock and detects no false deadlock with a worst-case message complexity of $O(n)$. Although some messages are of variable length with a maximum size of n, the total length of data transmitted by the proposed algorithm is still better than the best result of the previously published algorithms [1, 9, 16].

An equally important contribution of the algorithm is that it significantly eases the task of deadlock resolution in distributed systems. For every deadlock set in the system, a corresponding tie will be present in the WFG constructed by some instance of the algorithm. Not only deadlocked processes but also the wait-for relations between them are detected by our algorithm; whereas no other distributed deadlock detection algorithm known to us tries to provide the whole structure of a deadlock set. This nice property not only makes deadlock resolution much easier, but also makes optimum deadlock resolution possible. The issues of deadlock resolution are discussed in [3]. Again in [3], both the deadlock detection algorithm and the deadlock resolution algorithm for the p-out-of-q model are modified to detect deadlocks in the AND-OR model.

References

[1] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127 – 138, 1987.

[2] K. M. Chandy and J. Misra. Distributed deadlock detection. *ACM Transactions on Computer System*, 1(2):144 – 156, May 1983.

[3] S. Chen, Y. Deng, and P. C. Attie. Deadlock detection and resolution in distributed systems based on locally constructed wait-for graphs. *Technical Report, School of Computer Science, Florida International University*, August 1995.

[4] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley. A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions on Software Engineering*, 15(1):10 – 17, January 1989.

[5] V. D. Gligor and S. H. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5):435 – 440, September 1980.

[6] T. Herman and K. M. Chandy. A distributed procedure to detect AND/OR deadlocks. *Department of Computer Science, Technical Report, TR-LCS-8301, University of Texas, Austin, TX*, February 1983.

[7] S. T. Huang. A distributed deadlock detection algorithm for CSP-like communication. *ACM Transactions on Programming Languages and Systems*, 12(1):102 – 122, January 1990.

[8] E. Knapp. Deadlock detection in distributed database. *ACM Computing Surveys*, 19(4):303 – 328, December 1987.

[9] A. D. Kshemkalyani and M. Singhal. Efficient detection and resolution of generalized distributed deadlocks. *IEEE Transactions on Software Engineering*, 20(1):43 – 54, January 1994.

[10] L. Lamport. Time, clocks, and the order of events in a distributed system. *Communication of the ACM*, 21:558 – 565, July 1978.

[11] J. Misra and K. M. Chandy. A distributed graph algorithm: Knot detection. *ACM Transactions on Programming Languages and Systems*, 4(4):678 – 686, October 1982.

[12] N. Natarajan. A distributed scheme for detecting communication deadlocks. *IEEE Transactions on Software Engineering*, SE-12(4):531 – 537, April 1986.

[13] R. Obermarck. Distributed deadlock detection. *ACM Transactions on Database Systems*, 7(2):187 –208, June 1982.

[14] M. Roesler and W. A. Burkhard. Resolution of deadlocks in object-oriented distributed systems. *IEEE Transactions on Computers*, 38(8):1212 – 1224, August 1989.

[15] M. K. Sinha and N. Natarajan. A priority based distributed deadlock detection algorithm. *IEEE Transactions on Software Engineering*, SE-11(1):67 – 80, January 1985.

[16] J. Wang, S. Huang, and N. Chen. A distributed algorithm for detecting generalized deadlocks. *Technical Report, Department of Computer Science, National Tsing-Hua University*, 1990.