On-Line Detection and Resolution of Communication Deadlocks*

Wee K. Ng Chinya V. Ravishankar
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122

Abstract

We present a new distributed algorithm that detects and resolves communication deadlocks on-line, i.e., simultaneously detects and resolves deadlock as communication requests are made, at no additional message traffic overhead, and with bounded delay between the occurrence and detection of a deadlock. This is achieved via a novel technique for detecting knots, which suffice for the existence of communication deadlocks. Current distributed deadlock detection algorithms lack these features. Thus the algorithm is suitable for soft real-time systems and large distributed systems. We also prove that the algorithm detects communication deadlocks and that it is able to deal with false deadlocks.

1 Introduction

Deadlocks have been categorized into two types in the literature [12, 19, 21]. In the *resource* model (AND-model), a process that has multiple outstanding requests for resources suspends itself until all of them are serviced. Resources usually cannot be duplicated. In the *communication* model (OR-model), a process may proceed as soon as at least one of the outstanding requests is serviced. The process may thereupon discard the other requests. Some software resources are usually managed this way.

Many distributed algorithms have been proposed to detect deadlocks in each of these categories [3, 4, 5, 7, 13, 16, 17, 18, 20, 21]. There are four categories of deadlock algorithms [12]: *path-pushing*, *edge-chasing*, *diffusing computations* and *global state detection*. In this paper, we restrict our attention to edge-chasing (or probe-based) algorithms [5, 13, 16, 18, 20, 21]. These are elegant because they do not require the construction of a WFG (Waits-For Graph) as in path-pushing algorithms. Furthermore, probes are simply special messages exchanged among processes, and communication deadlocks are caused by the sending/receiving of messages. However these algorithms exhibit the following shortcomings:

1. Message traffic overhead is high because probing messages are required to perform detection. Most algorithms are evaluated on the basis of the number of messages exchanged to detect deadlocks and omit the messages exchanged when there are no deadlocks. The algorithms are more expensive than they seem.
2. Deadlock detection is usually initiated when a process has waited long enough [21] or when a higher priority process is blocked by a lower priority process [3, 13, 18]. These are arbitrary criteria.
3. Deadlock detection and resolution are performed separately, usually duplicating effort [19]. The duration of deadlocks wastes resources and increases response time to user requests. Unfortunately, deadlock resolution is sometimes neglected [1, 9] or is not handled properly [14, 17].
4. These inefficiencies and complications suggest that current algorithms are not scalable to large distributed algorithms. Message overhead adds unnecessary traffic to the network, and delayed deadlock detection and resolution wastes resources and reduces throughput.

We propose a new algorithm that overcomes these shortcomings. Our algorithm performs deadlock detection and resolution concurrently *on-line* at *no* additional message traffic overhead (see Section 5). The rest of the paper is organized as follows: The next section describes the algorithm. Section 3 shows that the algorithm is correct. Section 4 is an analysis of the

*This work was supported in part by the Consortium for International Earth Science Information Networking.

algorithm from various perspectives. In Section 5, we present several properties of our algorithm and conclude the paper.

2 Algorithm description

2.1 Preliminaries and assumptions

We model a distributed system as a graph with nodes corresponding to physical sites and edges corresponding to communication channels. We assume more than one process per site so that some of them may continue even when others are blocked. In addition, every process has a *mailbox* thread that performs forwarding of incoming requests even when the process is blocked (see Section 2.6). The terms *site* and *node*, are used interchangeably. We also assume that a transmitted message may be delayed but is always delivered. Moreover, messages are received in the order sent.

The notion of a *resource* in a communication deadlock is not as well-defined as in a resource deadlock. We assume here that it is a service request, such as a request for the results of some computation. Hence if a process blocks after sending a request to another process, it is waiting for a *service resource*. An important assumption is that once a process is blocked waiting for a reply, it does not abort for other reasons.

We adopt Hoare's Communicating Sequential Process [10] as the model for interprocess communication. A process may make multiple requests but it waits until *at least one* of them is serviced. This corresponds to the OR-model of deadlock in [12, 19, 21]. When a process enters a *communication* phase (see Section 2.2), it initiates a *transaction* corresponding to a set of processes that it intends to communicate with. These processes are called *dependents* (see Definition 3.1 for a formal definition). It then sends the request to the dependents, which in turn forward the request to other processes *only if* they are unable to service the request themselves. The transaction completes when one or more of the dependents have serviced the request.

It is well-known that a cycle is necessary for the existence of a deadlock, but insufficient for a communication deadlock [16, 21]. Due to the nature of a communication deadlock (OR-model), a process with multiple outstanding requests may be part of a cycle and not deadlocked if one of the requests can be granted. Communication deadlocks result when knots are formed [16, 21].

A *knot* is a minimal closed subgraph with no edges directed out of it, and where every node is reachable from every other node in the subgraph. Thus it consists of a set of closed cycles (see Definition 3.2 for a formal definition of a knot). The detection of communication deadlocks in distributed systems reduces to the detection of a knot. Observe that a process that is not part of a knot may still be in a deadlock if all its out-edges lead into knots. Several examples of knots are given in Figures 1(a), 2(a), 2(b), and 3(a).

The On-Line Deadlock Detection and Resolution (ODDR) algorithm that we propose in this paper exhibits four new features: (1) explicit representation of *deadlock context*, (2) a new algorithm for distributed knot detection, (3) on-line deadlock detection and (4) immediate resolution after detection. The following four subsections give an informal and intuitive description of each of them. The algorithm is formally presented in Section 2.6.

2.2 Deadlock context

A *communication transaction* is a cascade of messages with a single initiator, exchanged with the objective of responding the initiator's request. When a deadlock results, the transaction defines the *context* for the deadlock. Our algorithm differs from other algorithms in that the context (transaction) is explicitly represented during deadlock detection. In particular, the context is contained in each *probe list* (see Section 2.3).

Consider Figure 1(a). Suppose process 1 requires the services of processes 2 or 3 in order to complete some task. It initiates a communcation transaction and sends requests to processes 2 and 3. Assuming that they are unable to service the request, they in turn seek the services of processes 4 and 5. This cascade of requests occurs under the context of process 1.

This approach presumes a unique, system-wide identification for a context. We assume that each process contains two execution threads, one thread being the process thread, and the other being the mailbox thread mentioned earlier. A blocked process may never initiate a communication transaction, but the mailbox thread may continue to forward requests. We concatenate the machine ID and process ID to get the context ID.

2.3 Knot detection: Probe diffusion

Several algorithms have been proposed for knot detection [2, 15]. In [2], a *clustering* technique is used, in which cycles of clusters are detected and merged into bigger clusters. In [15], a diffusing computation

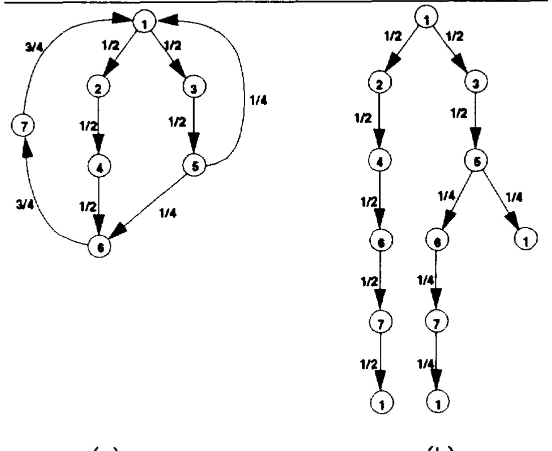

Figure 1: A knot (communication deadlock) under the context of process 1. The requests are indicated by the directed edges. The labels on the edges are numerical probe values (see Section 2.3) sent along with the request. In (b), the cascade of requests is depicted as a tree (see Section 2.3).

approach is used. Our knot detection algorithm is a form of diffusing computation that uses *probes*, but it differs from previous algorithms in that the probes carry additional information to permit the distributed detection of a knot.

The idea of our knot detection algorithm is that the initiator of a transaction *diffuses* a numerical probe value of 1 through the processes involved in the transaction. If a knot exists, the sum total of the values it receives is equal to 1 as well. If the initiator sends probes to m processes, it divides its initial probe value of 1 by m and sends the values $1/m$ to each of the m processes. Similarly, each process k that receives an incoming sum total probe value of V propagates the value V/m_k to each of the m_k processes that it forwards the request to. It is important to keep incoming probes values from different transactions distinct. This is achieved by storing the context identification in the probe as well. If a knot exists, the initiator will receive back a sum total probe values of 1 and declares a deadlock.

In Figure 1(a), process 1 sends a probe value of 1/2 to processes 2 and 3. Since process 2 is unable to service the request, it forwards the request to process 4 together with a probe value of 1/2, since process 4 is the only process it forwards to. Observe that process 5 sends a probe value of 1/4 to processes 6 and 1 since it receives 1/2 from process 3. Process 6 receives a sum total probe value of $1/2 + 1/4 = 3/4$ and forwards it to process 7 without any division.

The diffusion algorithm constructs a tree for the initiator where the *root* and *leaves* of the tree becomes the initiator when a knot exists. Figure 1(b) shows the tree for process 1 where the edges are labeled with the probe values. Observe how the probe values diffuse in numerical value from process 1 through the tree to the leaves where they are collected by process 1 again.

2.4 On-line detection

The knot detection algorithm just described may be initiated either after a WFG is formed, or during the incremental formation of the WFG. Most previous deadlock detection algorithms [1, 3, 5, 16, 21] adopted the former approach, and some time elapses before the deadlock is detected. The only algorithm to approximate *on-line* detection is by Isloor and Marsland [11]. Their proposed on-line algorithm detects deadlocks at the earliest possible instant—the time of making decisions about data allocation at the concerned site. However the algorithm is for the detection and resolution of resource deadlocks, not communication deadlocks.

Our knot detection algorithm is executed as the requests are made: Each time a process j is unable to service an incoming request from process i, it forwards the request to other processes together with the probe it has received as part of the incoming request. The probe value is divided by the number of processes forwarded to, as explained in the previous section. To avoid loss of precision, the value $1/(m_1 m_2 \cdots m_k)$ may be propagated as the product $m_1 m_2 \cdots m_k$.

With this approach, a deadlock is detected *as soon as* it is formed. This does not imply instantaneous detection as probes take time to propagate back. Consider Figure 1(a). Assuming that the request from process 5 to process 6 completes the knot, the tree in Figure 1(b) shows that two requests (6,7) and (7,1) are needed before process 1 detects a deadlock. In Section 3 we gave a bound on the time between the occurrence of a deadlock and the declaration of a deadlock.

Thus far, three features of the algorithm have been described. They are (1) the explicit capture of deadlock context, (2) the diffusion of probe values, and (3) sending the probes together with the request. A probe is a compound message containing the context identification and a numerical value. However, two problems may arise.

The first problem is that even when a knot exists, the initiator may not be included in it, so that the initiator does not receive all its probes back. Figure 2(a) illustrates the situation where deadlock occurs among a subset of the dependents of process 1 and the initiator never gets the probes because it is outside the knot. We therefore require *every* process that is unable to service an incoming request to initiate its *own* probe with its *own* context as it forwards the request and incoming probe to its dependents. These probes are contained in a *probe list*. That is, each process forwards a compound message comprising the request and a probe list containing both the incoming probes with value modified (as explained in Section 2.3), and its own probe with value 1. With this feature, some process in the knot is always able to detect and declare a deadlock.

In Figure 2(b), process 2 initiates its own probe and is able to declare a communication deadlock. Since every process appends its own probe to the incoming probe, every incoming probe contains the *ancestors* of the receiving process. This is a useful piece of information for detecting cycles, as described below.

The second problem is that there may exist *recurrent* cycles so that some processes never get back the sum total of their probe values in finite time, compromising the correctness of the algorithm. Therefore a process should not repeatedly forward probes to dependents that have already returned them. By checking the list of ancestor processes in the incoming probe, a process can decide whether to forward any further. The details are described in Section 2.6.

Figure 3(a) is an example with a recurrent cycle involving processes 2, 5 and 6. As process 6 forwards the request to processes 2 and 1, process 1 only gets back portion of the probe, the other portion being circulated in the cycle 2,5,6. If process 1 continues to forward the returned probes, process 6 will send back an even smaller value. Thus process 1 never receives its sum totality of probe values, but a sequence of diminishing values.

2.5 Deadlock resolution

Deadlock resolution is often neglected [1, 9] or is not handled properly [14, 17] in deadlock detection algorithms. There is often a trade-off between the volume of information exchanged during the deadlock detection phase and the amount of time needed to resolve a deadlock once it is detected [19]. Long messages during the detection phase permit quick resolution of deadlocks [11] while short messages may require extra computation to resolve a deadlock later [17]. Since we are targeting our algorithm for soft real-time systems, we adopt the former approach.

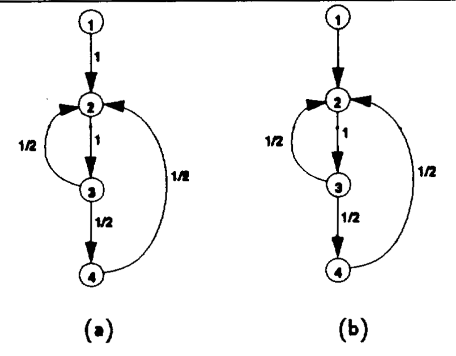

Figure 2: An example of an initiator (process 1) that is not included in the knot it has induced. In (a), the edges are labeled with the probe values under process 1's context. Note that process 1 is unable to detect the deadlock as no probes return. In (b), the edges are labeled with probe values under process 2's context and it is able to detect the deadlock.

We assume that there is a system-wide function, RESOLVE, that is commutative and associative. Given a set of processes involved in deadlock, this function returns a process whose outstanding request is to be aborted, thus ending the deadlock. In its simplest form, RESOLVE may simply pick the process with the lowest priority, though more complicated schemes are clearly feasible.

Thus far, a probe has contained a context ID and a numerical value. We add a third component, *abort_id*, to the probe, indicating the process that is to be aborted in the case of a deadlock. It is computed as follows: Whenever a probe list is propagated from process to process, the *abort_id* component in each probe in the probe list is simultaneously updated by applying RESOLVE on the *abort_id* and the receiving process's ID. As the request gets forwarded from process to process, the function is applied to all the processes in the WFG.

2.6 Formal description

This section presents the algorithm formally. The following are some features of the algorithm presented so far:

- A *probe* contains three components: the context ID, a numerical value, and the *abort_id*.

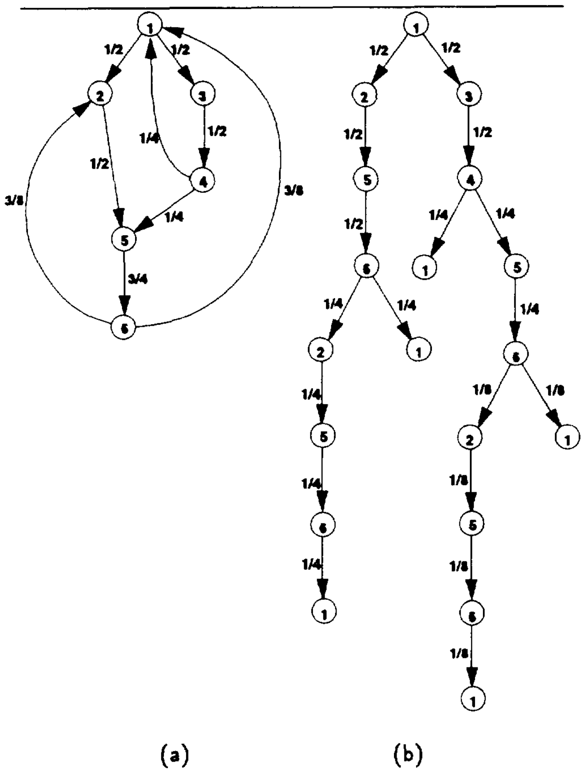

Figure 3: A WFG with recurrent cycles. There are two embedded cycles, {2,5,6} and {1,3,4}, within the knot in (a). (b) shows the tree of process 1 and how such cycles are handled by the path $6-2-5-6-1$.

- A *probe list* is a list of probes, each belonging to a different context, i.e., initiated by a different process. Note the distinction between a probe and a probe list.
- Every process forwards an incoming request together with the probe list to other processes only if it is unable to service the request.

Figure 4 presents a decision-tree version of the algorithm executed by each process of the WFG. Assume a probe list is received by a process p. If p is already blocked, its mailbox process takes over, and checks the probe list to determine whether p's context occurs in it (A). The two subtrees rooted at A and B in the figure correspond to the two outcomes. Consider the case when the probe list does not contain p's own probe (B). If p is active (E), and is able to satisfy the request, it does so and the algorithm halts, as do all forms of forwarding. Otherwise if p active (L) but is unable to satisfy the request, it updates all the probes in the probe list by reducing probe values and changing the *abort_id* if necessary (F). In addition, p creates a new probe under its own context and forwards it together with the probe list. In the case when p is blocked (F), it performs the same activities as in (F), i.e., updates probe lists and forwards request.

Now consider the upper subtree rooted at A, corresponding to the case when the received probe list contains p's own probe. First, p totals up the values received so far from its own context and checks whether this total value is equal to its original probe value 1. If so (C), a knot is detected and a deadlock resolution procedure is initiated. In particular, since the probe also contains the ID of the process to be aborted, a check is made to see if the process has already been aborted (H). If not, an abort message is sent to abort the process (G). If the probe values sum to less than 1, we have merely detected a cycle (D). Now we must decide which recipient processes to forward to again. We update incoming probes and forward only to those processes whose probes do not appear in the received probe list (J), that is, to the processes not forwarded to already. If all recipient processes appear in the received probe list, the current process removes the probes from the probe list individually and returns each of them directly to their initiators (I). This form of forwarding is called the *return step*.

Thus far, we have not considered what happens when a currently blocked process p receives a reply granting its request from one or more of its dependents. If so, p should inform all of its other dependents to stop all forwarding since it has already received a reply. These dependents in turn inform their dependents to abort all forwardings. This halts all unnecessary waiting and prevents a false deadlock from ever being declared.

The algorithm is presented in Figure 5. There are four procedures. Procedure PROBE-HANDLER() implements the decision tree described above with a modification and calls procedures CREATE-PROBE() and UPDATE-VALUE(). At point B in the decision tree, a check is made to determine if process p is active. If procedure PROBE-HANDLER() were to be executed by a process, this check could only be made before the procedure is executed. So if p is active, it executes the procedure. Otherwise it's mailbox thread takes over and execute the procedure. Thus the algorithm presented in the procedure

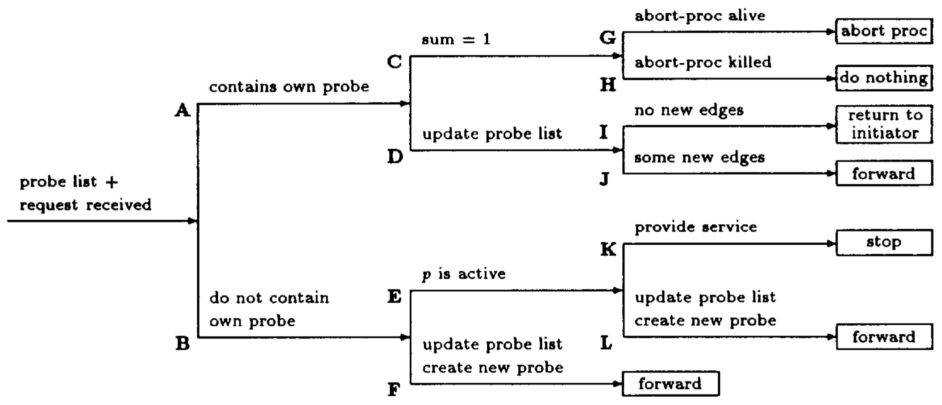

Figure 4: Decision tree version of algorithm ODDR.

does not perform the process status check. Procedure INIT-TRANS() is invoked whenever a process starts a new request. It creates a *transaction* with the new context and sends it together with the request to those processes that may service the request. Acknowledgements and other replies do not complete the transaction. In procedure CREATE-PROBE(), the new probe is initialized with the three components: *context_id* identifying the initiator of the request, *abort_id* identifying the process to be aborted in the case of a deadlock and the *probe_value*, which is initially taken to be 1. The variables used are described below:

- *ProbeList* is a list of ordered probes. n is the number of processes in *ProbeList*.
- *MyId* is a concatenation of the machine's identification and a process's identification.
- *out_degree* is a count of the number of processes the initiator intends to send to.
- *sum* keeps track of the sum total of the probe values a process will receive after it has initiator its own probe. It is initialized to 0.

3 Proof of correctness

Past experience [19] has shown the difficulty of designing correct distributed deadlock detection algorithms. Intuitive and informal arguments have proved to be highly unreliable. In this section, we present a graph-theoretic proof of the correctness of our algorithm. A deadlock detection algorithm is correct if it satisfies two criteria [19, 20]:

1. It detects all existing deadlocks in finite time.
2. It should not report false deadlocks.

We first define some terms and notations that will be used throughout the discussion.

Definition 3.1 (Waits-For) *$u \rightarrow v$ denotes that u is waiting for a reply from process v. $u \xrightarrow{\star} v$ denotes that process u is transitively waiting for process v.*

Process u can be waiting for process v only if it has sent v a message containing both the request and probes and has not received a reply from v. v is a *dependent* process of u. The Waits-For relationship is transitive. It is reflexive and symmetric only if there is a cycle containing both u and v.

Definition 3.2 (Communication Deadlock) *Given a distributed system represented as a directed graph $S = (V, E)$, where V and E are the set of sites and requests respectively, a communication deadlock is a knot in the Waits-For graph, i.e., a subgraph $D = (V_D, E_D)$, where $V_D \subseteq V$ and $E_D \subseteq E$ such that:*

1. *(Reachability) for all $u, v \in V_D, u \xrightarrow{\star} v$, and*
2. *(Reachability-Closure) for all $u \in V_D$, there does not exist any $w \in V - V_D$ such that $u \xrightarrow{\star} w$.*

```
ProbeList = {probe_1, probe_2, ..., probe_n};
probe ← record
      ContextId : integer;
      AbortId : integer;
      value : real;
end;

procedure CREATE-PROBE();
begin
      NewProbe.ContextId ← MyId;
      NewProbe.AbortId ← nil;
      NewProbe.value ← 1/out_degree;
      sum ← 0;
end;

procedure INIT-TRANS(Request);
begin
      OutNodes ← set of dependent processes;
      NewProbe ← CREATE-PROBE();
      ProbeList ← ProbeList ∪ {NewProbe};
      for each process in OutNodes do
            SEND(Request, ProbeList);
      endfor;
end;

procedure UPDATE-VALUE(ProbeList);
begin
      for each probe_i in ProbeList do
            probe.AbortId ← RESOLVE(MyId, probe.AbortId);
            probe.value ← probe.value/out_degree;
      endfor;
end;
```

Figure 5: Algorithm ODDR for on-line deadlock detection and resolution.

```
procedure PROBE-HANDLER(Request, ProbeList);
begin
      if (probe_i.ContextId = MyId) then
            sum ← sum + probe_i.value;
            probe_i.value ← nil;
            if (sum = 1) then
                  SEND(Abort-Signal, probe_i.AbortId);
            else
                  UPDATE-VALUE(ProbeList);
                  OutNodes ← set of dependent processes;
                  S ← set of OutNodes already in ProbeList;
                  OutNodes ← OutNodes - S;
                  if (OutNodes = empty)
                        strip ProbeList and forward;
                  else
                        for each process in OutNodes do
                              SEND (Request, ProbeList);
                        endfor;
            endif;
      else
            if (can satisfy request)
                  DO-REQUEST(Request);
            else
                  UPDATE-VALUE(ProbeList);
                  NewProbe ← CREATE-PROBE();
                  ProbeList ← ProbeList ∪ {NewProbe};
                  OutNodes ← set of dependent processes;
                  for each process in OutNodes do
                        SEND (Request, ProbeList);
                  endfor;
            endif;
end;
```

Figure 5: *continuation.*

Therefore a communication deadlock contains a knot such that every node is reachable from every other node and there are no edges out of the knot. By this definition, which also appears in [21, 15], a deadlock does not include nodes not in a knot, but which may be blocked on some nodes in a knot. An example is process 1 in Figure 2(a). The following lemma is needed:

Lemma 3.1 *Any probe initiated during the formation of knot K ultimately reaches all nodes in K.*

Proof: We first observe that each process may initiate only one probe during the formation of a knot, for it blocks immediately afterwards. Let u be a process that creates and forwards its probe to its immediate dependents, having appended it to the incoming probe list. Every recipient (active or blocked) of a probe list, in turn, forwards it to those of its dependents not already in the probe list. Thus, if $u \overset{*}{\rightarrow} v$, then v ultimately receives u's probe. If a knot K forms, Definition 3.2 guarantees that $u \overset{*}{\rightarrow} w$ for every $w \in K$. Thus, every process in the knot receives u's probe. ∎

Theorem 3.1 *The algorithm ODDR detects communication deadlocks.*

Proof: We express the theorem as the following statement: There is some process that receives back probe values summing up to 1 if and only if there is a communication deadlock. We shall show both necessary and sufficient conditions.

Let v be a node that receives back probe values summing to 1. If v were not in a knot, there exists a node u such that $v \overset{*}{\rightarrow} u$ but $u \overset{*}{\not\rightarrow} v$. The first condition implies that u receives some fraction f_u of the original probe values from v, but the second condition implies that v never receives back f_u from u. Thus the returned values could not have summed up to 1, a contradiction.

Conversely, suppose a knot exists, and let v be a process in this knot. The propagation of probe values originating from v defines a spanning tree rooted at v, with v at all leaf positions as well (Lemma 3.1). All propagation paths for probes originating at v end at v, so v receives back probe values adding up to 1. ∎

We next present the second part of the correctness proof. Communication deadlocks present a more difficult problem than do resource deadlocks because they may involve several cycles. An edge may be shared by multiple cycles, so that deleting the edge will break all the cycles. However, since the search for cycles may be performed independently by individual processes, the deadlock detection algorithm may not be aware of the deleted cycles, resulting in the reporting of false deadlocks. Therefore false deadlocks occur when deadlock detection and resolution is performed *simultaneously* by *independent* processes, each working with a past snapshot of the global state. We show in the following that algorithm ODDR does not report false deadlocks.

Theorem 3.2 *The algorithm ODDR does not detect false communication deadlocks.*

Proof: We show that every process in the knot identifies the same victim process to abort to resolve the deadlocks. False deadlocks are avoided by checking this victim process. By Lemma 3.1, the probes initiated by two nodes in the same knot define spanning traversals T_1 and T_2 in the knot. Thus the function RESOLVE sees *all* the nodes in the knot whether it is applied at the nodes of T_1 or those of T_2. Since RESOLVE is commutative and associative, it returns the same *abort_id* for T_1 and T_2. ∎

4 Complexity analysis

We evaluate the algorithm in terms of the number of messages, message size, local computation cost, and the length of time for which the probes circulate in the WFG before a deadlock is declared. As the probes are embedded in the normal request message, there are no separate messages exchanged for deadlock detection and resolution. However a tradeoff shows up directly in the message size. In fact, since the overall message size is extended by the probes, the message size is directly proportional to the number of probes in the probe list.

Theorem 4.1 *In the worst case, algorithm ODDR places $|E_D|$ probe lists in circulation at any time, where E_D is the set of edges in the WFG of the deadlock.*

Proof: Whenever a new probe list is received, a process introduces its own probe and forwards it together with the probe list to each of its dependents. Since each forwarding to a dependent corresponds to an edge in the WFG, there is a maximum of $|E_D|$ probe lists in circulation at any time. ∎

As deadlock detection and resolution are performed together, the number of messages for resolution up to the point of detection is also bounded by $|E|$. When a process detects a deadlock, it performs a multicast to the processes to be aborted, as indicated by the returned probes. This number is bounded by $|V_D|$. Therefore the message complexity is still the same, as expressed by the following corollary:

Corollary 4.1 *The upper bound on the number of probe lists to perform deadlock resolution by algorithm ODDR is $|E_D|$, where E_D is the set of edges in the WFG.*

Next, we estimate the length of time the probes lists are in circulation before their circulation is halted by the detection of a deadlock. Since probes are introduced as the requests are made, the algorithm may never terminate if no deadlock is formed. As soon as a deadlock is formed, the probes are already on their way to inform some processes of the deadlock.

Theorem 4.2 ((On-Line Property)) *There is a $O(|V_D|)$ delay between the occurrence of a communication deadlock and the declaration of a deadlock in algorithm ODDR.*

Proof: In the worst case, a probe may have to traverse the length of a cycle before returning. Since the length is in the worst case bounded by $O(|V_D|)$, the theorem follows. ∎

Upon receipt of a request and probe list, each node performs $O(|V_D|)$ divisions in the worst case because there are at most $|V_D| - 1$ probes in the probe list. As mentioned in Section 2.4, multiplication may be performed instead of division with no loss of precision. The computation cost does not slow down the local system because (1) it is relatively inexpensive compared to communication cost, (2) a process that is blocked after forwarding the request and probe list will not perform any more arithmetic under the same context as the request it is waiting.

5 Conclusions

We have considered the detection and resolution in communication deadlocks from the point of view of efficiency and timeliness. To the best of our knowledge, no other algorithm has adopted an integrated approach that achieves both algorithm efficiency and on-line performance.

Algorithm ODDR is clearly a fully distributed algorithm. There is no centralized detector of deadlocks, and no need for any election of deadlock detector nodes as required in [16]. Every process that is part of the same waits-for graph exercises the same protocol. Thus every process has an equal chance of detecting deadlocks. The algorithm will never terminate if there is no communication deadlock. But whenever it terminates, it not only declares a deadlock but also names the set of processes to be aborted in order to resolve the deadlock. The algorithm ODDR possesses the following properties:

1. The algorithm does not require explicit rounds of probe signals as do most probe-based algorithms. Thus it not does not incur additional message traffic. In Natarajan's algorithm [16], probe signals are sent periodically, regardless of whether deadlocks have occurred. This generates lots of messages. Haas and Mohan [9] presented an algorithm that requires the transmission of exponentially many messages for deadlock detection in the worst case, which is unacceptable in practical applications. In our algorithm, deadlock detection is part of the process's communication pattern. Hence the algorithm is cheap to execute.

2. Deadlock resolution is often cumbersome in distributed deadlock detection algorithms because several sites may not be aware of other sites and/or processes involved in the deadlock [19]. The ODDR algorithm avoids this problem. The probe lists that return to the initiators indicate the same *abort_ids* because they have traversed the entire knot.

   The function RESOLVE, which is used to select a process in a deadlock, subsumes all previous priority-based detection algorithms [5, 3, 4, 7, 18, 20]. In these algorithms, all transactions are assigned a priority under the assumption that there exists an *a priori* scale against which a transaction is marked.

3. A minor weakness of the ODDR algorithm is the trade-off of the message size for minimal deadlock duration. However, this is not a serious weakness since the additional information is simply piggybacked on to request messages. The overhead incurred is several orders of magnitude less than that incurred when separate messages are being sent. The size of the request message is variable and depends on two factors:

   (a) The proximity of a transaction to the initiator of the detection process. Processes that are farther away receive request messages with more probes.

   (b) The number of processes in the knot. The larger the knot, the more probe lists there are in circulation.

It is intuitively clear that algorithm ODDR detects existing communication deadlocks. However it is also interesting to note how it handles false deadlocks. Past algorithms fail in this aspect because they detect and resolve individual cycles in the deadlock. Our algorithm is unique in that if a deadlock exists, every process that reports the deadlock agrees on the same process to be aborted. Thus false deadlocks have no adverse effects.

Acknowledgements

The authors would like to thank Wei-Lun Yeh for contributing ideas leading to the initial version of this algorithm.

References

[1] K. M. Chandy, J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," *Proceedings 1st ACM Symposuim on Principles of Distributed Computing*, pp. 157–164, 1982.

[2] I. Cidon, "An Efficient Distributed Knot Detection Algorithm," *IEEE Transactions on Software Engineering*, Vol. 15, No. 5, pp. 644–649, May 1989.

[3] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, D. Towsley, "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Transactions on Software Engineering*, Vol. 15, No. 1, pp. 10–17, Jan. 1989.

[4] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, D. Towsley, "Correction to 'A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution'," *IEEE Transactions on Software Engineering*, Vol. 15, No. 1, pp. 10–17, Jan. 1989.

[5] K. M. Chandy, J. Misra, L. M. Haas, "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2, pp. 144–156, May 1983.

[6] K. J. Compton, C. V. Ravishankar, "Mean Time to Deadlock in a Multiprocessing System," under preparation for the *Journal of the ACM*.

[7] A. K. Datta, S. Ghosh, "Deadlock Detection in Distributed Systems," *Proceedings 9th Annual International Phoenix Conference on Computers and Communications*, pp. 131–136, Mar. 1990.

[8] J. Gray, P. Homan, R. Obermarck, H. Korth, "A Straw-Man Analysis of the Probability of Waiting and Deadlocks in a Database System," *IBM Research Report*, RJ 3066, IBM San Jose Research Laboratory, Feb. 1981.

[9] L. M. Haas, C. Mohan, "A Distributed Deadlock Detection Algorithm for a Resource-Based System," *IBM Research Report*, RJ 3765, IBM San Jose Research Laboratory, 1983.

[10] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, pp. 666–677, Aug. 1978.

[11] S. S. Isloor, T. A. Marsland, "An Effective On-Line Deadlock Detection Technique for Distributed Database Management Systems," *Proceedings COMPSAC '78*, pp. 283–288, Nov. 1978.

[12] E. Knapp, "Deadlock Detection in Distributed Systems," *ACM Computing Surveys*, Vol. 19, No. 4, pp. 303–328, Dec. 1987.

[13] A. D. Kshemkalyani, M. Singhal, "Invariant-Based Verification of a Distributed Deadlock Detection Algorithm," *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, pp. 789–799, Aug. 1991.

[14] D. A. Menasce, R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Transactions on Software Engineering*, Vol. 5, No. 3, pp. 195–202, May 1979.

[15] J. Misra, K. M. Chandy, "A Distributed Graph Algorithm: Knot Detection," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, pp. 678–686, Oct. 1982.

[16] N. Natarajan, "A Distributed Scheme for Detecting Communication Deadlocks," *IEEE Transactions on Software Engineering*, Vol. 12, No. 4, pp. 531–537, Apr. 1986.

[17] R. Obermarck, "Distributed Deadlock Detection Algorithm," *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp. 187–208, Jun. 1982.

[18] M. K. Sinha, N. Natarajan, "A Priority Based Distributed Deadlock Detection Algorithm," *IEEE Transactions on Software Engineering*, Vol. 11, No. 1, pp. 67–81, Jan. 1985.

[19] M. Singhal, "Deadlock Detection in Distributed Systems," *Computer*, Vol. 22, No. 11, pp. 37–48, Nov. 1989.

[20] S. H. Wang, G. Vossen, "Towards Efficient Algorithms for Deadlock Detection and Resolution in Distributed Systems," *Proceedings 5th International Conference on Data Engineering*, pp. 287–294, Feb. 1989.

[21] B. E. Wójcik, Z. M. Wójcik, "Sufficient Condition for a Communication Deadlock and Distributed Deadlock Detection," *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, pp. 1587–1595, Dec. 1989.