분산처리 시스템하에서의 모든 교착상태 발견을 위한 알고리즘

이수정
인천교육대학교 컴퓨터교육과

요 약

분산처리 시스템하에서 교착 상태를 발견하기 위한 대부분의 분산 알고리즘은 probe라는 짧은 메시지를 사용하지만 발생되는 메시지의 일부분만이 사용될 뿐이다. 따라서 불필요한 probe들은 communication traffic을 심히 초래하는 결과를 낳는다. 본 논문에서는 이러한 결점을 보완하여 모든 probe들이 유효하게 사용되어지는 알고리즘을 제시한다. Wait-for-graph상의 모든 edge의 수를 e라고 하였을 때, 제시된 알고리즘은 graph 상의 모든 교착 상태를 $O(e)$ 메시지를 사용하여 발견한다.

Complete Deadlock Detection in a Distributed System

Soojung Lee
Inchon National University of Education, Dept. of Computer Education

ABSTRACT

In most of the distributed deadlock detection algorithms using messages called *probes*, only a portion of the generated messages are effectively used, and hence the wasted probes cause heavy communication traffic. In this paper, a distributed deadlock detection algorithm is proposed which can efficiently detect deadlocks making use of those *residue* probes. Our algorithm is complete in the sense that they detect not only those deadlocks in which the initiator is involved as most other algorithms do, but all the other deadlocks that are present anywhere in a connected wait-for-graph. To detect all the deadlocks, the algorithms known to be most efficient require $O(ne)$ messages, where e and n are the number of edges and nodes in the graph, respectively. The single execution of the presented algorithm can accomplish the same task with $O(e)$ messages.

1. Introduction

In a distributed system, processes send messages through a network to request resources and wait until these requests are granted. A deadlock occurs when a group of processes are blocked indefinitely from access to exclusive resources held by one another. The resource requesting state between processes in a distributed system is represented by a directed graph, known as the wait-for-graph(WFG), where each vertex represents a process and an arc is originated from a blocked process waiting for a resource to a process holding the resource. A process requesting more than one resource remains blocked until all the requests are granted. A blocked process cannot continue its computation until it acquires all resources that it has requested. Thus, finding a deadllock in this model corresponds to finding a cycle in the WFG.

Various distributed algorithms for detecting deadlocks have been developed in the past, as surveyed in [6,10]. Most of the early algorithms try to maintain an explicit form of the WFG at each site, a portion of which is passed along inter-site edges[8]. These algorithms are, however, likely to falsely declare a deadlock, i.e., declare a deadlock where none exists, because of time discrepancies of the gathered WFGs. Due to such drawbacks as well as varying sizes of probes, people turned their attention to another more elegant group of algorithms that use a message called *probe*[2,3,5,7.9]. A probe is passed along the edges of the WFG to detect deadlocks; when the probe comes back to its originator, a deadlock is declared. We refer to those algorithms using probes as *probing algorithms.*

The computational complexity of the algorithms has not been given much attention, let alone improving it. The current best probing algorithms, in the worst case, require $O(ne)$ messages to find all deadlocks, where n is the number of nodes and e is the number of edges in the WFG[2,3,9]. This is due to the fact that there must be at least one initiator for each deadlock and a prob egenerated by each initiator might explore all the edges.

Besides the high message or time complexity, most probing algorithms are faced with several problems. Only those deadlocks in which an initiator is involved can be detected, even if the probes are delivered to all the reachable nodes from the initiator. In some algorithm, a probe initiated by a node outside a cycle floats around the cycle until it is broken, thereby causing an exponential number of messages to be generated. The main factor responsible for these unnecessary probes is the insufficient condition of a cycle declaration; an initiator must receive its own probe back to know if it belongs to a cycle.

A false deadlock detection is also possible in probing algorithms due to independent detection and resolution of deadlocks between algorithm computations. Some deadlocks cannot be detected due to the incorrect conditions for updating the resource waiting status or for algorithm invocations.

In this paper, we present a distributed deadlock detection algorithm which solves the above problems. A new condition for deadlock declaration is developed by which all probes from an initiator are useful and contribute to finding deadlocks. A single execution of our algorithm detects deadlocks that exist anywhere in the WFG at the time of initiation, and no false deadlocks are declared by employing $O(e)$ time and probes whose size is bounded by only two message fields. This result does not vary with initiators but applies equally well no matter which node starts the algorithm.

The rest of this paper is organized as follows: Our system model is described in the next section,

followed by a detailed description of our algorithm with its proof of correctness in section 3. Section 4 concludes this paper by discussing the merits of our algorithm over those that were previously published in the literature.

2.. The System Model

When a request for resource is made to the corresponding resource manager, the manager sends a grant or a reject message to the requester depending on the resource availability and maintains the waiting process table. Upon receiving a rejection on any request made, the process is blocked waiting for the resource to be released. Otherwise, it continues its execution.

In this paper, we make the following assumptions. Messages are delivered from one node to another without any error in the order in which they are sent (FIFO channels), and they are assumed to arrive at the destination in a finite time without being lost. Each processor, which never fails, executes one or more processes, and processes do not fail unless they are killed by the system. Each process in our system model is assigned a unique identifier. When a process is blocked waiting for a prespecified time period (time-out), it may initiate the deadlock-detection activity.

If there exists a directed edge in the WFG from process i to process j, we call process $j(i)$ a *successor*(*predecessor*) of $i(j)$. It is assumed that a node is able to figure out its predecessors and successors through communication with the resource managers whose resources it is currently holding or waiting for. A process is called *running* if it has no outgoing edge in the WFG.

In order to execute our algorithm, we take a static view of a dynamic resource requesting and releasing system. When node p first participates in the algorithm, either by time-out or upon receiving the first probe, its predecessors and successors at the moment are recorded in the local data structures named *pred_p* and *succ_p*, respectively. Once constructed, the probes are transmitted only to those nodes in the data structures in the forward or backward direction. The only modification made to these variables after the construction is when the WFG changes due to p releasing the resource to its predecessor. That is, if p releases the resource requested by a predecessor q, q is removed from *pred_p* and p is also removed from *succ_q*. Except such reflection of WFG changes, these variables are used to update the probe receiving and sending status.

A false deadlock can occur when a probe is simply propagated to one of the stored successors without any checking of who has sent the probe. The following scenario illustrates this situation. Suppose that a WFG consists of a single path of processes, $(p\_1,p\_2,...,p\_k)$, where each p_i is a predecessor of $p\_\{i+1\}$ for $i=1,...,k-1$ and p_k is running. Also, suppose that p_1 initiates a probe, makes up its data structures as $pred\_\{p\_1\}=\{\}$ and $succ\_\{p\_1\}=\{p\_2\}$, and transmits the probe to p_2. Then all the processes p_i for $i=2,...,k-1$ would form their data structures as $pred\_\{p\_i\}=\{p\_\{i-1\}\}$ and $succ\_\{p\_i\}=\{p\_\{i+1\}\}$ and propagate the probe. Assume that the probe is in transit to p_k and that in the meantime p_k has finished execution on the resource and released it to $p\_\{k-1\}$. Right after, p_k requests another resource that is currently held by p_1. Now, when p_k receives the probe sent by $p\_\{k-1\}$, it forms its data structures as $pred\_\{p\_k\}=\{\}$ and $succ\_\{p\_k\}=\{p\_1\}$. Thus, p_k sends the probe back to p_1.

Assuming that a node declares a deadlock when its own probe returns to itself, the above scenario shows a possibility of false deadlock detection; this condition for deadlock detection is true for

most algorithms including ours even though ours is more general as will be seen in the next section. Our algorithm avoids the problem simply by requiring the receiver of a probe to check whether it has released the resource requested by the sender; that is, whether the sender is included in the *pred* of the receiver. Should an inconsistency be discovered, the probe would not be any longer transmitted by the receiver.

3. The Presented Algorithm

In this section an algorithm is presented which employ a distributed depth-first search procedure to detect deadlocks. The description of the algorithm begins with a discussion of how to detect deadlocks residing in the reachable set of an initiator. Then we extend this strategy to detect deadlocks that exist anywhere in the system.

3.1 Algorithm Description

Finding All Reachable Deadlocks

The algorithm distinguishes various edge types to find cycles by appropriately managing the state information through a depth-first search. A node is assumed to be in NORMAL state initially. When a node takes part in the deadlock-detection activity, it changes its state from NORMAL to VISITED. The state is updated to FINISHED when a node completes its participation in the algorithm. During the operation of the algorithm, probes are passed in a depth-first search manner.

An initiator invokes its deadlock-detection activity by setting its state to VISITED and sending a SPAN probe to one of its successors intending to construct a depth-first search tree rooted at itself. When a node receives a SPAN for the first time while it is in NORMAL state and still holding the resource requested by the sender of the SPAN, it accepts the sender as its *father* in the tree, sets its state to VISITED, and passes the SPAN to one of its successors, if any. If the node has already released the resource, it sends back a SPAN_TERM probe to acknowledge the sender of the SPAN. Upon receiving a SPAN_TERM, a node sends the SPAN to another of its successors, if any. When the node visits all the successors or has no successors, it sets the state to FINISHED and notifies its father of the completion of the subtree construction through a SPAN_TERM.

When a node in FINISHED state receives a SPAN, it recognizes that there exists a forward-edge or a cross-edge between itself and the sender of the SPAN, because it has completed spanning all of its successors and thus cannot receive a SPAN from any of them. In this case, it simply replies with a SPAN_TERM to the sender of the SPAN as a signal for the latter to proceed with another successor. If a node receives a SPAN when it is in VISITED state, a back-edge (cycle) is found, because only one successor is searched at a time; that is, since the node has not completed probing its successors, its ancestors or siblings cannot send any other SPAN until it reports with a SPAN_TERM to its father. When a back-edge is found, the node declares the detection of a deadlock, and sends back a SPAN_TERM to the sender of the SPAN to allow the latter to probe another successor.

A SPAN_TERM carries a *type* field to inform its receiver whether the edge along which a probe is transmitted is a tree edge or not. A SPAN_TERM of REMOVE type indicates that the corresponding SPAN ends in discovering a back, cross, or forward edge and enables the receiver of the SPAN_TERM to remove the edge. In other cases, a SPAN_TERM carries a SUCCESS type.

This type information is used to build a tree.

The tree construction performed on the reachable set of the initiator terminates when the initiator explored all of its successors and has received a reply probe SPAN_TERM from each of them. Notice that the message and time complexities of one such deadlock-detection activity is *O(e)*. Formal specifications of the algorithm for node *i* are provided below. Descriptions on receiving the probes other than SPAN or SPAN_TERM in proc_expand_tree procedure are given later in this subsection.

Data Structures at node *i* (Initial values are inside the parentheses)

- *pred_i* : keeps track of the predecessors that have not sent any SPAN to *i*. (list of predecessors)
- *succ_i* : maintains the successors that have not received any SPAN from *i*. (list of successors)
- *state_i* : represents a progression status in building a tree. (NORMAL)
- *sons_i* : a set of children in the constructed tree. ({})
- *father_i* : father node in the constructed tree.(0)

Message Formats

- *SPAN(j)* : a probe to expand a tree sent by node *j*.
- *SPAN_TERM(type,j)* : a reporting probe of a subtree completion sent by node *j*. *type* can be either SUCCESS or REMOVE.

When an initiator *i* invokes the algorithm :
proc_tree_node(0);

On receiving a SPAN(*j*) :
/* Check if the resource has been already released */
if *j* is not in *pred_i* **and** *state_i*=NORMAL
then begin
  send SPAN_TERM(REMOVE,*i*) to *j*; return;
end
pred_i := *pred_i*-{*j*};
case *state_i* **begin**
  NORMAL : proc_tree_node(*j*);
  VISITED :
    cycle_detected_i := 1;
end
if *state_i*=VISITED **or** FINISHED **then**
    send SPAN_TERM(REMOVE,*i*) to *j*;

On a receiving SPAN_TERM(*type,j*) :
if *type*=SUCCESS **then** *sons_i* := *sons_i*U{*j*};
proc_expand_tree;

procedure proc_tree_node(*j*)
begin
state_i := VISITED; *cycle_detected_i* := 0;
father_i := *j*; proc_expand_tree;
end

procedure proc_expand_tree
begin
if *succ_i* != {} **then begin**
   send SPAN(*i*) to a node, say *j*, in *succ_i*;
   succ_i := *succ_i* - {*j*};
end
else begin
   state_i := FINISHED;
   if *father_i* != *0* **then**
      send SPAN_TERM(SUCCESS,*i*) to
      father_i;
   else execute the procedure on receiving a
      SEARCH;
end
end

Finding All Deadlocks In The System

In order to find deadlocks outside the tree built

as above, we exploit the same scheme as above on the outsiders (unexplored nodes);namely, detecting deadlocks in them by constructing one or more trees. At the end, this idea results in a forest of trees consisting of the nodes in the WFG while detecting all deadlocks in them.

Searching the outsiders is performed by one tree node after another starting from the root node. Upon recognizing the completion of the tree construction, the root searches an unexplored predecessor, to which it sends a START probe, implying that the receiver needs to start the algorithm for finding all reachable deadlocks as a root of a new tree. A START probe is acknowledged by a COMPLETE probe which is sent by this second root when it completes its tree expansion and also exploring any outsiders from its own tree recursively. On receiving a COMPLETE, a node looks for another unexplored predecessor and sends another START.

When a node finds no more such predecessors, it sends a SEARCH probe to one of its sons to perform the same; that is, transmitting a START to an unvisited predecessor and waiting for a COMPLETE from it. Similar to the way SPAN probes are propagated, the SEARCH is transmitted by tree nodes after it finishes the above procedure.

Upon finding no more sons to send a SEARCH to, a node reports its father with a SEARCH_TERM probe indicating that all the nodes connected to the subtree rooted at itself have finished exploration. If the father still has a son to which it has not transmitted a SEARCH yet, it propagates the SEARCH to the son as before. One more variable for node i is required to implement this scheme.

Data Structure at node i (An initial value is inside the parentheses)

- *boss_i* : the node identifier sending a START probe to i. (0)

Message Formats

- SEARCH : a probe for the receiver to look for predecessors outside the tree.
- SEARCH_TERM : a reply probe to a SEARCH.
- START(j) : sent by node j for the receiver to start the algorithm.
- COMPLETE : a reply to a START sent upon completion of the tree construction.

On receiving a START(j) :
if *state_i*=NORMAL **then begin**
  boss_i := *j*; *succ_i* := *succ_i*-{*j*};
  proc_tree_node(0);
end
else send COMPLETE to *j*;

On receiving a SEARCH, SEARCH_TERM, or COMPLETE :
if *pred_i* != {} **then begin**
  send START(*i*) to a node, say *j*, in *pred_i*;
  pred_i := *pred_i* - {*j*};
end
else if *sons_i*!={} **then begin**
  send SEARCH to a node, say *j*, in *sons_i*;
  sons_i := *sons_i*-{*j*};
end
else if *father_i*!= 0 **then**
  send SEARCH_TERM to *father_i*;
else if *boss_i*!= 0 **then**
  send COMPLETE to *boss_i*;
else terminate the algorithm;

3.2 Correctness

A path is denoted by a concatenation of edge

notations with no repetition of node identifiers.

For instance, a path *(i,j,k)* indicates the presence of two directed edges, *(i,j)* and *(j,k)*. Note that if the last node identifier is the same as the first one in a path notation, it indicates a cycle.

Theorem 1 *If there exists a cycle of resource requests at the time of invocation of the algorithm, it is detected by only one node.*

Proof Suppose that a cycle, $C=(n\_1,n\_2,...,n\_k,n\_1)$, is formed at or before the initiation time. There must be a node, say n_i, $i=1,...,k$, that receives a SPAN first among the nodes in the C. Then n_i would modify its state to VISITED and propagate the SPAN to the next node, n_j, for $j=i+1 \bmod k$, in the C, since the edge to n_j has lasted since the initiation time. Upon receiving the SPAN, n_j would perform similarly, and so would all the other nodes in the C. As a result, the SPAN would traverse all nodes in the C. When the SPAN arrives at n_i again, the condition for cycle detection becomes satisfied. QED

Theorem 2 *If a node declares a cycle detection, that node is in fact contained in the cycle.*

Proof Assume that a SPAN traversed a sequence of nodes, $n\_1,n\_2,...,n\_k,n\_1$ and made n_1 declare a cycle. Suppose that n_i, $i=1,...,k$, became active after sending the SPAN to its successor n_j in the sequence, $j=i+1 \bmod k$. Then n_j must have released the resource to n_i at time t before the latter became active. Since under our assumed model a node cannot grant a resource held by itself to any until it acquires all the resources that it has requested, all successors of n_j must have also granted the resources to n_j some time before t. From the above argument and the assumption that the SPAN went through all the nodes in sequence, it can be inferred that every node has received and passed the SPAN before it released the resource to its predecessor in the sequence, since otherwise, the SPAN should not be delivered any more according to our algorithm.

For any two adjacent nodes in the sequence, n_i and its successor n_j, satisfying the following,
At $t\_\{i1\}$, n_i received and passed the SPAN.
At $t\_\{i2\}$, n_i released the resource.
At $t\_\{j1\}$, n_j received and passed the SPAN.
At $t\_\{j2\}$, n_j released the resource.
we can conclude the timing relations such as
$t\_\{i1\} < t\_\{j1\} < t\_\{j2\} < t\_\{i2\}$.
Inductively, we obtain between n_1 and n_2,
$t\_\{11\} < t\_\{21\} < t\_\{22\} < t\_\{12\}$,
between n_2 and n_3, $t\_\{21\} < t\_\{31\} < t\_\{32\} < t\_\{22\}$,
. . . .
between $n\_\{k-1\}$ and n_k, $t\_\{(k-1)1\} < t\_\{k1\} < t\_\{k2\} < t\_\{(k-1)2\}$,
and between n_k and n_1, $t\_\{k1\} < t\_\{11\} < t\_\{12\} < t\_\{k2\}$,
which results in
$t\_\{11\}<t\_\{21\}< . . . <t\_\{(k-1)1\}<t\_\{k1\}<t\_\{11\} <t\_\{12\} <t\_\{k2\}<t\_\{(k-1)2\}< . . .<t\_\{22\}<t\_\{12\}$.
Therefore, $t\_\{11\}=t\_\{21\}= . . . =t\_\{k1\}$, $t\_\{12\}=t\_\{22\}= . . . =t\_\{k2\}$,
and $t\_\{11\}<t\_\{12\}$.
This implies that every node passed the SPAN at $t\_\{11\}$ simultaneously and that every node released the resource at the same time $t\_\{12\}$. Hence n_1 must have detected a cycle at $t\_\{11\}$. However, when it did, no node in the cycle could be running because no node released resources until $t\_\{12\}>t\_\{11\}$.

3.3 Complexity Analysis

Lemma 1 *Assume that for a particular node i, if $i \in pred\_j$, for any node j, then $j \in succ\_i$.*

Then i can receive at most one START message.

Proof Suppose that there are k nodes, $p\_1, p\_2, \ldots, p\_k$, with $i \in pred\_\{p\_j\}$ for $j=1,\ldots,k$ and that each p_j sends a START to I. Due to the assumption, $succ\_i$ includes all the p_j. Let p_1 be the first node to send the START. According to our algorithm, the time when the next node transmits a START to i must be after i replies with a COMPLETE to p_1 upon completion of spanning its tree. However, during the process of constructing a tree rooted at i, a SPAN should be sent to all the successors in $succ\_i$ including each p_j. Upon receiving the SPAN, each p_j is supposed to delete i from its *pred* list. Hence, any other predecessor cannot send a START to I. QED

By Lemma 1, the number of START messages cannot exceed n under the given assumption. If the assumption is violated, the responsible edge must have disappeared before node i is involved in the algorithm or it must have formed after. The number of STARTs can be deduced by observing that our algorithm allows such message to be transmitted through an edge provided that edge has never delivered a SPAN before. Hence, the total number of START and SPAN messages cannot exceed e. Their corresponding replies, COMPLETE and SPAN_TERM would produce the same amount.

At most one SEARCH and SEARCH_TERM can be sent along each tree edge, which results in a total of $2(n-t)$, where t is the number of trees built by the algorithm. Therefore, the algorithm detects all deadlocks spending $2(e+n-t)$ number of messages and as many time units since at most one message is delivered at a time.

4. Discussion

In this paper, we concentrated on the problem of finding deadlocks in the AND model[6] in a distributed system. Almost all of the algorithms using probes in the literature are based on such condition of finding a cycle that the originator of the probe must receive its probe back. This condition brings about some restrictions on the qualification of a node for detecting a cycle; it should not only invoke the algorithm but belong to the cycle. Therefore, the elapsed time to find all cycles in the system can be hardly figured out, since at least one node per cycle has to initiate the algorithm and the initiation time is mostly determined either periodically or by time-out. Furthermore, we can view the independent executions of the algorithm by distinct initiators as another shortcoming since the number of generated probes is left uncontrolled, which often ends in exponentially many probes.

We presented and proved an efficient deadlock detection algorithm which does not lean on the state-of-the-art technique for detecting cycles, but offers a new paradigm. A single invocation of the presented algorithm detects all the deadlocks present in the system. The greatest advantage of our algorithm is its thorough usage of all the transmitted probes so that they are not wasted as in most probing algorithms. Even though the presented algorithm also does not consider handling of multiple initiations, it does not produce an exponential number of messages as some published algorithms do.

The simplest way to reduce the number of generated probes by several independent initiations is to use priority information carried along each probe. That is, an initiation of the lower priority gives up to that of the higher priority. However, it is enough to invoke our algorithm only once to detect all deadlocks in the system. Therefore, with

a timing gap larger than the duration of the algorithm between the consecutive invocations, our algorithm should exhibit the best performance. This would bring another favorable property for deadlock resolution that each cycle is detected by only one node.

References

[1] A. V. Aho and J. E. Hopcroft and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[2] G. Bracha and S. Toueg, "A distributed algorithm for generalized deadlock detection", ACM Symposium on Principles of Distributed Computing, pp.285-301, Aug. 1984.

[3] K. M. Chandy and J. Misra and L. M. Haas, "Distributed Deadlock Detection" ACM Trans. Computer Syst., Vol. 1, pp. 144-156, May 1983.

[4] A. N. Choudhary, "Cost of Distributed Deadlock Detection: A Performance Study", Proc. 6th Int'l Conf. Data Eng.,pp. 174-181, Feb. 1990.

[5] A. N. Choudhary and W. H. Kohler and J. A. Stankovic and D. Towsley,"A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution", IEEE Trans. Softw. Eng., Vol. 15, pp. 10-17, Jan. 1989.

[6] E. Knapp, "Deadlock Detection in Distributed Databases", ACM Computing Surv., Vol. 4, pp. 303-328, Dec. 1987.

[7] A. D. Kshemkalyani and M. Singhal,"Distributed Detection of Generalized Deadlocks", Proc. 17th Int'l Conf. Distributed Computing Systems, pp. 553-560, May 1997.

[8] R. Obermarck, "Distributed Deadlock Detection Algorithm", ACM Trans. Database Syst., Vol. 7, pp. 187-208, June 1982.

[9] M. Roesler and W. A. Burkhard,"Resolution of Deadlocks in Object-oriented Distributed Systems",IEEE Trans. Computers, Vol. 38, pp. 1212-1224, Aug. 1989.

[10] M. Singhal, "Deadlock Detection in Distributed Systems", IEEE Computer, Vol. 22, pp. 37-48, 1989.