Proceedings of the 5th WSEAS/IASME Int. Conf. on SYSTEMS THEORY and SCIENTIFIC COMPUTATION, Malta, September 15-17, 2005 (pp336-341)

A Concurrent Distributed Deadlock Detection/Resolution Algorithm for Distributed Systems

CHENG XIN, YANG XIAOZONG
School of Computer Science and Engineering
Harbin Institute of Technology
Harbin, Xidazhi street No.92 150001
CHINA

Abstract: - Numerous deadlock detection algorithms were proposed for distributed systems, but most of them assumed the static wait-for graph (WFG), which is inconsistent with the dynamic application environment of distributed systems, in fact they can not run concurrently. A novel instance of diffusion-computation algorithms is proposed in this paper to resolve the concurrency problem. In our algorithm, a Dynamic WFG (DWFG) is raised, where the blocked transactions creating or quitting is responded to the nodes joining or disappearing in DWFG. Three additional detection termination conditions are assigned for the concurrent running of the proposed algorithm: the being detected node is a leaf or quitted, or fault occurs in the system. By these methods the concurrent running detections can terminate eventually and the deadlock can be resolved correctly. The correctness of the proposed algorithm is proven. Performance evaluation shows the time and message complexity of our algorithm outperforms the existing algorithms under a static WFG.

Key-Words: - Distributed systems; Deadlock detection/resolution; Dynamic wait-for graph; Concurrent

1 Introduction

The distributed deadlock problem has been extensively studied and numerous deadlock detection algorithms were developed. Based on the deadlock detection technique, the existing algorithms can be classified into four categories: *path pushing*, *edge chasing*, *diffusion computation* and *globe state detection* [1]. Usually deadlock detection approach responds the blocked system to a WFG firstly, afterwards the detection messages, such as *probe* [2], *label* [3], *token* [4] and *deadlock detection agent* [5] were propagated along the edges of WFG to find the closed deadlock cycles or knots, finally at least one victim in a cycle or knot is aborted to resolve the infinitely waiting state of the underlying system.

However, most of the existing distributed deadlock detection algorithms assume explicitly or implicitly that the WFG is static [6~13], but in real systems, the structure of WFG may be changed in such situations: 1) the undetermined massage passing delay makes the blocked transactions unblocking latterly; 2) the transactions executions is stopped by node failures; 3) new blocked transactions are created before the deadlock resolution finish. Without these consideration, many algorithms were proved to be incorrect: the detection messages were discarded falsely or lost their detect objects so that the algorithms ran into disorder.

This paper presents a DWFG based distributed deadlock detection algorithm, which is a variation of diffusion computation. In our algorithm, the blocked transactions creating or quitting is responded to the nodes joining or disappearing in DWFG. Every transaction has a *resource request satisfaction predication* $f(n, T)$, only if it is true can the transaction be committed. The notion *Virtual Victim* is invented to reflect the dynamic shift in DWFG: besides the active victims produced by the normal deadlock detections, any quitted transaction in above situation 1) and 2) is stipulated as the virtual victim of a deadlock detection, which will makes the algorithm terminated passively. Therefore, not only the single running detection but also the concurrent detections can terminate eventually. In the case of 3) above, we prove that they are trivial for a detection.

The rest of this paper is organized as follows: Section 2 introduces the general distributed system model, provides $f(n, T)$ and illustrates how it can be used to express the resource request models. Section 3 presents the new deadlock detection and resolution algorithm. Section 4 discusses the deadlock detection correctness criteria of our algorithm in DWFG and proves it. Section 5 evaluates the message and time complexities of the new algorithm in a static environment. Section 6 concludes our works.

2 The Distributed System Model

Without loss of generality, a distributed system can be viewed as consisting of a collection of nodes on which transactions and resources are residing; each node has a manager, i.e., *transaction manager* (TM)

and *resource manager* (RM). Every transaction or resource has a unique *id* for the ordering purpose. After a transaction quitted, it reenters the system with a new *id* to keep the system concurrency. TM is responsible for its ever managed transactions. In order to simplify the presentation, we assume that each TM and RM controls one transaction and resource respectively, so we will often use synonymously the term transaction for TM and resource for RM.

No shared storage exists in the system model. TMs and RMs communicate via message passing. We assume the messages arriving at object in their sending order and the communication channel is error-free, note the executions in local node may be fail. If these assumptions can not be held, some fault tolerance schemes should be applied.

A resource can be accessed exclusively. RM receives the requests and schedules the requesters in a FCFS table, only the requester in the top of the table can be granted. If a request is released or canceled, the RM cancels the requester from table and updates it.

A two-phase locking protocol is assumed for the transaction execution. In the first phase, TMs send all resources *requests* on behalf of the transactions; RMs reply with *grant* or *await* messages according to the current accessing states of the residing resources. In the second phase, TMs commit the transactions and release their locked resources.

A transaction consists of a set of resource requests with their combination style. The generality or the system transparency of the proposed algorithm is realized by the application of $f(n, T)$.

Definition 1 The *resource request satisfaction predication f* is a 2-tuple (n, T), where $n = \{r_1, r_2, \ldots, r_M\}$ is a set of required resources; $T = \{\Phi, and, or\}$ is the set of connection operators which represent the resources request combination style.

Different combinations of n and T denote various resource request models of distributed systems: $(1, \Phi)$ denotes the one-resource request model, which is the mostly used model where a transaction has only one resource request at a time, only if it is satisfied can the transaction send the next request; (n, and) denotes the *AND* model where the resource requests of a transaction are sent simultaneously and all of them must be satisfied for the transaction committing; (n, or) denotes the *OR* model where the transactions committing need only part of the resource requests being granted; (n, T) denotes the *AND-OR* model or k out of N model where the resource request satisfaction condition is undefined for every transaction. The applications of these models can be found in the Distributed Data Base systems (DDBS) and communication networks etc.

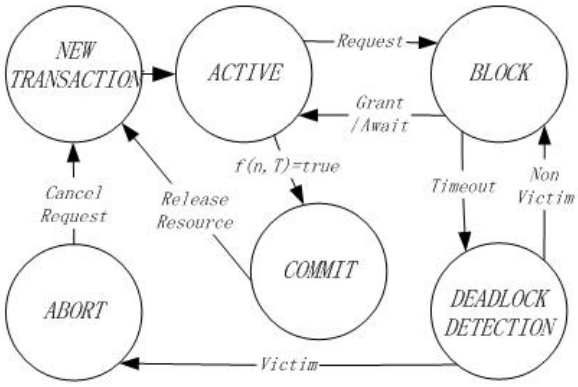

Fig. 1 Transaction life-cycle

The running of general distributed system model can be presented by the state transformation graph in figure 1, which is explained by message passing as follows, where i and r denote the transaction node and resource node respectively.

I TM_i executes: {i has four states： *ACTIVE*; *BLOCK*; *COMMIT* and *ABORT*}

I.1 when i is initialed, it is in *ACTIVE* state, sends all *request* messages to the required RMs; puts i into *BLOCK* state;

I.2 upon receives *grant* or *await* messages, computes $f_i(n, T)$; if $f_i(n, T)$ is true, puts i into *COMMIT* state, executes I.3; else executes I.4;

I.3 sends *release* messages to all granted RMs; sends *cancel* messages to all un-granted RMs; puts i into *ABORT* state and abort it;

I.4 if $f_i(n, T)$ is false until *Timeout*, executes *DEADLOCK DETECTION*;

II RM_r executes: {r has two states: *BUSY* or *IDLE*}

II.1 upon receives a *request* message, adds the requester to the request table; if r is in *IDLE* state, executes II.2; else sends *await* message to the requester;

II.2 sends *grant* message to the TM of the first requester in request table; puts r into *BUSY* state;

II.3 upon receives *release* or *cancel* messages, purges the requester from the request table; if the message is *release*, puts r into *IDLE* state;

II.4 if the request table is not empty, executes II.2 ;

In above executions, if I.4 happens, the system involved in blocks. Now we give the WFG and DWFG definition which reflects the blocking state.

Definition 2 WFG is a direct graph (N, E), where each node $i \in N$ denotes a blocked transaction and its holding resources, the direct edge $e \in E$ between a pair of nodes denotes the waiting relation, such as $ij \in E$ denotes the execution of transaction i is blocked, it is waiting for the transaction j release its holding resource, in such situation, i is called the *predecessor* of j and j is called the *successor* of i.

Definition 3 A *Cycle* is a loop with the same originator and terminus in WFG; A *knot* is a nonempty set of nodes such that the reachable set of each node in the knot is exactly the kont.

Definition 4 The DWFG is a 4-tuple (N, E, ^, D), where N and E are the same as that in WFG; the operator ^ stands for a discrete time, $D = \{q, j, \lambda\}$ stands for the quitting, joining and static actions of a node in N respectively. The dynamic shift of DWFG can be expressed by such infinite sequences: S^λ, S^q, S^j, S^q^j, S^j^q^q … where S is the initial value of N.

Although the waiting relation between transactions may be uncertain in DWFG, the topologies of the deadlocks is not changed, namely, there are still cycles or knots exist in DWFG.

3 New Algorithm

By definition 3, the existence of a cycle is the necessary condition for a knot existence, so the knot is more general than the cycle in describing a deadlock topology. In the field of knot detection, diffusion computation is the most suitable approach. The basic idea of the diffusion computation is that the algorithm responds the knot in WFG to a direct spanning tree (DST): the deadlock detection initiator is the *root*, the node without successor is the *leaf*, the nodes between *roots* and *leaves* are the *neutral* nodes, then the deadlock detection is realized by a depth-first search of the DST: A root sends the detection messages, probes to all of its successors, the successors propagate probes to their successors in turn and echo the predecessors and/or roots based on their current states, finally the root concludes the knot members and designates a victim node quitting from the system to resolve the deadlock.

The diffusion computation algorithms can be classified according to the knot reduction happening places: the initiator reduction (IR) algorithms and the neutral node reduction (NR) algorithms. The knot reduction happens at root in IR algorithms [11, 14] while at neutral nodes in NR algorithm [4, 10]. Furthermore, some algorithms [9, 12] collect all the knot members for the manager.

The proposed algorithm is an optimized IR algorithm and gives the total members of a knot in a special set *blk_s*; a successor set *suc_s*, records the current successors informed by the un-granted resource nodes; an initial node set *ini_s*, records the initiators of the probes. The data structure of a probe is *Probe*(*ini*, *suc*, *ts*), where *ini* is an identifier stands for the initiator of the probe, *suc* is a parameter stands for the next detecting object node, *ts* is the time stamp of the probe, it is assigned with the local time of the initiator. The timing order of *ts* can be used for the priority comparison between nodes in DWFG either. Every transaction node in DWFG can send probe and a node can send multi probes. The data structure of a echo message for a probe is *Ack*(j, sp, *ts*), where j and sp stands for the identifier and state of the on going detected node respectively, *ts* comes from the latest arrived probe.

3.1 Deadlock Detection

The outline of our deadlock detection can be described by the executions of a root i and its direct or indirect successor j:

III Executions of TM_i :

III.1 initial: $blk\_s_i := suc\_s_i$; $ini := i$; $ts := local\ time$; sends *Probe*(i, j, *ts*) to all $j \in suc\_s_i$;

III.2 upon receives *Ack*(j, sp, *ts*), $blk\_s_i := blk\_s_i \cup suc\_s_j$;

III.2.1 if $sp = suc\_s_j$ and $i \in suc\_s_j$, declares *blk_s* is a cycle; *Abort victim*(i); /*real victim*/

III.2.2 elseif $sp = \Phi$, declares *blk_s* is a knot; *Abort victim*(j); /*virtual victim*/

III.2.3 elseif sp = stop, declares *blk_s* is a knot; *Abort victim*(j); /*virtual victim*/

III.2.4 else $sp = -k$, $blk\_s_i := blk\_s_i - k$;

IV Executions of TM_j :{ upon receives *Probe*(i, j, *ts*)}

$ini\_s_j := ini\_s_j + i$;

IV.1 if *ts* > *local time*, discards *Probe*(i, j, *ts*); /*an obsolete probe*/

IV.2 elseif $suc\_s_j \neq \Phi \wedge ts < local\ time$, sends *Ack*($j$, suc_s_j, *ts*) to i; sends *Probe*(i, k, *ts*) to all $k \in suc\_s_j$; /*j is not a leaf*/

IV.3 elseif $suc\_s_j = \Phi$, sends *Ack*(j, Φ, *ts*) to all $i \in ini\_s_j$; /*j is a leaf*/

IV.4 elseif j is in *COMMIT* or *ABORT* state, sends *Ack*(j, stop, *ts*) to all $i \in ini\_s_j$; /*j has already quit*/

IV.5 elseif $k \in suc\_s_j$ and j is in *COMMIT* or *ABORT* state, sends *Ack*(j,-k, *ts*) to all $i \in ini\_s_j$;

/*one successor quit*/

3.2 Deadlock resolution

The victims in deadlock resolution have to be aborted actively or passively in order to resolve the deadlock.

V *Abort victim* (v) /*deadlock resolution*/

V.1 TM_v executes I.3;

V.2 the root sends resend resource request messages to all $k \in blk\_s \backslash v$;

3.3 Discussion

Definition 5 The *virtual victim* is the midway quitting node before the deadlock detection finish.

If multi deadlock detections running concurrently, when a victim node receive probes after it quitting in III.2.2 and III.2.3, its TM will echo the probe with the messages in IV.4 and IV.5, now the status of the quitted node transforms from the successor of one detection to the quitted victim of other running detections, by this way, other detections may terminate passively because the victim is known. The alternative passive and active termination conditions ensure multi deadlock detections can terminate eventually. Since the victims in III.2.2 and III.2.3 are not the real victims detected by the normally deadlock detection, they are named the virtual victim.

In order to decrease the detection overhead and select the single victim, in IV.1, only the probes with the lower *ts* than the detected nodes can be forwarded.

4 Correctness Proof

The dynamic shift of DWFG is unordered, by definition 4, if infinite q and j actions happen in a period of time, the correctness proof of the algorithm will be very complicate. For the paper length limited, we give the correctness proof under the assumption of only one q or j action happening in a deadlock detection round. Because of the rare happening rate of deadlocks in distributed systems (or else the deadlock avoid algorithms should be adopted), and the few quitting or joining nodes in DWFG, the assumption is acceptable.

Singhal put forward the correctness criteria [15] of a deadlock detection algorithms, that is 1) *Liveness*. If a deadlock happens, the algorithm should detect it in finite steps; 2) *Safety*. If an algorithm declares a deadlock happens, the deadlock exists truly. While in a dynamic distributed system, the safety condition can not be satisfied: during a deadlock detection round, any halfway quitting node would destroy the integrity of a cycle or knot. Besides, the correctness criteria proposed by Singhal lacks termination conditions for concurrent running deadlock detections, it is useful just for one deadlock detection round. Therefore the correctness conditions of the proposed algorithm should be: 1) *Narrow sense consistency*: once deadlock detection produces only one victim; 2) *Broad sense consistency*: the concurrent deadlock detections can terminate eventually; 3) *Livness*.

For describe simply, we assume that the lower priority of the node, the lower value of its *ts*; the character n and e represent the number of nodes and edges of DST respectively; all messages passing have the same interval and other executions are instant.

Theorem 1 *The proposed algorithm satisfy narrow sense consistency.*

Proof Recall the deadlock detection termination condition in section 3 part III, we have three cases:

Case1. In III.2.1, if the deadlock is a cycle, there must be the lowest priority node exists in the cycle, by IV.1, only the probes originated from the node can pass through all the nodes in the cycle, any probe initialed by other nodes may be discarded at least by the lowest priority node. Namely, only the lowest priority node can become the victim;

Case2. In III.2.2, if the deadlock is a knot, by IV.3, the root near the leaf would receive the reply in advance, but no matter how many deadlock detections find the knot, the victim is the same leaf, thus the unique property of the leaf guarantees only one victim can be selected;

Case3. In III.2.3, if a node quits during the deadlock detection, it is the virtual victim, obvious it is unique.

Theorem 2 *The proposed algorithm satisfy broad sense consistency.*

Proof The running of nodes in distributed system are independent, none of them has the global state information. Whether or not the concurrently running detections can terminate eventually is decided by the reduction results of the roots. If some concurrent deadlock detections intersect at some nodes and one of them terminates before others, by discussion in section 3.3, the victim created by the first terminate deadlock detection will be transformed to other deadlock detections virtual victims, so every deadlock detection will terminates eventually.

Theorem 3 *Liveness*

Proof Consider all the possible dynamic shift in DWFG, we have:

Case1. S^λ

1.1 Deadlock is a cycle

Although it is possible that each node in the cycle can create probes, by case 1 in theorem 1, only one root can conclude the entire cycle, the deadlock detection steps are the same as the diameter of the cycle in DWFG;

1.2 Deadlock is a knot

In theorem 1 case 2, the lowest deadlock detection steps is 2 when a root sends probes to a leaf directly; the upper limit deadlock detection steps are no more than *n*.

Case2. S^*q*

The quitted node is defined as the virtual victim in our algorithm, it will lead to the termination of the current running deadlock detection, so no matter the original topology of a deadlock in DWFG is cycle or knot, this quit action will transform the topology to a knot and the deadlock detection steps are decreased by at least 1. By IV.4 in section 3.1，there may be multi roots execute III.2.3, but because the total number of *n* is decreased, the upper limit deadlock detection steps are *n*-1.

Case3. S^*j*

3.1 Deadlock is a cycle

3.1.1 If the joined node hasn't the lowest priority in DST, by IV.1, the probes originated from it must be discarded by some nodes in DST. The joined root can not affect original deadlock detection;

3.1.2 If the joined node has the lowest priority in DST, the probes originated from it will either arrive at all members of the cycle or be discarded because of the original deadlock detection termination. While in the former situation, the joined root will not satisfy any termination condition in section 3.1 III.2.1, III.2.2 and III.2.3; in the last situation, the joined root may be satisfy the termination condition in III.2.3, but it can not affect the original deadlock detection.

So the original deadlock detection will continues and terminate eventually.

3.2 Deadlock is a knot

3.2.1 If the joined node is waiting a leaf directly, it will sends probes to the leaf, by section 3.1 IV.3 and III.2.2, the deadlock detection will terminate within 2 steps. This detection result will make the former running detections executing IV.4 and III.2.3 and terminating eventually;

3.2.2 If the joined node is waiting for the leaf indirectly, the original detections will keep running like 1.2 above. It is possible the joined node executes IV.4 and III.2.3, but in any situation the upper limit of the detection is less than *n* steps;

So the upper timing limit for all deadlock detections is *n* steps.

5 Performance Evaluation

Due to the dynamic property of the DWFG, the performance of the algorithm is undetermined. In order to given a valid performance evaluation, we only consider the situation of ^λ in DWFG.

By the proofs of section 4, if the deadlock is a knot, the lower limit of time complexity is 2: a root and its directly waiting node, the leaf forms the simplest knot. It will be detected in 2 steps; while the upper limit is known as *n*. The lower limit of message complexity is also 2：a probe and a reply message; the upper limit for a single deadlock detection is *e*+*n*-1: if the distance between the root and the leaf is *n*，the probes initialed by the root will be sent to all the edges of the DWFG, the total number is *e*, all the detected nodes will reply the probes, the total number is *n*-1.

If the deadlock is a cycle, the lower limit of time complexity is still 2 in case of the cycle consists of two nodes; the upper limit is still *n* by theorem 3. The lower limit of message complexity is obvious 2; the upper limit of message complexity can be computed as follows: for the node with the lowest priority in the cycle, its probes will be propagated all the successors, so there are *e* probes for a single deadlock detection, every detected node except the root will reply each probe, so there are *n*-1 reply messages, the total number of message is *e*+*n*-1.

The comparison of the time and message complexity of the proposed algorithm and previous algorithms is listed in table 1. It shows our algorithm is favorable than other algorithms.

Table 1 Performance comparison

Algorithms	Messages /n	Delay
Misra and Chandy [2]	4*e*	2*n*
Kshemkalyani [10]	4*e*-2*n*+2*L*	2*n*
Kshemkalyani [11]	2*e*	2*n*
Boukerche [12]	2*e*	2(*n*+1)
Chen [13]	2*n*	2*n*
Lee [14]	<2*e*	*n*+2
Proposed	<*e*+*n*-1	*n*

6 Conclusion

Deadlock detection in distributed system is difficult because of the independent running and lacking of global information of each node. Most existing proposals just represented how to search cycle or knot in a static WFG, while neglected the dynamic shift of the underlying distributed system,

they failed to run concurrently or have false detections. The algorithm proposed in this paper eliminates these defects. Main advantage of our algorithm is that the occurring of leaf, fault or node quitting which may make the detection interrupt are viewed as the detection termination conditions, enable the multiple detections terminate when node disappear in the corresponding DWFG. On the other hand, all the resource request condition is denoted by a resource request satisfaction predication, therefore the proposed algorithm can be applied to any distributed systems. The correctness including narrow sense consistency, broad sense consistency and Livness for our algorithm is proved. Performance evaluation shows the message and time complexity of the proposed algorithm is $<e+n-1$ and n respectively under a static WFG, where n and e stand for the number of nodes and edges of the DST respectively, the performance is better than that of the previous algorithms.

References:

[1] E. Knapp, Deadlock Detection in Distributed Databases, *ACM Computing Surveys*, Vol. 19, No. 4, 1987, pp. 303-328.

[2] K.M. Chandy and J.Misra, A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems, *In: Proc ACM SIGACT-SIGOPS Symp Principles of distributed computing*, 1982, pp. 157-164.

[3] D.P. Mitchell, M.J. Merritt, A Distributed Algorithm for Deadlock Detection and Resolution," *In: Proc ACM conf Principles of distributed computing*, 1984, pp. 282-284.

[4] J. Brzezinski et al., Deadlock Models and a General Algorithm for Distributed Deadlock Detection, *IEEE Trans. Parallel and Distributed Systems*, Vol. 31, No. 2, 1995, pp. 112-125.

[5] N. Krivokapic, A. Kemper, E. Gudes, Deadlock Detection in Distributed Database Systems: A New Algorithm and a Comparative Performance Analysis *VLDB Journal*, Vol. 8, No. 2, 1999, pp. 79-100.

[6] S. Lee, Performance Analysis of Distributed Deadlock Detection Algorithms, *IEEE Trans Knowledge and Data Engineering*, Vol. 13, No. 4, 2003, pp. 623-636.

[7] D. Manivannan, M. Singhal, "A Distributed Algorithm for Knot Detection in a Distributed Graph," *Proc Int'l conf Parallel Processing*, 2002, pp. 485-492.

[8] G. P. Souza, G.H. Pfitscher, An Implementation of a Distributed Algorithm for Detection of Local Knots and Cycles in Directed Graphs Based on the CSP Model and Java, *In: Proc 6th IEEE Int'l Workshop on Distributed Simulation and Real-Time Applications*, 2002, pp.143-150.

[9] D. Manivannan, M. Singhal, An Efficient Distributed Algorithm for Detection of Knots and Cycles in a Distributed Graph, *IEEE Trans Parallel and Distributed Systems*, Vol. 14, No. 10, 2003, pp. 961-972.

[10] A.D. Kshemkalyani, M. Singhal, Efficient Detection and Resolution of Generalized Distributed Deadlocks, *IEEE Transaction on Software Engineering*, Vol. 20, No.1, 1994, pp. 43-54.

[11] A.D. Kshemkalyani, M. Singhal, Distributed Detection of Generalized Deadlocks, *In: Proc 17th Int'l Conf Distributed Computing Systems*, 1997, pp.553-560.

[12] Boukerche, C. Tropper, A Distributed Graph Algorithm for the Detection of Local cycles and knots, *IEEE Trans Parallel and Distributed Systems*, Vol. 9, No. 8, 1998, pp. 748-757.

[13] S. Chen et al., Optimal Deadlock Detection in Distributed Systems Based on Locally Constructed Wait-for Graphs, *In: Proc 16th Int'l Conf Distributed Computing Systems*, 1996, pp. 1-15.

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

[15] M. Singhal, "Deadlock Detection in Distributed Systems," *IEEE Computer*, Vol. 22, No. 2, 1989, pp. 37-48.