Efficient Deadlock Detection in Distributed Systems

Shigang Chen, Yi Deng, Cyril Orji and Wei Sun
School of Computer Science
Florida International University
Miami, Florida 33199

Abstract

The performance of a deadlock detection scheme, in terms of number of message transmission and the size of the messages, is an important concern in distributed systems. In this paper, we propose an incremental approach for deadlock detection, which can dramatically improve the performance of previously published centralized and hierarchical deadlock detection schemes. Two deadlock detection algorithms, a centralized and a hierarchical, are proposed. These algorithms are capable of detecting all deadlocks and detecting no false deadlock. Correctness proofs and detailed performance analysis are provided.

1 Introduction

For distributed systems, the performance of a deadlock detection algorithm, in terms of number of message transmission required and the size of the messages, is an important concern, because it directly contributes to the load of, and has great impact on the performance of the entire system in which the deadlock detection algorithm is deployed [8].

Generally speaking, deadlock detection algorithms for distributed systems can be classified into three classes: *centralized, hierarchical* and *distributed*. Hierarchical deadlock detection represents a good compromise between centralized and distributed deadlock detection algorithms [9]. On one hand, the hierarchical solutions do not have the problems of single point failure and communication congestion around control sites as in centralized deadlock detection algorithms. On the other hand, comparing to distributed deadlock detection algorithms [1, 2, 3, 6, 7, 10] they have simpler control structure, require much less message transmission for deadlock detection, and allow simpler deadlock resolution strategies [5].

In hierarchical deadlock detection algorithms [4, 5], sites are grouped as clusters based on resource access patterns, and clusters are organized in a hierarchical fashion. In each cluster, a designated control site is responsible for detecting deadlock within the cluster using a centralized algorithm. The control site accomplishes this by collecting system status information from its descendant sites, constructing a cluster resource graph (RG) (also often called wait-for graph (WFG)), and detecting directed cycle(s) in the graph. For the global system, a site is designated as the central control site. The central control site is responsible for detecting inter-cluster deadlocks by periodically collecting inter-cluster status information from its immediate descendant cluster control sites, and constructing inter-cluster RG in a similar centralized fashion (the central controller and its immediate descendant cluster controllers form a cluster representing the entire system).

The performance of a hierarchical deadlock detection algorithm, in particular, highly depends on its underlying centralized deadlock detection algorithm. In the previous proposed algorithms [4], every time a cluster control site initiates a new round of deadlock detection, every site in the cluster must send the *complete* status information about the site in order for the control site to construct a RG that reflects the state of the cluster. This *complete* status information not only includes the status changes occurred between the last and the current rounds of detection, but also include the entire history of resource allocations and requests, as well as the status of the transactions (processes) at the site. Consequently, the deadlock detection algorithms requires transmitting large-sized messages.

In this paper, we propose an **incremental** approach for deadlock detection in distributed systems, which can dramatically reduce communication cost in centralized and hierarchical deadlock detections. We first present a new centralized deadlock detection algorithm. In this algorithm, the control site maintains a RG, which records the history of *wait-for* and *holding* relations between the transitions (processes) and resources in the system (or cluster). Every time the

control site initiates a new round of deadlock detection, each site in the system only sends the necessary status changes at the site occurred between the last round and the current round of deadlock detection to the control site. Because each site only sends status changes (which may be none) to the control site, our algorithm dramatically reduces the size of the messages sent, and may reduce the number of message needed as well. Correctness proofs and performance analysis of the algorithm are provided. Followed the above discussion, we then present a hierarchical deadlock detection algorithm by combining our new approach with a similar hierarchical algorithm as in [4]. We show that with a straightforward addition to our centralized algorithm, a low cost hierarchical deadlock detection algorithm is produced.

The rest of the paper is organized as follows: In Section 2, the new centralized deadlock detection algorithm and its correctness proofs are provided, followed by its performance analysis in Section 3. The hierarchical deadlock detection algorithm is presented in Section 4. We conclude the paper in Section 5.

2 An Efficient Deadlock Detection Scheme

In this section, an efficient centralized deadlock algorithm based on the idea of Imprecise Resource Graph (IRG), which supports the incremental approach for deadlock detection, is first presented. The discussion about the algorithm is followed by correctness proofs. The algorithm serves as the basis of the hierarchical deadlock detection algorithm to be discussed in Section 4.

It is assumed that a distributed system is composed of a collection of sites (machines) with local memory and CPU connected by a communication network. Transactions and resources are spread over all sites in the system. For simplicity and without losing generality, we assume that each transaction resides at one site, and it acquires one resource at a time. Every site and every transaction have a unique identification called *Site_ID* and *Trans_ID*, respectively. It is further assumed that communication channels are error free.

2.1 The algorithm

As a centralized deadlock detection scheme, our algorithm shares the same characteristics of other centralized algorithms, namely, a designated control site is responsible for deadlock detection. The control site has the knowledge of all other sites in the system. Periodically, the control site initiates a new round of deadlock detection by broadcasting a message to all other sites. Upon receiving the message, each site reports its local information to the control site. Once the information from all the sites are received, the control site constructs a global Resource Graph (RG) (such a graph is called a demand graph in [4]), and tries to detect directed cycle in the RG. However, a problem with previously proposed algorithms, e.g. [4], is that every time the control site initiates a deadlock detection, every site has to send the complete local information, e.g. the entire local RG, to the control site, even though part of the information has been sent to the control site in the previous rounds of detection, and/or part of the information sent does not contribute to the detection of the deadlock. Consequently, these algorithms suffer from large communication overhead in terms of both number of messages sent and the size of the messages.

The key difference between our algorithm and those in [4] is that, in our algorithm, the control site keeps a RG, called *Imprecise RG (IRG)*, which contains the deadlock detection information collected in the previous rounds of detections, and only partially reflects the real status of the system (see discussion below). Every time a new round of detection is initiated, a site only sends the minimal update information (which has not been reported before) to the control site, which in turn uses the information to incrementally update the IRG. No same piece of information will be sent twice, and certain classes of information, which are transmitted (possibly more than once) in the existing algorithms, e.g. [4], are never transmitted in our algorithm. Therefore, our algorithm dramatically reduces the communication overhead.

Definition 1 (1) A RG is a directed graph composed of two classes nodes, transactions or resources. An edges from a resource R to a transaction T indicates that T is holding R; an edge from a T to a R means T is waiting for R. (2) For a transaction node T in the RG, the set of its incoming edges is called its holding set (HSet for short); an edge in the set is called a $HEdge$. Its outgoing edge, if any, is called its waiting-for edge (WFEdge for short) [1].

Generally speaking, there are four types of events that may change a RG (or WFG) in deadlock detection:

[1] Since a transaction requests one resource at a time, the transaction node has at most one outgoing edge.

1. A transaction requests a resource, and the resource is granted to the transaction immediately.
2. A transaction requests a resource and the resource can't be granted immediately (The transaction is thus blocked).
3. A resource is granted to a blocked transaction (The transaction is unblocked).
4. A transaction releases one or more resources it holds.

In our algorithm, however, only the information about type 2 and 3 events are stored at a local site and may be transmitted to the control site (one of the reasons that improves the performance of our algorithm, see Section 4 for details).

The name of IRG comes from the fact that a waiting-for (or holding) edge in IRG doesn't necessarily mean that there is a waiting-for (or holding) relationship between the corresponding transaction and resource (because such a relation may be outdated at the time of observation). On the other hand, when a transaction is waiting-for (or holding) a resource, there may not be such a waiting-for (or holding) edge in the IRG (because the true status information is not sent to the control site).

Each site maintains two tables called $ForePool$ and $BackPool$, which are used to store status information about the transactions local to the site. The pools have the following structure and are manipulated in the following way (Pool update rules):

1. A transaction T is blocked if it requests a resource, but fails to get it immediately (type 2 event). Whenever this happens, an entry $< trans\_id, wfedge, hset >$ is inserted into the $BackPool$, where trans_id is the identification of T, $wfedge$ represents the resource waited by T and $hset$ represents the set of resources currently held by T. Such an entry is called a $BlockEntry$.
2. Whenever a transition is unblocked due to the resource it is waiting for is granted (type 3 event), one of the following actions are taken:

   2.1. If there is a $BlockEntry$ in the $ForePool$ or $BackPool$ with the same $trans\_id$, remove the $BlockEntry$.

   2.2. Otherwise, insert an entry $< trans\_id >$ to the $ForePool$. Such an entry is called an $UnblockEntry$.

Clearly, the above rules guarantee that for each transaction T, there is at most one entry in the $ForePool$. The same properties holds for the $BackPool$ as well. The proposed deadlock detection algorithm is described as follows:

Algorithm 1
1. Periodically, the control site initiates a round of deadlock detection by broadcasting an (initiation) message to all sites.
2. Whenever a site S receives the initiation message, the following actions are taken:

   2.1. If the $ForePool$ at S is not empty, send the $ForePool$ to the control site; otherwise, simply send the $Site\_ID$ of S to the control.

   2.2. Replace the $ForePool$ with the content of the $BackPool$, and set the $BackPool$ empty. The reply message sent by a site to the control site is called a $SiteMessage$.
3. When the control site receives replies from all the sites, for each $ForePool$ message (FPM) received it takes the following steps:

   3.1.
   For each $BlockEntry < trans\_id, wfedge, hset >$ in FPM, use the $wfedge$ and $hset$ to replace the $WFEdge$ and $Hset$ of the node associated with $trans\_id$ in the IRG.

   3.2. For each $UnblockEntry < trans\_id >$ in the FPM, remove the $WFEdge$ of the transaction node associated with $trans\_id$ in the IRG.
4. Search for directed cycle in the IRG. The system is deadlocked iff there is a directed cycle in the IRG.

2.2 Correctness Proof

In this sub-section, we show that the deadlock detection algorithm presented in the previous sub-section is capable of detecting all deadlocks and no false deadlock is detected.

The following lemmas demonstrate the properties of the IRG at the control site after the $SiteMessages$ from every sites in the system are received (i.e. no $SiteMessage$ is under transmission), and all the updates of the graph defined in Algorithm 1 are completed.

Lemma 1 If a transaction node T has a $WFEdge$ in the IRG, the last status information about T received by the control site must be a $BlockEntry$ of T; if T has no $WFEdge$ in the IRG, the last status information about T received must be an $UnblockEntry$.

Proof: A direct consequence of the algorithm. □

Lemma 2 For a transaction node T in the IRG, if T has a $WFEdge$, one of the following two statements must be true:

1. There is an $UnblockEntry$ in the $ForePool$ of the site where T resides.

2. i) T is still blocked; ii) the $WFEdge$ of T reflects a true waiting-for relation; and iii) every $HEdge$ in $HSet$ of T reflects a true holding relation.

Proof: We show that (2) must be true if (1) is false. Suppose there is no $UnblockEntry$ of T in the $ForePool$ at the site where T resides. Because T has a $WFEdge$ in the IRG, the last status information of T received by the control site must be a $BlockEntry$ (Lemma 1). Since no $SiteMessage$ is on the way to the control site and no $UnblockEntry$ of T is in the $ForePool$, T hasn't yet been unblocked. Therefore, the $WFEdge$ of T reflects a true waiting-for relation. Because T remains to be blocked after the last $BlockEntry$ of T received by the control site, T can't release any resource it holds. So All the $HEdges$ in its $HSet$ are also true. □

Lemma 3 If an $UnblockEntry$ is created in the system before a $BlockEntry$, regardless whether or not they are created by the same transaction, and regardless whether or not they are created at the same site, the $UnblockEntry$ will arrive at the control site before the $BlockEntry$, or they will arrive in the same round of deadlock detection.

Proof: Suppose an $UnblockEntry$ A is created before a $BlockEntry$ called B, where A and B may be created by the same or different transactions.

Case 1: A and B are created in the same site. According to the algorithm, an $UnblockEntry$ is always put in the $ForePool$ and a $BlockEntry$ is always put in the $BackPool$. The $ForePool$ will be sent to the control site before $BackPool$. So A will be sent to the control site at least one round of deadlock detection before B.

Case 2: A and B are created at different sites. When B is put in the $BackPool$ of some site, A has been put in the $ForePool$ of some other site, or has been already sent to the control site. Based on the algorithm, in the former, A will be sent to the control site in the current or next round of deadlock detection; but the earliest time B will be sent is the next round. Therefore, the lemma is true. □

Theorem 1 If the system is not in a deadlock, there will be no direct cycle in IRG.

Proof: Assume the system is not in a deadlock, and there is a directed cycle, $T_0 \rightarrow R_0 \rightarrow T_1 \rightarrow ... \rightarrow T_{n-1} \rightarrow R_{n-1} \rightarrow T_0$, in the IRG.

First, we show that there is at least one transaction in the cycle having an $UnblockEntry$ in the $ForePool$ of the site where it resides. That is, Lemma 2 (1) is true for at least one transaction in the cycle, because otherwise, from Lemma 2 (2), every transaction in the cycle is blocked; and all the edges in the cycle represents the true (waiting-for or holding) relations, which means the system in in the deadlock, thus contradicts the assumption.

Second, without losing generality, assume transaction T_0 has an $UnblockEntry$ U_0 in $ForePool$, and T_0 creates U_0 at time t_0, we further show that every transaction in the cycle must has an $UnblockEntry$ in the $ForePool$ of the site where it resides. According to Lemma 1, the last status information about T_1 received by the control site is a $BlockEntry$ (called B_1). Let B_1 be created at time t_1. According to Lemma 3, B_1 must be created before U_0. At time t_1, R_0 is held by T_1 (see the rule for updating $BackPool$). Before U_0 can be created at time t_0, R_0 has to be acquired by T_0 to unblock T_0. For this to happen, the following sequence of events must occur during the time period $[t_1, t_0]$: (a) T_1 is unblocked, so that (b) T_1 can release R_0, and then (c) R_0 is granted to T_0.

When T_1 is unblocked, an $UnblockEntry$ called U_1 is created (see Pool update rules), thus U_1 is created before U_0. Because the control site hasn't received U_1 when the cycle in the IRG is detected, U_1 must be in the $ForePool$ of the site where T_1 resides. By the same token, there are $UnblockEntries$ U_2,..,U_n for T_2 .. T_n in the $ForePools$ at the sites where T_2,..,T_n reside, respectively. Furthermore, $U_{(i+1) \ mod \ n}$ is created before U_i, $i = 0..n-1$. However, this implies that U_0 is created before itself, thus cause a contradiction.

Therefore, the theorem holds. □

Theorem 2 If the system is in deadlock, a direct cycle in IRG will be detected within the following two rounds of deadlock detection.

Proof: Suppose the system is in deadlock and the deadlock cycle is $T_0 \rightarrow R_0 \rightarrow T_1 \rightarrow ... \rightarrow T_{n-1} \rightarrow R_{n-1} \rightarrow T_0$. We show that, once the deadlock is formed, the above cycle appear in the IRG within two rounds of deadlock detection.

When the last time T_i, $i = 1..n-1$, is blocked before the formation of the deadlock, a $BlockEntry$ B_{T_i}

for T_i is created in the $BackPool$ at the site where T_i resides. Since T_i remains to be blocked after B_{T_i} is created, and its status remains unchanged, according to the algorithm, the entry B_{T_i} will be send to the control site within two rounds of detection. Therefore, the waiting-for and holding relations between T_i and the resources it waits for and it holds are correctly reflected in the IRG. This also implies that, if $T_j, 0 \leq j \leq n-1$, is the transition causing the deadlock, B_{T_j} will also be received by the control site within two rounds of deadlock detection after the deadlock is formed. □

3 Performance Analysis

Our algorithm provides a better performance because the IRG at the control site are built incrementally instead of built from scratch in every round of deadlock detection as in the existing algorithms. In any round of detection, a site only sends the status changes occurred between the last and current rounds of detection. Furthermore, as shown in Section 2, only part of the changes will be transmitted to the control site. In this section, detailed performance analysis about our new algorithm is provided, and compared with previously published algorithms. The following notations and conventions are used in the analysis:

R is the average rate the transactions at a site request resources, that is, there are on average R requests issued at a site per unit of time.

P is the probability that a request is blocked (i.e. not granted immediately). Such a request is called a *blocked request* in the sequel. Every occurrence of such an event will create a $BlockEntry$.

Blocking time of the blocked request (or the $BlockEntry$) is the time interval from the moment the request is issued to the moment the request is granted.

T_0 is the number of time units between two consecutive rounds of deadlock detection.

M is the probability for the *blocking time* of a *blocked request* to be $> 1.5T_0$.

T_b is the expected mean of *blocking time*.

N_t is the average number of transactions at a site at a given instant.

Let's first consider the average size of the messages transmitted between a site and the control site in the system.

A $BlockEntry$ (or $UnblockEntry$) will be inserted to $BackPool$ (or $ForePool$) only when a transaction is blocked(unblocked). No action will be taken either when a transaction requests a resource and receives it immediately, or when it releases some resources it holds. Furthermore, when an $UnblockEntry$ A about a transaction T is created, if there is a $BlockEntry$ B about T in $ForePool$ (or $BackPool$), then both A and B will be removed from $ForePool$ (or $BackPool$). This means a *blocked request* with short *blocking time* will never be known by the control site. Due to the above reasons, the $SiteMessage$ in our algorithm has a much shorter average size than traditional centralized deadlock detection algorithms. A quantitative analysis is given below:

On average there are $R \times T_0$ requests in a site during each T_0, among which $P \times R \times T_0$ are *blocked requests*. This means that $P \times R \times T_0$ $BlockEntries$ will be created during each T_0. For each $BlockEntry$ created, there will be a corresponding $UnblockEntry$ when the blocked request is granted. The average time for a $BlockEntry$ to stay in the $BackPool$ is $0.5T_0$, and the time for the same entry to stay in the $ForePool$ is T_0. So the total average time for a $BlockEntry$ to stay in the Pools is $1.5T_0$ (Therefore, on average, for a $BlockEntry$ to be sent to the control site, the *blocking time* of the entry has to be larger than $1.5T_0$). So on average a $SiteMessage$ will consist of $M \times P \times R \times T_0$ $BlockEntries$ and $M \times P \times R \times T_0$ $UnblockEntries$ (see definition for M). For every transaction, it may have either a $BlockEntry$ or an $UnblockEntry$ in a $SiteMessage$, but not the both. Therefore, $M \times P \times R \times T_0 \leq 1/2N_t$. Because the body of an $UnblockEntry$ only contains $Trans\_ID$, the size of an $UnblockEntry$ is a constant c (several bytes). The size of $BlockEntries$ differ from transaction to transaction and from time to time. Suppose the average size of a $BlockEntry$ is C. The average size for $SiteMessage$ is $M \times P \times R \times T_0(c+C)$. Therefore, the average SiteMessage size in our algorithm is bound by $1/2N_t(c+C)$.

However, the actual average $SiteMessage$ size is smaller than the above bound. From the above discussion, the size of $SiteMessage$ depends on the production of M and T_0. For simplicity, assume the *blocking times* of all *blocked requests* evenly distributed in the interval $(0, 2T_b]$. This assumption is reasonable because T_b is the average blocking time. In practice, there may be a very small number of

blocked requests whose *blocking time* is greater than $2T_b$, but the number is so small that can be negligible. Based on the assumption, it can be derived that $M = 1 - 3 \times T_0/(4 \times T_b)$. For the above equation, it can be seen that $M \times T_0$ will reach its maximum value ($= T_b/3$) when T_0 is equal to $2/3T_b$. $P \times R$ is the average rate of *blocked requests* in a site. N_t/T_b is the maximum rate of *blocked requests*, which only occurs in the case that whenever a transition at a site is unblocked, it becomes blocked again immediately. $P \times R \leq N_t/T_b$ (normally, $P \times R \ll N_t/T_b$). Therefore, the average size of SiteMessage is $M \times P \times R \times T_0(c+C)$ $\leq (T_b/3) \times (N_t/T_b) \times (c + C) = N_t(c + C)/3$.

Notice that the above result is extremely conservative. In practise, the average message size in our algorithm should be much smaller than $N_t(c + C)/3$. Consider the following: (a) it is extremely unlikely that $P \times R = N_t/T_b$ (i.e. Every transaction becomes blocked immediately after it is unblocked); (b) It is reasonable to choose T_0 to be larger $4/3T_b$, which implies $1.5T_0 > 2T_b$. If so, $M \times T_0$ will be much smaller $T_b/3$, because most $BlockEntries$ will be removed from the $ForePool$ (or $BackPool$) before the entries have a chance to be moved to the $ForePool$, and sent to the control site. (Recall that, for a $BlockEntries$ to be sent to the control site, the *blocking time* of its corresponding *blocked requests* has to $> 1.5T_0$.)

Even with the upper bound, this average message size is much better than the previously published algorithms of the same class, e.g. [4], in which the average message sizes in their two-phase and one-phase algorithms are N_tC , $2N_tC$, respectively. Normally c is much less than C, thus the average SiteMessage size in our algorithm is approximately 1/3 of the message size in the two-phase algorithm and 1/6 of the one-phase algorithm in the worst case.

In terms of the number of messages transmitted during a round of deadlock detection, our algorithm, in the worst case, has the same complexity comparing to previously published algorithms, e.g. the one-phase algorithm in [4]. This is because that all the algorithms in this class follow the same basic framework, that is, in each round, the control site initiates deadlock detection by broadcast a message to all other sites, and they in turn send a $SiteMessage$ to report the status information at each site. More specifically, in the worst case, our algorithm needs 1 broadcast message + N_s $SiteMessages$ (N_s is the number of sites in the system).

However, in average cases, our algorithm requires less number of messages. Recall from our algorithm, if the $ForePool$ at a site S is empty at the time when the initiation message from the control site is received, S only needs to send its $Site\_ID$ to the control site. This communication between S and the control site can be easily implemented by attaching a flag bit in the Acknowledgement (ACK) (which is required by the network communication protocol to realize error-free message passing) to the control site. No actual message needs to be sent.

Therefore, the only time that S needs to send a $SiteMessage$ to its control site is when S's $ForePool$ is not empty. The $ForePool$ is empty if the following conditions are true: (a) The *blocking time* of all the $BlockEntries$ in the $ForePool$ of S is smaller than $1.5T_0$ (this condition ensures that all the $BlockEntries$ in the $ForePool$ will be removed before the receipt of the message from the control site); and (b) for every $UnblockEntry$ created, there is a $BlockEntry$ in either the $ForePool$ or $BackPool$ with the same $Trans\_ID$ (this condition prevent any generated $UnblockEntry$ from being added to the $ForePool$). Based on the earlier analysis, if $T_0 > 4/3T_b$, there is a large possibility that both of the above conditions can be satisfied.

4 A Hierarchical Deadlock Detection Algorithm

A purely centralized algorithm is not desirable to large distributed systems, because (1) it may cause communication congestions around the control site, and (2) the failure of the central control site will disable the entire system. For these reasons, hierarchical deadlock detection schemes are proposed [4], [5], which effectively remedy the above problems, while having much simpler control structures than distributed deadlock detection algorithms. Such hierarchical schemes are particularly effective if the resource access pattern is very localized.

In this section, we show that with a trivial addition to our centralized deadlock detection algorithm, our algorithm can be combined with the hierarchical scheme [4]. However, because the cost of our centralized algorithm is much smaller than the one in [4], the resulting hierarchical deadlock detection algorithm will have a much better performance. In the following, we first briefly describe a hierarchical deadlock scheme [4], and then show how to adapt our algorithm to the scheme.

In this hierarchical scheme, the sites in a distributed system are grouped into a number of *clusters* based on the resource access pattern. Periodically, a central

control site is chosen as the control of all the clusters, and a site in each cluster is chosen as the cluster control site.

Definition 2 (1) In the RG of a cluster, a transaction node T is called an *input transaction node* (or *output transaction node*) if its $WFEdge$ (or at least one of its HEdge) connects a resource node in another cluster. (2) A resource node R is called an *input resource node* (or *output resource node*) if at least one of its incoming edge is from (or at least one of its outgoing edge is to) another cluster.

Definition 3 For a cluster, its *Compressed IRG (CIRG)* is defined as follows: (1) all the nodes in CIRG are input/output (transaction or resource) nodes; (2) an input node I is in CIRG iff there is a directed path from I to an output node O, and vice versa; (3) For every path in (2), there is an edge $I \rightarrow O$ in CIRG.

An example CIRG is shown in Figure 4.

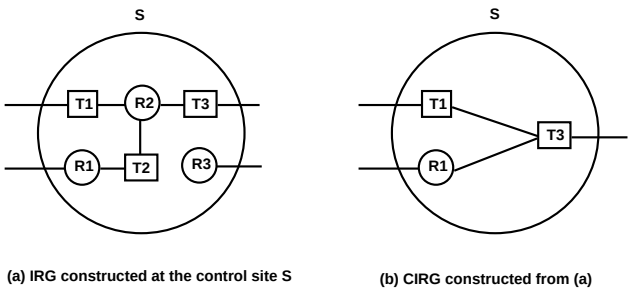

Figure 1: An Example of Constructing CIRG

Definition 4 A waiting-for (holding) edge is called an inter-cluster edge if it is across two clusters.

The hierarchical deadlock detection algorithm is described as follows:

1. The central control site broadcasts an (initiation) message to all cluster control sites requesting them to send their CIRGs and inter-cluster edges, and waits until all information is received.
2. When a cluster control site receives the initiation message, it
   (a) performs the deadlock detection algorithm of Section 2 within the cluster, and
   (b) constructs its CIRG (see below), and send the CIRG and its inter-cluster edges to the central control site.
3. When the central control site receives replies from all the cluster control sites, it constructs a RG of the whole system using both the CIRGs and inter-cluster edges. The system is deadlocked if there is a directed cycle in the constructed RG.

Based on Definition 3, to construct CIRG for each cluster, we need to know the input/output (transaction/resource) nodes in the IRG of the cluster. The information about input/output transaction nodes is readily available in the IRG. However, the information about the input/output resource nodes can not be directly derived from the graph. To find these nodes, additional information is needed. We first have the following definition:

Definition 5 A *ResourceEntry* is defined as a tuple: $< Res\_ID, Trans\_ID, In, Out >$, where Res_ID and $Trans\_ID$ are resource and transaction identifications, respectively; and In and Out are integers in the domain of $\{-1, 0, 1\}$, where -1 means the resource node (represented by Res_ID) is no longer an input (In $= -1$) or output (Out $= -1$) node, 1 means the node becomes an input/output node, and 0 means no change. $< Res\_ID, Trans\_ID >$ is called the header of the entry.

Definition 6
Given two *ResourceEntries*, $< R, T, In_1, Out_1 >$ and $< R, T, In_1, Out_1 >$, their addition "+" is defined as $< R, T, In_1 + In_2, Out_1 + Out_2 >$.

Consider that a transaction T in site S_1 needs a resource R in site S_2, and S_1 and S_2 belong to different clusters. There are four possible relations between T and R:

- T issues a request for R and the request cannot be granted immediately. This case is represented by a *ResourceEntry* $RE =< R, T, 1, 0 >$;
- T issues a request for R, and the request is granted immediately, represented by $RE =< R, T, 0, 1 >$;
- T is unblocked due to the receipt of R, represented by $RE =< R, T, -1, 1 >$; and
- T releases R, represented by $RE =< R, T, 0, -1 >$.

When one of the above events occurs, the following action is taken at site S_2:

- If there is no *ResourceEntry* in the $ForePool$ of S_2 with the same header as RE, insert RE to the $ForePool$; or

- if there is an *ResourceEntry* RE' in the $ForePool$ of S_2 with the same header as RE, replace RE' with $RE' + RE$ if $RE' + RE \neq < R, T, 0, 0 >$, otherwise remove RE' from the pool.

To fine the input/output resource node of a cluster, the cluster control site maintains two variables for each resource R, $counter[R, I]$ and $counter[R, O]$. The cluster control site performs the following action upon the receipt of $SiteMessage$ from each site in the cluster: for Each $ResourceEntry < R, T, In, Out >$, $counter[R, I] = counter[R, I] + In$, and $counter[R, O] = counter[R, O] + Out$.

The CIRG for the cluster is constructed at the end of each round of deadlock detection in the cluster. During the construction, a resource node R is an input resource node iff $counter[R, I] > 0$; and R is an output resource node iff $counter[R, O] > 0$.

5 Conclusion

We have introduced an incremental approach for deadlock detection in distributed systems, which can significantly improve the performance of centralized and distributed deadlock detection schemes in terms of the number of messages sent, and particularly the size of the messages transmitted between the sites and the (system or cluster) control sites. A centralized and a hierarchical deadlock detection algorithms are presented under the approach. Comparing to existing algorithms, our algorithms provide better performance, because the RG at the control site(s) is built incrementally instead of built from scratch in every round of deadlock detection as in the existing algorithms. In any round of detection, a site only sends the status changes occurred between the last and current rounds of detection. Furthermore, only part of the changes, which is absolutely necessary for detecting deadlock, will be transmitted to the control site.

Acknowledgements

This work is supported in part by the National Science Foundation (NSF) under Grant No. CDA-9313624.

References

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

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

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

[4] G. S. Ho and C. V. Ramamoorthy. Protocol for deadlock detection in distributed database systems. *IEEE Trans. on Software Eng.*, SE-8(6):554 – 557, Nov. 1982.

[5] D. A. Menasce and R. R. Muntz. Locking and deadlock detection in distributed data base. *IEEE Trans. on Software Eng.*, SE-5(3):195 – 202, May 1979.

[6] R. Obermarck. Distributed deadlock detection. *ACM Trans. on Database Sys.*, 7(2):187 –208, June 1982.

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

[8] M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, pages 37 – 48, Nov. 1989.

[9] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems.* McGraw-Hills, 1994.

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