Deadlock Resolution and Semantic Lock Models in Object–Oriented Distributed Systems

*Marina Roesler**
Walter A. Burkhard†

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093

Abstract

We propose a distributed algorithm for detection and resolution of resource deadlocks in object-oriented distributed systems. The algorithm proposed is shown to detect and resolve all $O(n!)$ cycles present in the worst case waits-for-graph (WFG) with n vertices by transmitting $O(n^3)$ messages of small constant size. Its average time complexity has been shown to be $O(ne)$, where e is the number of edges in the WFG. After deadlock resolution, the algorithm leaves information in the system concerning dependence relations of running transactions. This information will preclude the wasteful retransmission of messages and reduce the delay in detecting future deadlocks.

1 Introduction

Many previous attempts to provide a distributed algorithm for handling resource deadlocks exist. However, as we will see below, these attempts have not been entirely successful. The algorithms proposed either fail to work correctly, or exhibit poor performance. Furthermore, most algorithms proposed do not provide a full solution to the problem. Clearly, it is not sufficient to detect deadlocks; deadlocks must also be resolved.

* Partially supported by the CNPq — Brazilian Research Council.

† Partially supported by the NCR Corporation and the University of California MICRO grant.

Our algorithm detects deadlock in a manner similar to that of some other path-pushing deadlock detection algorithms [CM82,HM83,SN85,CKST87,BT87] in that no explicit maintenance of the waits-for-graph (WFG) is required. As opposed to many existent deadlock detection algorithms [Gra78,CM82, CMH83,HM83,BT87], it will also eliminate the deadlock after it is detected. Furthermore, after deadlock resolution, the algorithm leaves information in the system concerning dependence relations of running transactions. This information will preclude the wasteful retransmission of messages and reduce the delay in detecting future deadlocks.

Previous attempts to provide an algorithm which would resolve deadlocks in addition to detecting them [MM79,GS80, Obe82] have not been successful. The main source of deficiencies for those algorithms lies on their inability to perceive/restore the WFG to a consistent state after the resolution phase is completed [Elm86]. As a consequence, these algorithms may either detect false deadlocks, or fail to detect real ones.

More recently, based on work by Chandy and Misra [CM82], and Moss [Mos81], Sinha and Natarajan proposed two algorithms to detect and resolve deadlocks for distributed database systems [SN85]. Their first algorithm is of limited interest since it can only be used in conjunction with concurrency control algorithms (CCA) based on the (very restrictive) exclusive lock model. Their second algorithm applies to CCAs using the read/write lock model. Under some circumstances, both algorithms may detect false deadlocks and fail to detect real ones. The nature of the deficiencies of their second algorithm is discussed in [RBC87]. Problems associated with their first algorithm are presented, and a modified version of the algorithm is proposed in [CKST87]. The modified algorithm is still only applicable to CCAs based on the exclusive lock model. Moreover, as we have shown in [RBC87], one of the modifications proposed is itself deficient, leading to false deadlock detection under some circumstances.

We present a distributed algorithm for detection and resolution of resource deadlocks in object-oriented distributed system. The algorithm proposed applies to concurrency control algorithms which use the semantic lock model [Kor83,Sch84,Wei84, RB87]. Since the read/write and the exclusive lock models are restricted forms of the semantic lock model, our algorithm applies to CCAs based on these models as well.

In order to drastically reduce message traffic, our algorithm properly identifies and eliminates redundant messages. It will detect and resolve all O(n!) cycles in the worst case WFG by transmitting $O(n^3)$ messages of constant size. In contrast, the second algorithm proposed in [SN85] needs to transmit an exponential number of messages[1]. Very few of the algorithms proposed in the literature address the issue of performance. Three papers which present some performance results are [CM82], [HM83], and [BT87]. However the algorithms proposed in these papers do *not* address the issue of deadlock resolution. In [CM82] the algorithm presented has two phases. In the first phase, vertices in the WFG independently initiate computations to detect whether or not they are part of some deadlock cycle. When every vertex initiates a probe computation, this phase of the algorithm requires the transmission of $O(n^4)$ messages for the worst-case WFG. In the second phase, each deadlocked vertex will determine which other vertices belong to the same deadlocked portion of the graph. However, all vertices which are not part of a cycle, but wait transitively on vertices in the cycle, also will, unnecessarily, compute information about the cycle. No performance results are presented for this expensive phase of the algorithm. In [HM83] an algorithm combining the two phases of the above algorithm is proposed. The authors state that their algorithm requires transmission of O(n!) messages to detect all cycles in the worst case WFG. A general algorithm adopting the N–out–of–M request model is proposed in [BT87] having the lowest worst-case performance among previously published algorithms. In their algorithm, a single invocation requires the transmission of $O(e)$ messages. Since every vertex in the WFG may independently initiate an algorithm invocation, the algorithm will transmit $O(n^3)$ messages in the worst-case WFG.

This paper is organized as follows. We introduce a model for object oriented distributed systems in section 2. In section 3, a distributed algorithm for detection and resolution of resource deadlock is proposed. The algorithm's time and space complexity is analyzed in section 4. In section 6, we present our conclusions and final comments.

2 The Model of Computation

An object oriented distributed system is modeled as a collection of sites containing transactions and data objects which synchronize their operations through messages. Transactions and data objects are controlled by ***transaction managers*** and ***object managers***, respectively. A transaction accesses objects indirectly by communicating its desire to its controlling transaction manager (TM), which then sends a message to the appropriate object manager (OM). Although transaction and object managers may maintain more than one transaction or object, we assume, with no loss of generality, that each transaction manager controls one transaction and each object manager controls one object. We will refer to transaction T_i's manager as TM_i. We use a similar convention for object managers.

[1]The reported performance of their first algorithm is $O(n^2)$. The performance of our algorithm, when used with exclusive–lock models, is $O(n^2)$ also.

2.1 Object Managers

The internal structure of an object manager is shown in figure 1. An OM has the following attributes:

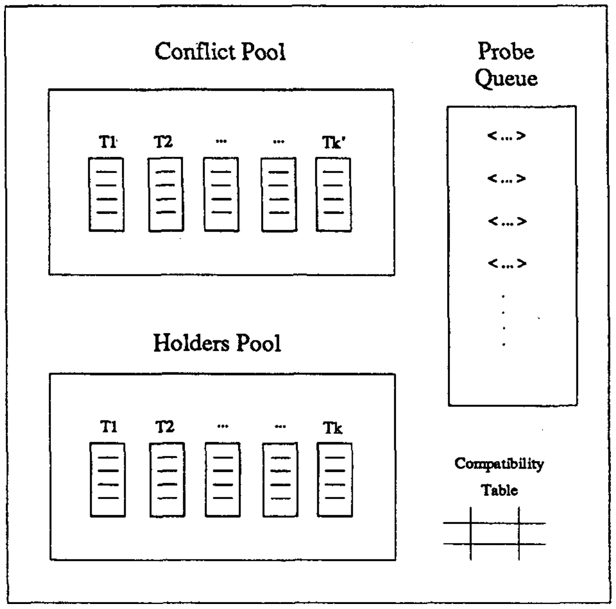

Figure 1: Object Manager Structure

Holders Pool Contains information about those transactions which currently hold a lock on the object. We refer to these transactions as *holders*.

Conflict Pool Contains information about those transactions which currently have an outstanding request on the object, in conflict with one or more holders[2]. We refer to these transactions as *requestors*. The Conflict Pool associates with each requestor a list of all the holders with which it conflicts.

Probe Queue Contains messages which have passed through this object manager in the process of deadlock detection and resolution. In section 3 we will describe how Holders Pools, Conflict Pools, and Probe Queues interact in this process.

Compatibility Table Contains information about compatibility of operations on the object. This table is used by the concurrency control algorithm only [3].

Notice that concurrency control algorithms using locking may differ in the locking model adopted. For example, algorithms based on the exclusive lock model allow only one holder per object at a time. In contrast, algorithms based on the read/write lock model may allow more than one holder since the object can

[2]Since nested transactions are not considered here, transactions may only have one outstanding request at one time.

[3]For examples of different compatibility tables see [Gra78,Kor83, Sch84,RB87].

be shared among transactions requesting read locks. Accordingly, requestors in the object's Conflict Pool are assumed to be in conflict with *all* holders. Algorithms based on semantic lock models [Sch84,BBG86,RB87] are more flexible. Each requestor may be in conflict with only a subset of the holders, and *further*, each requestor may conflict with a different subset of holders. For example, suppose that the Holders Pool of object Set contains two operations Insert(A) and Insert(B), which belong to transactions T_1 and T_2, respectively. Let T_3 request operation Delete(B), and T_4 request operation Delete(A). Then, T_3 is in conflict with T_2 but not with T_1, and T_4 is in conflict with T_1 but not with T_2. The Holders and Conflict Pools structures described above capture the requirements of all these different locking models.

2.2 Transaction Managers

A transaction manager (TM) consists of two components: the transaction's body and a probe queue.

When a transaction T_i requests an object X, TM_i sends a message to OM_x which either grants or denies the request, depending upon whether or not T_i is in conflict with some transaction already holding X. OMs grant requests by sending a response[4] back to the requesting TM, which in turn passes the response to the transaction it manages. Until a transaction receives a response, it is considered to be in a *wait* state. While it is in a wait state, this transaction is essentially frozen. However, a TM is always active with respect to deadlock detection and resolution, regardless of the state of the transaction it manages.

When a transaction T_i waits on object X because it is in conflict with some transaction T_j holding X, T_i is said to *depend* on T_j. Traditionally, the set of dependencies among transactions has been depicted by a directed graph, the transaction WFG, where a vertex i represents transaction T_i, and a directed edge (i, j) represents T_i's dependency on transaction T_j. Although our algorithm does not require the explicit maintenance of the WFG, we will use it as an abstraction to aid the reader in visualizing the algorithm's behavior. Thus, when we say that an edge (i, j) was inserted (removed) in the WFG, we mean that an OM has detected that a dependency between transactions T_i and T_j exists (ceases to exist).

3 The Deadlock Resolution Algorithm

In a manner similar to other path-pushing deadlock detection algorithms, our algorithm detects deadlock by generating messages, propagating them through the WFG, and detecting a deadlock cycle as some message returns to its initiator. These messages are referred to as *probes*. In our algorithm a probe consists of the ordered pair $< initiator, last >$, where $initiator$ is the index of the transaction which initiated the probe, and $last$ is the index of the transaction (manager) which last forwarded this probe. When a deadlock cycle is detected, the algorithm resolves the deadlock by aborting $T_{initiator}$. We also introduce a special type of message called an *antiprobe*. An antiprobe has the same structure as a probe[5]. The roles of probes and antiprobes are discussed next.

3.1 Creation and forwarding of probes and antiprobes

Probes (antiprobes) will be created and forwarded according to a priority scheme. The use of priority schemes to reduce message traffic was first suggested by [Obe82]. The priority scheme we adopt (similar to the one proposed in [SN85]) further assures that each deadlock cycle in the WFG will be detected by one and only one of the cycle's participants.

When a transaction is activated in the system it is assigned a unique integer identifier. Identifiers are assigned in increasing order. This identifier determines the priority of the transaction (younger transactions have higher priority). Subscripts are used to denote priority levels, i.e. T_i has priority i, and for any two transactions T_i and T_j, if $i > j$, then T_i has higher priority than T_j. If $i > j$ then edge (i, j) is said to be an *antagonistic* edge. The concept of antagonism holds for probes as well. If the attribute *initiator* in a probe has higher priority than some transaction T_k, the probe is said to be antagonistic with T_k.

Probes originate in object managers. When an antagonistic edge (i, j) is introduced into the WFG, the OM detecting the edge creates a probe p on the edge's behalf and forwards it to TM_j. Probe p consists of $< i, \perp >$ where $\perp$ represents a null value. If T_j is in a wait state when TM_j receives p, it forwards p to the OM on which T_j is waiting. Recall that in the semantic lock model, a transaction may depend on a subset of the object's holders. Therefore, when an OM receives a probe p it must determine which transaction (manager) forwarded it in order to route p to the appropriate object's holders. The attribute *last* is used for this purpose. Moreover, the OM only forwards p to the TMs of all transactions on which T_{last} depends, *and with which p is antagonistic*. As an example consider the scenario depicted in figure 2.

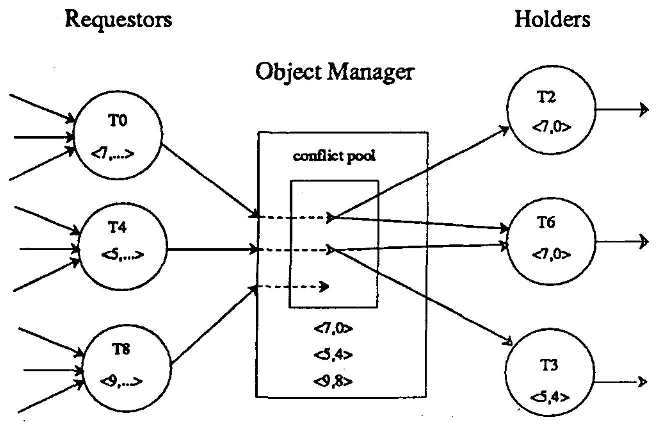

Figure 2: Probe Flow

T_0 is in conflict with T_2 and T_6, and T_4 is in conflict with T_6 and T_3. A probe $< 7, 0 >$ entering the OM from TM_0 would be forwarded to TM_2 and TM_6 since it is antagonistic with both T_2 and T_6. In contrast, probe $< 5, 4 >$ entering from TM_4 would be forwarded to TM_3, but not to TM_6.

[4]The contents of the response may vary for each C.C.A. In the case of the Peephole Scheduler and other semantics-based algorithms [Sch84,Wei84], the response contains the return value of the requested operation on the object.

[5]It is differentiated from a probe by a leading bit.

Probes are assumed to arrive at their destination in finite time, error-free, and in the order in which they were sent. Notice that there is often a communication delay between the time a transaction requests an object and the time it receives a response, even when the request is immediately granted. Thus, a TM cannot differentiate between the following two situations, both of which cause the transaction to wait : a) the transaction request could not be granted and, b) the transaction request was granted but the response message didn't arrive yet. While the transaction waits for a response, the transaction's TM assumes it is dependent upon some transaction holding the object, and forwards all probes received during this interval to the object's OM. A probe received by an OM is considered *meaningful* if the transaction sending the probe does indeed depend on some other transaction holding the object[6]. Probes which are not meaningful are discarded by the OM.

As an example consider figure 2 again. Transaction T_8 has made a request to the OM shown. Upon receiving this request, the OM verifies that T_8 is *not* in conflict with any of the object's holders. Meanwhile, until T_8 receives a response from the OM it is in a wait state; any probes received by TM_8 must be forwarded to this OM. For example, probe $< 9, \dots >$ would be forwarded to the OM. However, the OM will discard the probe when it receives it, since $< 9, 8 >$ is not meaningful. Notice that probes flow only from OMs to TMs and conversely, but never between two TMs, or between two OMs.

As a probe traverses the WFG, copies of it are stored in the probe queues of the OMs and TMs through which it passes. For example, let P be the directed path from T_i to T_n in the WFG depicted in figure 3 (for simplicity we omit the OMs from the figure). Assume T_i is the highest priority transaction in P.

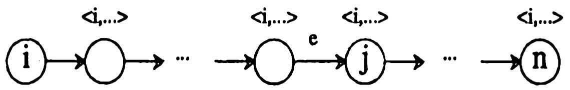

Figure 3: Stale Probes

Then probe $< i, \dots >$ will be stored in the probe queues of all transactions (managers) T_k in P, $i \neq k$. Let P' be the directed subpath from T_j to T_n. If edge e is later removed, then the directed paths connecting T_i to each transaction T_q in P' cease to exist. Thus the probe queue of each such T_q contains a stale probe $< i, \dots >$. An antiprobe $< i, \dots >$ is then created and forwarded to T_j with the purpose of destroying all records of probe $< i, \dots >$ in P'. An antiprobe mimics the behavior of its associated probe, following its same paths. However, an antiprobe has a "negative" effect on the probe queues of the OMs and TMs through which it passes – it causes the associated probe to be removed from these probe queues. Recognition of the associated probe will require only a match of the probe's and antiprobe's attributes *initiator* and *last*. Antiprobes are used to restore the WFG to a consistent state after the deadlock is resolved.

[6]Since messages arrive in order, if a probe was sent after a request message, it will arrive at the OM after the request does. Accordingly, the OM must process the request first, in order to be able to decide whether or not the probe is meaningful.

3.2 Elimination of redundant probes

Consider the partial WFG G presented in figure 4 (for simplicity we omit the OMs from the figure).

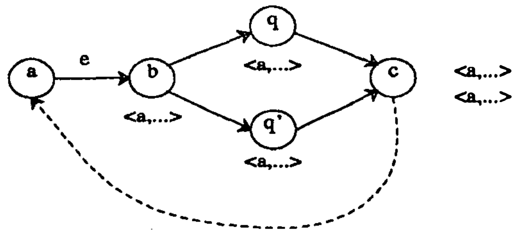

Figure 4: Redundant Probes

There exist two distinct paths connecting vertex a to vertex c. For simplicity, assume T_a is the highest priority transaction in this WFG. When a probe $p = < a, \dots >$ generated on behalf of edge e arrives at some vertex v in G it will be forwarded along all outgoing edges of v. The number of replicas of p which will finally arrive at vertex c (TM_c) represent the number of distinct paths from a to c traversed by p (two in our example). We argue that it is sufficient to store only one of these replicas in TM_c's probe queue, and maintain a counter for p. The argument is as follows. Suppose, on the contrary, that both probes are stored in TM_c's probe queue, and let (c, a) be a new edge inserted in the graph. Then both stored probes are forwarded and two deadlock cycles will be detected. Indeed G contains cycles a–b–q–c–a and a–b–q'–c–a. However, when the first cycle is detected, T_a is aborted and consequently *both* deadlocks are resolved. We conclude that the paths represented by the two replicas of p which arrived at TM_c are indistinguishable with respect to deadlock resolution – when T_a is aborted, both paths are severed. Thus, one of these replicas is redundant and doesn't need to be forwarded. A natural question arises : could the second replica arriving at TM_c then be discarded? Intuitively not. The number of p's replicas arriving at TM_c represent the number of paths connecting a to c traversed by p. This information should not be disposed of. In fact, if the second probe is discarded and no record is kept that this probe existed, real deadlocks may not be detected later. For example, consider graph G' presented in figure 5.

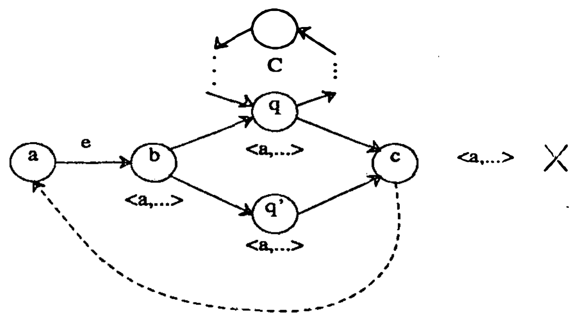

Figure 5: Unnoticed Real Deadlock

Assume that the second probe was discarded and no record was kept that this probe existed. Let T_q may be the highest priority transaction in some cycle C. When C is detected, T_q is aborted severing a path from a to c. Accordingly, an antiprobe for $< a, \ldots >$ will arrive at TM_c (section 3.1). This antiprobe would then remove the only probe $< a, \ldots >$ stored in TM_c. When edge (c, a) is now inserted in G', cycle a–b–q'–c–a is created but it will not be detected.

In order to preserve information about the number of paths traversed by p, we discard one of the replicas but keep a counter for p. We adapt the antiprobe's behavior accordingly . When an antiprobe finds a matching probe p, it first decrements p's counter. Only when the decremented counter is zero — indicating that there are no more remaining paths — are both the probe discarded from the probe queue and the antiprobe forwarded. In our example, probe $< a, \ldots >$'s counter would be equal to 2 before T_q is aborted. Afterwards, when the antiprobe arrives at TM_c, it decrements $< a, \ldots >$'s counter to 1 and stops. When edge (c, a) is inserted, probe $< a, \ldots >$ will be forwarded, and the deadlock is then detected.

The actual algorithm follows. It is composed of two main modules. The first module is executed by each TM in the system, and the second one by each OM. We describe below how OMs and TMs cooperate to accomplish deadlock detection and resolution. In response to the following events, a TM/OM invokes the procedures described below. Each procedure is executed atomically.

3.3 Module for TMs

- A lock request on object X is issued by TM_i

```
tm.request_object (X)
begin
    for all probes $P_q$ in $TM_i$'s probe queue
    do
            make copy $P$ of $P_q$
            forward $P$ to $OM_x$
    od
end
```

As we have seen in section 2.2, TM_i considers T_i to be in a wait state until T_i gets a response from OM_x. Accordingly, the probes in TM_i's probe queue are forwarded to OM_x (which will then decide whether or not they are meaningful).

- Probe P arrives at TM_j

```
tm.save_and_forward (P)
begin
    $P.last := j$
    if $P$ is redundant
    then increment counter for $P$
            discard $P$ ; return
    fi
    make copy $P'$ of $P$
    set $P'$'s counter to 1
    save $P'$ in $TM_j$'s probe queue
    if $T_j$ is waiting on object X
    then forward $P$ to $OM_x$
    else discard $P$
    fi
end
```

When TM_j receives a probe P, it updates its attribute last. Then, TM_j compares P to each probe in its probe queue to determine if P is redundant. If it is, TM_j increments P's counter and discards P. Otherwise, TM_j saves a copy of P in its probe queue and, if T_j is in a wait state, forwards P to the object manager on which T_j is waiting.

- Antiprobe A arrives at TM_j

```
tm.clean_and_forward (A)
begin
    $A.last := j$
    decrement counter for probe $P$ matching $A$
    if $T_j$ is waiting on object X and counter for $P$ is 0
    then forward $A$ to $OM_x$
    else discard $A$
    fi
end
```

Since the goal of an antiprobe is to follow the course of its matching probe, the forwarding of antiprobes is analogous to that of probes. Accordingly, if redundant probes arrive at TM_j, matching redundant antiprobes will also arrive at TM_j. Since no redundant probes are forwarded, a redundant antiprobe arriving at TM_j will not be forwarded either. If the arriving antiprobe removes from the probe queue the last matching probe (counter=0) then it must be forwarded since the first arriving probe was forwarded (assuming the transaction is in a wait state).

3.4 Module for OMs

- OM_x detects that a dependency (antagonistic) between T_i and T_j exists.

```
insert_edge(i,j)
begin
    if $i > j$
    then create newprobe := $< i, \perp >$
            forward newprobe to $TM_j$
    fi
    for all probes $P_q$ in $OM_x$'s probe queue
    do
            if $P_q.last = i$
            then if $P_q.initiator = j$
                    then abort($j$)
                    else  if $P_q.initiator > j$
                            then make copy $P$ of $P_q$
                                    forward $P$ to $TM_j$
    od      fi      fi      fi
end
```

When OM_x detects that an antagonistic dependency between T_i and T_j exists, it creates and forwards a probe on behalf of the new edge. Notice that, because T_i can be in conflict with more than one of X's holders, OM_x's probe queue may already contain probes which entered OM_x from TM_i. Thus, OM_x next verifies whether the new edge closed a cycle. Also, OM_x forwards to TM_j all probes which entered OM_x from TM_i, and which are antagonistic with T_j.

- OM_x detects that a dependency (antagonistic) between T_i and T_j ceases to exist.

```
remove_edge(i,j)
begin
    if $i > j$
    then create antiprobe := $< i, \perp >$
            forward antiprobe to $TM_j$
    fi
    for all probes $P_q$ in $OM_x$'s probe queue
    do
            if $P_q.last = i$
            then  if $P_q.initiator > j$
                    then  make antiprobe $A$ for $P_q$
                            forward $A$ to $TM_j$
    od      fi      fi
    if j was last holder on which i depended
    then  for all probes $P_q$ in $OM_x$'s probe queue
            do
                    if $P_q.last = i$
                    then discard $P_q$
    fi      od      fi
end
```

When OM_x detects that an antagonistic dependency between T_i and T_j has ceased to exist, OM_x creates and forwards an antiprobe on behalf of the stale edge. Additionally, OM_x must create and forward antiprobes for all probes which entered its probe queue from TM_i, and which are antagonistic with T_j. Notice that there is no need to create an antiprobe for a probe which entered OM_x from TM_i but signaled deadlock, since probes signaling deadlock are never forwarded. If the edge (i, j) removed is the last outgoing edge from i (i.e. T_j was the only holder of object X on which T_i depended), all probes in the probe queue that had entered OM_x from TM_i become meaningless and should be removed from the probe queue.

- Probe P arrives at OM_x

```
om.save_and_forward (P)
begin
    if $P.last$ doesn't depend on some holder
    then discard $P$ ; return
    fi
    make copy $P'$ of $P$
    save $P'$ in $OM_x$'s probe queue
    for all holding transactions $T_h$ in $OM_x$
    do
            if transaction $P.last$ depends on $T_h$
            then  if $P.initiator = h$
                    then abort($h$)
                    else  if $P.initiator > h$
                            then forward $P$ to $TM_h$
    od      fi      fi      fi
end
```

Recall that when some TM_i issues a lock request on some object X, TM_i forwards its probe queue to OM_x. Thus, when OM_x receives a probe it must first check whether or not T_i is a requestor in its Conflict Pool. If not, then T_i's lock request was granted, the probe is deemed not meaningful, and it is discarded. When a meaningful probe P arrives at OM_x, a copy of P is saved in OM_x's probe queue. Additionally, OM_x determines with which of its holders the transaction sending the probe conflicts. The probe is compared against these conflicting holders checking a) for deadlock and b) for antagonistic conflicts. For each holder, if a cycle is detected, then the holder is aborted and the probe is not forwarded. Otherwise, if an antagonistic conflict is detected, the probe is forwarded to the holder's TM.

- Antiprobe A arrives at OM_x

```
om_clean_and_forward (A)
begin
    if $A.last$ doesn't depend on some holder
    then discard A ; return
    fi
    remove matching probe from $OM_x$'s probe queue
    for all holding transactions $T_h$ in $OM_x$
    do
            if transaction $A.last$ depends on $T_h$
            then  if $A.initiator > h$
                    then forward A to $TM_h$
    od      fi      fi
end
```

Since the goal of an antiprobe is to follow the course of its corresponding probe the forwarding of antiprobes is analogous to that of probes. Notice that if the removed matching probe from OM_x's probe queue had signaled deadlock, there is no need to forward A further (probe signaling deadlock is never forwarded).

4 Performance Issues

In this section, we analyze the algorithm's asymptotic time and space complexities. Since the cost of transmitting a message through a communication network greatly exceeds the cost of executing an instruction, we adopt the cost of transmitting one message as our time unit. We will show that our algorithm detects and resolves all $O(n!)$ cycles contained in the worst case WFG by transmitting $O(n^3)$ messages of constant size. We also show that each TM needs space to store O(n) messages, where the length of each message is $O(\log_2 n)$ bits. We have shown in [RBC87] that the algorithm's average time complexity is $O(ne)$, where e represents the number of edges in the WFG.

In the algorithm described in section 3 probes flow from TMs to OMs and from OMs to TMs. In order to simplify the counting of messages transmitted we will assume that a probe flows directly from a TM to another TM (represented by vertices in G_0). Deadlocks will now be detected at TMs. We adapt the notion of antagonism accordingly : i is antagonistic with j iff $i \geq j$. Since we eliminate the intermediate transmission step through the OM, we will be undercounting the total number of messages transmitted by a constant factor smaller than 2. Clearly the undercounting has no impact in the asymptotic time and space complexities of the algorithm.

4.1 The Worst Case Scenario

Given a system with n running transactions, the current state of dependencies among transactions is depicted by a WFG with n vertices. As usual, we label each vertex of the WFG with the priority of the transaction associated with it. We refer to the the graph for which our algorithm, in order to detect and resolve all existent deadlock cycles, will transmit the maximum number of messages as the *worst case WFG*. We denote by M_i the number of messages transmitted in order to detect and resolve all existent deadlocks in a graph G_i. Recall that, in our algorithm, probes are only created on behalf of antagonistic edges. Likewise, a probe is transmitted to a vertex only when it is antagonistic with this vertex. Consequently, given any two isomorphic directed graphs G_1 and G_2, *with distinct labelings*, M_1 and M_2 may differ. For example, in figure 6 below, $M_1 = 2$ and $M_2 = 0$.

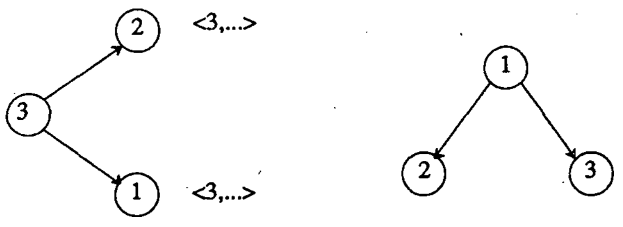

Figure 6: Distinct Labelings for G_1 and G_2

Definition Let L be the set of isomorphic directed graphs with vertices labeled $\{1, 2, \ldots, n\}$. A *maximal* graph for L is a graph G_m s.t. $M_m \geq M_i$, $\forall\ G_i$ in L , $i \neq m$.

Lemma 1 *All complete directed graphs with n vertices are isomorphic.*

Proof. It's easy to show isomorphism of complete directed graphs by looking at their complement graphs. Given two complete directed graphs G_1 and G_2 with n vertices, both $\bar{G}_1$ and $\bar{G}_2$ are graphs with n vertices and no edges. Thus, $\bar{G}_1$ and $\bar{G}_2$ are trivially isomorphic. As a consequence, G_1 and G_2 are isomorphic also. □

Lemma 2 *All complete directed graphs with vertices labeled* $\{1\ ,\ 2\ ,\ \ldots\ ,\ n\}$ *are maximal, i.e.* $M_i = M_j$, $\forall\ G_i, G_j$ *in* $L, i \neq j$.

Proof. Let G_1 and G_2 be two complete directed graphs. By lemma 1, G_1 and G_2 are isomorphic. That is, there exists a one-to-one correspondence between the vertices of G_1 and G_2 such that a pair of vertices is adjacent in G_1 iff the corresponding pair of vertices is adjacent in G_2. Since G_1 is a complete graph, a vertex labeled i is adjacent to each vertex labeled j in the graph, $1 \leq i\ ,\ j \leq n\ ,\ i \neq j$. Same for G_2. Clearly, there exists a mapping $h : \{1, 2, \ldots, n\} \mapsto \{1, 2, \ldots, n\}$, $h(i) = i$, $1 \leq i \leq n$. Therefore, $M_1 = M_2$. □

Theorem 1 *G_0 is a worst case WFG if and only if G_0 is a complete directed graph.*

Proof. (Sketch) In the discussion below, all graphs and subgraphs considered are WFGs with n vertices.
(*Only if*) Assume towards a contradiction that $G_0 = (V_0, E_0)$ is not a complete directed graph. Then let $G_1 = (V_1, E_1)$ be a complete directed graph s.t. $V_1 = V_0$ and $E_1 = E_0 \cup E$, $E \neq \emptyset$. We argue that $M_1 \geq M_0 + 1$. The argument is as follows. If $e = (i, j)$ is in E then either e is an antagonistic edge or it is not. a) If e is antagonistic then $i > j$ and a probe for e is created. Since the probe created is trivially antagonistic with v_j, at least one more message is transmitted in G_1. Thus $M_1 \geq M_0 + 1$. b) If $e = (i, j)$ is not antagonistic then $i < j$. As G_1 is a complete directed graph, then v_i's probe queue will eventually receive a probe containing v_j as initiator. Since $i < j$ and edge (i, j) exists in G_1, this probe is antagonistic with v_i and will be transmitted to v_j. Therefore $M_1 \geq M_0 + 1$. We conclude from a) and b) that $M_1 > M_0$. Thus, G_0 is not a worst case graph. Contradiction.
(*If*) Assume towards a contradiction that G_0 is not a worst case WFG. That is, there exists a graph G_1 s.t. $M_1 > M_0$. By hypothesis, G_0 is a complete directed graph. Therefore, by lemma 2, G_1 is not a complete directed graph. G_1 is then either a subgraph of G_0 or a subgraph of a complete directed graph G_2. a) Let G_1 be a subgraph of G_0. Let E_0 and E_1 be the sets of edges for G_0 and G_1 respectively. Since $E_1 \subset E_0$, all probes created for E_1 are also created for E_0. Note that the converse is not true unless all edges in $E_0 - E_1$ are not antagonistic (in which case the sets of probes created for G_0 and G_1 are the same). Thus, $M_1 \leq M_0$. b) G_1 is a subgraph of G_2. By similar argument as in a), $M_1 \leq M_2$. However, since both G_0 and G_2 are complete directed graphs then $M_2 = M_0$. Therefore, $M_1 \leq M_0$. We conclude from a) and b) that there exists no graph G_1 such that $M_1 > M_0$. Contradiction. □

We showed above what is the worst case configuration G_0 for a WFG with n transactions. The theorem below will establish how many cycles exist in G_0.

Theorem 2 *The number of cycles in G_0 is $O(n!)$.*

Proof. Since G_0 is a complete directed graph, any subgraph of G_0 with i nodes contains $(i-1)!$ cycles of length i (number of circular arrangements). There exist C(n,i) subgraphs with i nodes in G_0. Thus the total number of cycles in $G_0 = \sum_{i=2}^{n}$ C(n,i) $(i-1)! = \sum_{i=2}^{n} \frac{n!}{i(n-i)!} \leq cn!$. □

4.2 The Worst Case Time Complexity

Since we have assumed that probes flow directly from TMs to TMs, the flow of a probe can now be described as follows. Let a probe p arrive at a vertex v_a with neighbor v_b; p is stored in v_a unless it is redundant with a probe already stored in v_a[7]. If any of the *stop conditions* described below holds, p stops at v_a. Otherwise, a replica of p is transmitted to v_b. The stop conditions are :

- Probe p in not antagonistic with v_b,
- Probe p is redundant,
- Probe p signals deadlock (v_a initiated the probe).

Let's analyze the flow of some probe $p =< i, \dots >$, initiated by v_i on behalf of edge (i,j)[8]. As we have seen above, p is replicated at branching vertices and forwarded to their neighbors, successively, until all probes which are replicas of p stop. Notice that all replicas of p contain the same initiator as p (the attribute *last* is irrelevant to the analysis). We refer without distinction to all replicas of the probe, as well as to the original probe, as p.

Theorem 3 *Let $e = (i,j)$ be an antagonistic edge in G_0. Let $M_{i,j}$ be the number of messages transmitted in G_0 due to the insertion of edge e. Then :*

a) $M_{i,j} = i + (i-1)^2$, *if (i,j) is i's first outgoing edge.*

b) $M_{i,j} = 1$, *otherwise.*

Proof. Let (i,j) be i's first outgoing edge, and let p be a probe created on its behalf. We remark that, although in a distributed system the transmission of messages is asynchronous, we incur in no loss of generality by studying the flow of probes in phases. In particular, we will assume that a probe p' created on behalf of some other edge (i,j) which is *not* i's first outgoing edge will only be forwarded when the flow of probe p and its replicas stop. The key observation is that any redundant probe arriving at a vertex will be stopped; which phase stored the first non redundant probe in the vertex is irrelevant.

a) Edge (i,j) is i's first outgoing edge.

Since (i,j) is an antagonistic edge, let $j = h$, $1 \leq h < i$. When edge (i,h) is inserted in G_0 a probe $p =< i, \dots >$ is created on its behalf. For clarity, we study the flow of probe p in two synchronized phases. In phase A, p arrives at v_h and is transmitted to all v_h's pertinent neighbors v_a. In phase B, each probe arriving at each v_a, and which doesn't stop at v_a, is transmitted to all v_a's pertinent neighbors v_b. It will be shown that all probes arriving at each v_b stop. Let M_A and M_B represent the number of messages transmitted in phases A and B respectively. We conclude that $M_{i,j} = M_A + M_B$.

Phase A Probe p is stored when it arrives at v_h. Then p is transmitted to all neighbors v_a of v_h with which p is antagonistic. There exist $i-1$ such neighbors. Thus $M_A = 1+i-1 = i$. Figure 7 below depicts the contents of all vertices in G_0 after this phase, formalized by the following lemma.

[7]If p is redundant, p's counter is incremented

[8]Probe p will necessarily arrive at v_j since edge (i,j) must be antagonistic or else the probe would not have been created.

Lemma 3 *After phase A, each vertex v_a in the WFG has the following property.*

- *If a is in the interval $[i+1,n]$, there exists no probe in v_a.*
- *If a in the interval $[1,i]$, v_a contains one probe p.*

Proof. Probe $p =< i, \dots >$ is not antagonistic with vertices v_a, a in $[i+1,n]$. Thus p could not have been transmitted to v_a. First part of lemma holds. All other neighbor vertices v_a, i.e. $1 \leq a \leq i$, are antagonistic with the probe. Thus one probe p will arrive at each of these vertices. Second part of lemma holds. □

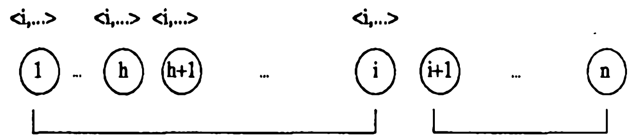

Figure 7: Contents of Probe Queues after Phase A

Phase B In phase A, the propagation behavior of the probe p stored in v_h was studied. In phase B, we study the propagation behavior of the probes p stored in v_a, $a \neq h$. By lemma 3 there exists one such probe in each v_a, $a \in [1,h-1]$ or $a \in [h+1,i]$. Each of these $i-1$ probes is transmitted to all neighbors v_b of v_a, with which the probe is antagonistic. There exist $i-1$ such neighbors. Thus $M_B = (i-1)^2$.

Claim 1 : Each probe transmitted will stop at its destination vertex v_b.

Proof of claim. Recall that each probe $p =< i, \dots >$ is never transmitted to a neighbor vertex v_b in $[i+1,n]$. Thus $1 \leq b \leq i$. By lemma 3 probe p already exists in v_b; probe is not stored and stops. □

Since no probe remain to be transmitted after phase B, the total number of messages transmitted due to the creation of probe $< i, \dots >$ is $M_{i,j} = M_A + M_B = i + (n-i)^2$. □

b) Edge (i,j) is not i's first outgoing edge.

Since (i,j) is an antagonistic edge and it is not i's first outgoing edge, let $j = k$, $1 \leq k < i$, $k \neq h$. A probe $p' =< i, \dots >$ is created on behalf of edge (i,k) and forwarded to v_k. When p' arrives at v_k it is discarded since, by lemma 3, v_k already contains $< i, \dots >$. Thus $M_{i,j} = 1$. □

Theorem 4 *The number of messages M_{G_0} transmitted in G_0 due to the insertion of all possible antagonistic edges is $O(n^3)$.*

Proof. Only antagonistic edges yield the creation of probes in G_0. Since G_0 is a complete directed graph, vertex v_i has $i-1$ antagonistic neighbors. By theorem 3, $i + (i-1)^2$ messages are transmitted due to the creation of the first antagonistic edge (i,h). Also by theorem 3, one message is transmitted due to the insertion of each edge (i,k), for the remaining $i-2$ antagonistic neighbors v_k of v_i.

Then, $M_{G_0} = \sum_{i=2}^{n}(i+(i-1)^2+(i-2)) = \sum_{i=2}^{n}(i^2-1)$. After the sum is evaluated, the term with highest order in the expansion is $\frac{n^3}{3}$. □

Theorem 5 *The total number of messages transmitted to detect and resolve all deadlocks in G_0 is $O(n^3)$.*

Proof. Let C be a cycle in G_0. Let v_i be the vertex with highest priority in C. Then there exists an antagonistic edge (i,j) in C, and a probe p is created on behalf of this edge. Since p is antagonistic with all vertices v_k in C, p will necessarily traverse the cycle and return to v_i signaling the deadlock. So, for each cycle in G_0, one probe is created which will signal deadlock. Thus the number of messages transmitted to detect all cycles is not greater then the number of messages transmitted due to the creation of all probes in G_0. We conclude, by theorem 4, that the number of messages transmitted to detect all deadlocks is $O(n^3)$.

Notice that the maximum number of transactions that can be aborted to resolve all deadlocks is $n-1$. In which case, all edges will be removed from the graph. An antiprobe is created for each antagonistic edge that existed in G_0. As we have seen before, these antiprobes follow the same paths traversed by their corresponding probes. Thus, the number of messages transmitted due to the creation of each of these antiprobes is the same as for the corresponding probe. Therefore, by theorem 4, the total number of messages transmitted to resolve all deadlocks is $O(n^3)$. □

4.3 The Worst Case Space Complexity

Lemma 4 *The number of messages stored in a vertex's probe queue is $O(n)$.*

Proof. Notice that only probes antagonistic with v_g may arrive at some vertex v_g, g in $[1,n]$. Thus, only probes with initiator $i \geq g$ arrive at v_g. Since only one probe with initiator i is stored at v_g's probe queue (the others are redundant and are therefore discarded), the number of messages stored in v_g is at most $n-g+1$, $g \in [1,n]$. □

Theorem 6 *The amount of storage required by a vertex's probe queue is $O((n \times \log_2 n))$ bits.*

Proof. The *stop conditions* prevent a vertex to transmit more than once the same probe to its neighbor vertices. Since a vertex has $n-1$ incoming edges, no more than $n-1$ replicas of the same probe may arrive at a vertex. Thus, the probe's counter requires $\log_2 n$ bits of space. Each probe requires $2 \times \log_2 n$ bits of space. The result follows directly from lemma 4. □

5 Correctness Issues

Let C be a cycle in some WFG. We will differentiate between two types of deadlocks. A deadlock is *stable* if, in the absence of deliberate resolution, it persists forever in the system. That is, resolving the deadlock would require deliberate intervention from the resolution algorithm. A deadlock is *unstable* otherwise. In other words, a deadlock is unstable if it could be spontaneously resolved. Examples of unstable deadlocks are: a) A transaction in a cycle C is forced to abort as a consequence of site failure. C represents an unstable deadlock since deliberate intervention was not required to resolve it. b) A transaction which participates in two cycles C_1 and C_2 is aborted by the deadlock resolution algorithm in order to resolve C_1. As a side-effect, C_2 will be resolved also. C_2 represents an unstable deadlock. c) When locking models using knowledge about states of objects are considered, an edge (i,j) representing a dependency between two transactions may be eliminated from the WFG as a side-effect of some third transaction committing and updating the object's state[9].

An important consequence of these observations is that, due to timing issues in distributed systems, a detected cycle may represent a deadlock which was already resolved (*phantom* deadlock). This will happen when an edge e in an unstable deadlock cycle is removed, and meanwhile, a message, having already traversed e, returns to its initiator. Unfortunately, determining the stability of deadlocks would require looking at the system's history of future events. Thus, when a cycle in a WFG is detected, it is not possible to decide in general whether or not this cycle represents an unstable deadlock. For this reason, we require that our algorithm systematically breaks detected cycles. Timing issues in distributed systems make the detection of phantom deadlocks a problem inherent to all distributed algorithms of deadlock detection and resolution, regardless of the lock model adopted. Our algorithm tries to minimize occurrences of phantom deadlocks by using antiprobes to promptly restore information about the WFG to a consistent state.

Theorem 7 *All stable deadlocks are detected. All unstable deadlocks are either detected or spontaneously resolved.*

Theorem 8 *All cycles reported represent either stable or unstable deadlocks.*

6 Conclusions

We have presented a general distributed algorithm to detect and resolve resource deadlocks in distributed systems. The algorithm can be used in conjunction with a large class of concurrency control algorithms — the class of locking schedulers. In particular, this deadlock detection and resolution algorithm applies to concurrency control algorithms which use the semantic lock model.

The algorithm proposed was shown to detect and resolve all the $O(n!)$ cycles present in a worst case WFG by transmitting

[9]Consider the following scenario allowed by the Peephole Scheduler [RB87]. The Holders Pool of object FifoQueue contains two operations enqueue(A), each belonging to transactions T_1 and T_2 respectively. The current state of the object is an empty queue. If transaction T_3 now requests operation dequeue( ), it will be placed in the FifoQueue's Conflict Pool, since operations enqueue(A) and dequeue( ) do not commute over an empty queue. Thus transaction T_3 depends on both T_1 and T_2. After T_2 commits the state of the queue is updated. It now contains one element, namely A. As the queue is not empty anymore, operations enqueue(A) and dequeue( ) do not conflict. Therefore, operation dequeue( ) is also placed in the Holders Pool. Note that as a consequence of T_2 committing, not only was the dependency between T_3 and T_2 eliminated, but the dependency between T_3 and T_1 was eliminated as well.

$O(n^3)$ messages of small constant size. A strategy to identify and eliminate messages bearing redundant information was devised and adopted. Compared to previous algorithms, this strategy contributed to a major reduction on the number of messages required to detect and resolve all cycles. The average performance of the algorithm has been shown to be $O(ne)$, where e represents the number of edges in the WFG [RBC87]. The algorithm's worst and average time complexities are lower than those of the best algorithms for deadlock detection and resolution previously reported.

References

[BBG86] C. Beeri, P.A. Bernstein, and N. Goodman. *A Model for Concurrency in Nested Transactions Systems.* Technical Report 86-03, Wang Inst. of Grad. Studies, 1986.

[BT87] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[CKST87] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley. A priority based probe algorithm for distributed deadlock detection and resolution. In *Proceedings 7th IEEE Int. Conf. on Distributed Computing Systems*, pages 162–168, September 1987.

[CM82] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proceedings ACM Symp. on PODC*, pages 157–164, 1982.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2), May 1983.

[Elm86] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Record*, 15(3), 1986.

[Gra78] J. N. Gray. *Operating Systems, An Advanced Course*, chapter Notes on Database Operating Systems, pages 398–481. Springer–Verlag, 1978.

[GS80] V. Gligor and S. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5), September 1980.

[HM83] L. M. Haas and C. Mohan. *A Distributed Deadlock Detection Algorithm for a Resource-Based System.* Technical Report RJ 3765, IBM Research Laboratory, 1983.

[Kor83] H. Korth. Locking primitives in a database system. *Journal of ACM*, 30(1), January 1983.

[MM79] D. Menasce and R. Muntz. Locking and deadlock detection in distributed data bases. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[Mos81] J. E. B. Moss. *Nested Transactions : An Approach to Reliable Distributed Computing.* Technical Report 260, MIT, April 1981.

[Obe82] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7:187–208, June 1982.

[RB87] M. Roesler and W. A. Burkhard. Concurrency control scheme for shared objects: a peephole approach based on semantics. In *Proc. 7th IEEE Int. Conf. on Distributed Computing Systems*, pages 224–231, September 1987.

[RBC87] M. Roesler, W. A. Burkhard, and K. Cooper. Efficient deadlock resolution for lock-based concurrency control schemes. In *Proc. 8th IEEE Int. Conf. on Distributed Computing Systems*, June 1988 (to appear).

[Sch84] P. M. Schwarz. *Transactions on Typed Objects.* PhD thesis, CMU, December 1984.

[SN85] M. K. Sinha and N. Natarajan. A priority based distributed deadlock detection algorithm. *ACM Transactions on Software Engineering*, SE-11(1), January 1985.

[Wei84] W. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, MIT, March 1984.