A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems

K. M. Chandy and J. Misra
Computer Sciences Department
University of Texas; Austin, TX 78712

ABSTRACT

This paper presents a distributed algorithm to detect deadlocks in distributed data bases. Features of this paper are (1) a formal model of the problem is presented, (2) the correctness of the algorithm is proved, i.e. we show that all true deadlocks will be detected and deadlocks will not be reported falsely, (3) no assumptions are made other than that messages are received correctly and in order and (4) the algorithm is simple.

1. INTRODUCTION

A great deal of effort has gone into developing a distributed algorithm for detecting resource deadlocks in distributed data bases (DDBs) [3,4,7]. In a September 1980 paper Gligor and Shattuck [4] state "Renewed interest in distributed systems has resulted in the publication of at least ten protocols for deadlock detection. However, few of these protocols are correct and fewer appear to be practical." In this paper we present a solution to this much-studied problem.

The following paragraph briefly reviews the literature on distributed deadlock detection. A model of deadock and an algorithm for deadlock detection suitable for message passing systems appears in [1]. The message model of deadock assumes that a process which is waiting to communicate with other processes, cannot proceed with its execution until it communicates with any one of the processes it is waiting for. The DDB model considered in this paper and in [3,4,6,7] assumes that a process can proceed only when it receives all resources that it is waiting for. The any/all difference in these models results in completely different algorithms for deadlock detection. Deadlock detection for a class of communicating finite state machines is considered in [5]. In this paper we are concerned with dynamic detection of deadlocks rather than with proving that specific communicating sequential machines do not deadlock, which is the problem considered in [5]. We consider the general class of problems appearing in [3,4,7]. In particular, the DDB model we use is derived from Menasce and Muntz, one of the first papers in this area. For a complete review of the literaure see [4,6,8].

The organization of this paper is as follows. Section 2 presents a simple formal model of a distributed system; this model is called the basic model. Section 3 describes an algorithm to detect deadlock in the basic model and presents its proof. Performance issues are found in section 4. A distributed algorithm by which a deadlocked process can determine the identity of other processes in the deadocked set is presented in section 5. In section 6 we review the distributed data base model presented by Menasce and Muntz [3], who were about the first to treat the problem. We then show how the basic model algorithm can be extended to solve the DDB problem.

2. THE BASIC MODEL

2.1. Goal of This Section

One of the difficulties with work in the area of DDBs is in describing the model of a DDB clearly and unambiguously. Since informal, operational models often result in ambiguity we have chosen to describe our model by axioms. Our proofs of correctness use these axioms; they do not rely on implicit assumptions about DDBs. The basic model which is described in this section is a simple, abstract model; its relevance to DDBs may not be clear immediately, but is discussed in detail in section 6. In the basic model, the state of computation is represented by a graph

* This work was supported in part by the Air Force Office of Scientific Research under grant AFOSR 81-0205 and the University Research Institute at The University of Texas.

called a wait-for graph [3] in which the vertices represent processes which may send and receive messages. We use a wait-for graph model because much of the earlier work is based on wait-for graphs. The graph also helps to distinguish the underlying DDB computation from the computation associated with deadlock detection.

The basic model is described by two sets of axioms: graph axioms and process axioms. Graph axioms specify how the wait-for graph may change over time. Graph axioms are concerned exclusively with the underlying DDB computation and not with the computation associated with deadlock detection. Process axioms are concerned with the relationship between the deadlock detection computation and the underlying DDB computation. The goal of this section is to present and motivate the graph and process axioms. The model is described and the graph axioms are motivated in section 2.2, the graph axioms are presented in 2.3 and the problem of distributed deadlock detection in the basic model is described in 2.4. The problem description relies on the graph axioms alone. The process axioms (section 2.5) are the rules which must be obeyed by any deadlock detection algorithm. An explanation for the process axioms is presented in section 2.6.

2.2. Model Description

A distributed system consists of a finite set of processes. A process is in one of two states: active or blocked. A process p_i is blocked if it is waiting for one or more processes to carry out some action (such as releasing resources needed by p_i). An active process is not waiting for any other process. When p_i needs p_j to carry out some action it sends a request to p_j; when p_j carries out the requested action it sends a reply to p_i. Only active processes may carry out actions for other processes, hence only active processes can send replies. The state of execution of all processes in a system is captured by a directed graph G called the wait-for graph. There is a one-to-one correspondence between vertices in G and processes in the system, with vertex v_i corresponding to process p_i. Edge (v_i,v_j) exists in G if and only if p_i has sent a request to p_j and has not yet received a reply.

Edge Colours: The edges in G are coloured grey, black or white. Edge (v_i,v_j) is:

grey: if p_i has sent a request to p_j which p_j has not received (yet).

black: if p_j has received a request from p_i and has not sent the corresponding reply to p_j.

white: if p_j has sent a reply to p_i which p_i has not received (yet).

We assume, for convenience, that there are vertices in the wait-for graph corresponding to terminated processes and to processes that have yet to be created. This allows us to ignore the addition and deletion of vertices in the wait-for graph. Of course, unborn and terminated processes cannot carry out actions for other processes or request actions from other processes.

We now describe the behavior of a network of processes in terms of coloured graphs. We use process p_i and vertex v_i, interchangeably.

2.3. Graph Axioms G1 - G4

G1: (Creation): A grey edge (v_i,v_j) may be created if edge (v_i,v_j) does not exist.

G2: (Blackening): A grey edge will turn black after an arbitrary, finite time.

G3: (Whitening): A black edge (v_i,v_j) may turn white only if v_j has no outgoing edges. (Only active processes may reply).

G4: (Deletion): A white edge will disappear after an arbitrary, finite time.

We next define the deadlock detection problem for the basic model and present the process axioms which must be followed by a deadlock detection algorithm.

2.4. The Deadlock Detection Problem in the Basic Model

A dark cycle, i.e. a cycle in which all edges are grey or black (some may be grey and others black), will persist forever because, it follows from the graph axioms that edges in a dark cycle cannot be whitened or deleted.

Problem PROB1: Construct a distributed algorithm by which a vertex v_i can detect if it is part of a dark cycle.

The algorithm by which v_i determines if it is part of a dark cycle is called a probe computation. In probe computations vertices send messages, called probes, to one another; probes are concerned with deadlock detection exclusively and are distinct from requests and replies. We now present axioms which describe how processes communicate; these axioms show the relationship between requests, replies and probes. We assume that messages (i.e. requests, replies and probes) are received in finite time in the order sent.

2.5. Process Axioms P1 - P4

An explanation of these axioms is given in section 2.6.

P1: If a probe is sent by v_i to v_j when edge (v_i,v_j) is grey, edge (v_i,v_j) will turn black sometime after this probe is sent and before it is received. If a probe from v_i is received by v_j when edge (v_i,v_j) is black then edge (v_i,v_j) existed and was dark (grey or black) at all times from the instant at which the probe was sent, to the instant the probe was received.

P2: If a probe is sent by v_j to v_i when (v_i,v_j) is white then (v_i,v_j) will disappear sometime after this probe is sent and before it is received.

P3: A vertex v_i can determine (locally) if there is an outgoing edge (v_i,v_j) to any v_j, though it cannot determine its colour (locally). A vertex v_j can determine (locally) if there is an incoming black edge (v_i,v_j), from any v_i.

P4: Every probe will be received in some arbitrary finite time after it is sent.

2.6. Explanation of the Process Axioms

P1: A probe sent by v_i to v_j when (v_i,v_j) is grey must have been sent after v_i sent v_j the request which caused grey edge (v_i,v_j) to be created. Since messages are received in the order sent, the request must be received by v_j (causing edge (v_i,v_j) to turn black) before the probe is received. The explanation for the second part of P1 is similar.

P2: A probe sent by v_j to v_i when edge (v_i,v_j) is white must have been sent after v_j sent v_i the reply which caused edge (v_i,v_j) to change colour from black to white. Since messages are received in the order sent, the reply must be received by v_i (causing edge (v_i,v_j) to disappear) before v_i receives the probe.

P3: An edge (v_i,v_j) can be created and deleted by v_i, and v_i alone; hence v_i can determine if it exists. An edge (v_i,v_j) is black only if v_j has received a request from v_i and it has not yet sent a corresponding reply. Hence v_j is aware of black edge (v_i,v_j).

P4: Basic rule of message communication.

This completes the description of the basic model. From now on, we will use only the axioms G1 - G4 and P1 - P4 to reason about the computation. Therefore, we do not use the terms "request," "reply," "resource," etc. hereafter.

3. AN ALGORITHM FOR THE BASIC MODEL

3.1. Goal of This Section

The goal of this section is to present a solution to the problem, PROB1, presented in section 2.4: construct a distributed algorithm (i.e. a probe computation,) by which a vertex can detect if it is part of a dark cycle. In this section we do not discuss the question of when a vertex should initiate such a computation, this question is considered in section 4. Section 3.2 introduces probe computations. Section 3.3 presents the desired properties of probe computations while section 3.4 presents the probe computation algorithm itself. Correctness proofs are found in section 3.5.

3.2. Introduction to Probe Computations

To determine whether it is on a dark cycle, a vertex v_i initiates a computation called a probe computation. Several vertices may initiate probe computations and the same vertex may initiate several probe computations. To distinguish each probe computation, the messages and variables used in the n-th computation initiated by vertex i are tagged (i,n). In the next paragraph we shall discuss one probe computation, say the (i,n)th. In the interests of brevity we shall not tag messages and variables in the following discussion with (i,n); the tag should be understood implicitly.

A vertex v_j will send at most one probe to any v_k in one probe computation. The probe is said to be meaningful if and only if edge (v_j,v_k) exists and is black at the time that v_k receives the probe. From P3, v_k can determine if a probe is meaningful.

3.3. Properties of a Probe Computation: QRP1, QRP2

A probe computation is designed to have the following two properties (proofs are in section 3.5):

QRP1: If the initiator of a probe computation is on a dark cycle when it initiates the probe computation then the initiator will eventually receive a meaningful probe.

QRP2: If the initiator of a probe computation receives a meaningful probe then it is on a black cycle at the time at which it receives the probe.

3.4. Algorithm for a Probe Computation

Algorithm for the initiator, v_i

A0: Send probes along all outgoing edges.

A1: Upon receiving the first meaningful probe declare that "v_i is on a black cycle."

Algorithm for a vertex v_j other than the initiator

A2: Upon receiving the first meaningful probe send probes on all outgoing edges.

Note: Each step A0,A1,A2 of the algorithm, once started must be completed before the process can send or receive other messages. Therefore the set of outgoing edges from process v_i in step A0 (and process v_j in step A2) do not change during the step.

3.5. Proof of Correctness of a Probe Computation

Theorem 1 (Property QRP1)

If the initiator is on a dark cycle when it initiates the probe computation then it will eventually get a meaningful probe.

Proof: Let the initiator v_i, be on a dark (and therefore permanent) cycle C. v_i will send a probe to its successor vertex v_j in C (i.e. edge (v_i,v_j) is in C), and from P1 this probe is meaningful; similarly v_j will send a meaningful probe to its successor in C, and so on, and thus every vertex on C (including the initiator) will eventually receive a meaningful probe.

Theorem 2 (Property QRP2)

If the initiator receives a meaningful probe then it is on a black cycle when this probe is received.

Proof: The initiator is the only vertex which can send a probe without having received a meaningful probe (follows from step A2 of the algorithm). Hence if the initiator v_i receives a meaningful probe, there exists a finite sequence $v_{j(0)},\ldots,v_{j(n)}$ where (1) $v_{j(0)} = v_{j(n)} = v_i$ and (2) $v_{j(k)}$ received a meaningful probe from $v_{j(k-1)}$ at time t_k, and $t(k-1) < t(k)$, $k = 1,..n-1$. Let e_k denote the edge $(v_{j(k-1)},v_{j(k)})$. We will prove the following assertion for all k, $1 \leq k \leq n$ by induction on k: at time t(k) the edges $e_1,e_2,\ldots,e_k$ are all black. The theorem then follows by setting k=n in this assertion. For k=1, the assertion follows from the definition of meaningful probe. Now inductively assume that $e_1,e_2,\ldots,e_K$, $K<n$, are all black at t(K); we will prove that $e_1,e_2,\ldots,e_{K+1}$ are all black at t(K+1). We first prove that e_{K+1} exists in the interval [t(K), t(K+1)] and that it is black at t(K+1). From step A2 of the algorithm, e_K existed at time t(K). From the definition of meaningful probe, e_{K+1} exists and is black at a later time t(K+1). From P1, e_{K+1} existed from the instant t' that $v_{j(K)}$ sent the probe to time t(K+1) at which $v_{j(K+1)}$ received the probe. Note $t(K) \leq t' \leq t(K+1)$. From the algorithm (see note below algorithm) this edge existed at all times from t(K) to t'. Hence e_{K+1} exists at all times from t(K) to t(K+1). We now prove that edges $e_0,\ldots,e_K$ existed and were black in this interval. This follows from the observation that if e_k exists in the interval [t(K),t(K+1)], then e_{k-1} exists and remains black in this interval (from induction hypothesis and G3), for k = 1,..,K. This proves the assertion.

We have shown that a probe computation satisfies the desired properties presented in section 3.3. In the next section we discuss issues related to performance.

4. PERFORMANCE ISSUES

4.1. Goal Of This Section

In section 3 we presented an algorithm (probe computation) by which a vertex can determine if it is on a dark cycle. In this section we will begin by discussing the question of when a vertex should initiate a probe computation (4.2). The volume of message traffic associated with probe computations and methods for reducing the number of probe computations are discussed in section 4.3.

4.2. When Should a Vertex Initiate a Probe Computation?

It is sufficient for any one vertex on a dark cycle to detect that it is deadlocked provided this vertex later informs all other vertices on the dark cycle that they are deadlocked too. An algorithm by which a deadlocked vertex informs other vertices that they too are deadlocked is presented in section 5. Therefore, in this section we need only be concerned with an initiation rule by which at least one vertex in a dark cycle will detect deadlock.

We employ the following initiation rule: A vertex v_i initiates a probe computation when any outgoing edge (v_i,v_j) is added to the wait-for graph. With this rule, if the addition of edge (v_i,v_j) creates a dark cycle in the wait-for graph, then v_i will detect that it is on a dark cycle, and hence deadlocked. Rules which yield better performance are treated in the next section.

4.3. Performance Aspects of the Algorithm

Recall that to distinguish probe computations initiated by different vertices, and by the same vertex at different times we tag the n-th probe computation initiated by v_i with (i,n), i.e. all probes and variables associated with that computation are tagged (i,n). If probe computation (i,n) is initiated, all probe computations (i,k) with k<n may be ignored. Therefore, every vertex need only keep track of one, (the latest) probe computation initiated by each vertex. Hence every process must keep track of N probe computations where N is the number of vertices in the graph. For a given probe computation, a vertex sends only one probe on any outgoing edge. Hence, there can be at most N^2 probes in a single probe computation.

The number of probe computations initiated can be reduced by having a vertex initiate a probe computation only if an outgoing edge (v_i,v_j) has been in existence continuously for some time T, where T is a performance parameter. If edge (v_i,v_j) is deleted before T time units have elapsed then v_i has avoided initiating a probe computation. Issues related to determining the optimum value of T are found in [6]. The basic tradeoff is that if T is too small too many probe computations are initiated and if T is too large the time taken to detect deadlock (which is at least T) is too large.

5. PROPAGATING WAIT-FOR GRAPH INFORMATION TO DEADLOCKED VERTICES

5.1. Goal of This Section

A distributed algorithm by which a vertex can determine all permanent black paths leading from it is presented in this section; the permanent black paths form the deadlocked portion of the wait-for graph, and determining the edges and vertices in the deadlocked portion of the graph is useful in breaking deadlocks. The question of how deadlocks should be broken is not treated here; the reader is encouraged to read [3,6].

5.2. Computation to Determine the Wait-For Graph (WFGD Computation)

Messages in a WFGD computation consist of sets of edges. A message M sent to a vertex v_j is a set containing only edges on permanent black paths (i.e. paths all of whose edges are black and are guaranteed to remain black) from v_j. Each vertex v_j has a local variable S_j, which is the set of edges (that v_j is aware of) on permanent black paths leading from v_j. Initially S_j is empty, for all j. After the initiator v_i of a probe computation receives a meaningful probe, it declares that it is on a black cycle and thereafter sends a message $M = \{(v_j,v_i)\}$ to every vertex v_j if edge (v_j,v_i) is black. Since v_i is on a black cycle (v_j,v_i) must be permanently black. On receiving a message M, v_j sets $S_j = S_j$ M and thereafter sends M' where $M' = \{(v_k,v_j)\}$ S_j to every vertex v_k where (v_k,v_j) is black, if it has not already sent the same message, M' to v_k. Since M only contains edges on permanent, black paths leading from v_j, M' only contains edges on permanent black paths leading from v_k. It is evident that every vertex will determine all permanent black paths leading from it in finite time. A WFGD computation will cease because a vertex never sends the same message (set of edges) twice to another vertex.

6. THE DISTRIBUTED DATA BASE PROBLEM

6.1. Goal of This Section

We have presented and proved an algorithm for the basic model. We now show how the algorithm for the basic model can be extended to handle the distributed data base model considered in [3,4]. We first review the Menasce-Muntz DDB model (section 6.2) and point out the differences between the DDB model and the basic model in section 6.3. An abstraction of the DDB model, based on coloured graphs is found in section 6.4. Probe computations for the DDB model are introduced in section 6.5. The algorithm to solve the DDB deadlock problem is presented in section 6.6, and a performance issue specific to DDBs is discussed in section 6.7.

6.2. An Introduction to the DDB Deadlock Problem

A DDB is implemented by N computers $S_1,\ldots,S_N$. There is a local operating system or controller C_j at each computer S_j to schedule processes, manage resources and carry out communications. There are M transactions $T_1,\ldots,T_M$ running on the DDB. A transaction is implemented by a collection of processes with at most one process per computer. Each process is labeled with a tuple (T_i,S_j) where T_i is the identity of the transaction that the process belongs to and S_j is the computer on which the process runs. The tuple (T_i,S_j) uniquely identifies a process.

A controller C_j sends a message to a process (T_i,S_j) by putting the message in the process's memory area and scheduling the process. A process (T_i,S_j) sends a message to its controller C_j by putting the message in the controller's memory area and then returning control to the controller. A process (T_i,S_j) communicates directly only with its own controller C_j. Controllers may send messages to one another. Messages sent by any controller C_j to any controller C_m will be received by C_m in finite time and in the order sent by C_j.

At some stage in a transaction's computation it may need to "lock" resources (such as files). There are different kinds of locks (read locks and write locks for instance) but the details regarding locks and locking protocols are not relevant to the problem described here; the reader is referred to [3,6]. When a process (T_i,S_j) needs a resource it sends a request to its controller C_j. If C_j manages the resource it may accede to the process's request immediately or the process may have to wait to acquire the requested resource. If the requested resource is managed by some other controller C_m, then C_j transmits the request on to process (T_i,S_m) via controller C_m; the request is now made locally by process (T_i,S_m) to its own controller C_m. When (T_i,S_m) acquires the requested resource from C_m, it sends a message to (T_i,S_j) (via C_m and C_j) stating that the requested resource has been acquired. (T_i,S_j) may now proceed with its computation. When processes in a transaction T_i no longer need a resource managed by controller C_m, they communicate with process (T_i,S_m) who is responsible for releasing the resource to C_m.

A process cannot proceed with its computation unless it acquires every resource that it requests. Thus a process is blocked permanently from proceeding with computation if it never acquires a requested resource. We assume that if a single transaction runs by itself in the DDB it will terminate in finite time and eventually release all resources. When two or more transactions run in parallel, deadock may arise because each transaction may be blocked needing resources held by other transactions. The problem is to construct an algorithm to detect deadlock.

6.3. Difference Between the DDB and Basic Model

In the basic model, one process directly requests another to carry out some action. In the DDB model, a process may not be aware of other processes; furthermore, a process only communicates directly with its controller. Hence, the primary difference between the basic model and the DDB model is that in the basic model a process determines locally which processes to (request actions from and) wait for, whereas in the DDB model the controller at each computer determines the process waiting behavior at that computer.

6.4. A Graph Model of DDB Deadlock

As in the basic model there is a one-to-one correspondence between processes in the system and vertices in the wait-for graph G. There is an edge in G from a process (T_i,S_j) to another process (T_k,S_j) at the same computer S_j, if controller C_j has a request from (T_i,S_j) for resources held by (T_k,S_j). Such an edge in G (which is incident on vertices corresponding to processes at a single controller) is called an intra-controller edge. There is an edge in G from a process (T_i,S_j) to another process (T_i,S_m) within the same transaction T_i (but at a different computer) if (T_i,S_j) is waiting for a message that it has acquired a resource managed by C_m; such an edge is called an inter-controller edge.

The colour of an inter-controller edge from (T_i,S_j) to (T_i,S_m) is grey, black or white, where the colours have the same meaning as in the basic model, i.e. it is grey, if (T_i,S_j) has requested a resource managed by C_m and C_m has not received the request yet; it turns black when C_m receives the request and white when C_m gives the requested resource to (T_i,S_m) (at which point it sends a message to (T_i,S_j) saying that the resource has been acquired). Since the existence of an intra-controller edge $((T_i,S_j),(T_k,S_j))$ depends only upon controller C_j's awareness that (T_i,S_j) requires a resource held by (T_k,S_j), and since C_j schedules (T_i,S_j) and (T_k,S_j) we may assume that all intra-controller edges are black. The formal graph model is described by the following axioms.

Graph Axioms G1-G6 for a DDB

Axioms regarding intra-controller edges

G1: A black intra-controller edge $((T_i,S_j),(T_k,S_j))$ may be added to G if none exists.

G2: A black intra-controller edge $((T_i,S_j),(T_k,S_j))$ may be deleted if (T_k,S_j) has no outgoing edges.

Axioms regarding inter-controller edges (analogous to the basic model)

G3: A grey inter-controller edge $((T_i,S_j),(T_i,S_m))$ may be added to G if the edge does not exist.

G4: A grey inter-controller edge will turn black in an arbitrary, finite time.

G5: A black inter-controller edge $((T_i,S_j),(T_i,S_m))$ can turn white if (T_i,S_m) has no outgoing edges.

G6: A white inter-controller edge will disappear in arbitrary, finite time.

A dark cycle in G will persist forever. The problem is to construct a distributed algorithm by which a controller C_j can determine if one of its processes (T_i,S_j) is on a dark cycle. The algorithm must satisfy the following process axioms which are analogous to the process axioms for the basic model.

P1: If a probe is sent by C_j to C_m when edge $((T_i,S_j),(T_i,S_m))$ is grey, then the edge will turn black some time after the probe is sent and before it is received. If a probe from C_j is received by C_m when the edge is black then the edge existed and was dark from the instant that the probe was sent to the instant that the probe was received.

P2: If a probe is sent by C_m to C_j when edge $((T_i,S_j),(T_i,S_m))$ is white, then the edge will disappear some time after this probe is sent and before it is received.

P3: A controller C_j can determine locally if there is an outgoing edge from any of its processes (T_i,S_j) to any other process; however, it cannot determine locally the colour of inter-controller edges outgoing from (T_i,S_j). A controller C_m can determine locally if there is an incoming black edge to any of its processes (T_i,S_m).

P4: A probe sent along any edge is received correctly and within finite time.

6.5. The Probe Computation in the DDB Model

A probe computation in a DDB model is exactly the same as in the basic model except that instead of processes, controllers send probes to one another. Instead of having a process (T_i,S_j) send a probe to another process (T_k,S_j) at the same computer S_j, controller C_j merely labels (T_k,S_j) as having received a meaningful probe. As in the basic model, the n-th probe computation initiated by controller C_j is tagged (j,n), i.e. all labels and probes are tagged (j,n). If there is an outgoing inter-controller edge $((T_i,S_j),(T_i,S_m))$ from a labeled process (T_i,S_j), then C_j sends a probe to C_m. This probe carries with it the tag (j,n) as well as the identity of the edge $((T_i, S_j),(T_i,S_m))$; this probe is said to be sent along edge $((T_i,S_j),(T_i,S_m))$. This probe, from controller C_j to another controller C_m, is said to be meaningful if the edge $((T_i,S_j),(T_i,S_m))$ exists and is black at the time at which C_m receives the probe. We now describe a single probe computation, say the (j,n)th. Though the tag (j,n) does not appear explicitly in the description, it should be assumed.

6.6. Algorithm for a Probe Computation

Algorithm initiated by C_j to determine if process (T_i,S_j) is on a dark cycle

A0: Label all processes (T_k,S_j) reachable from process (T_i,S_j) along intra-controller edges. If (T_i,S_j) is labelled, then declare that it is on a black cycle of intra-controller edges. Otherwise, if there is an inter-controller edge from a labelled process (T_a,S_j) to any process (T_a,S_b) then send a probe to C_b along edge $((T_a,S_j),(T_a,S_b))$.

A1: Upon receiving a meaningful probe along any inter-controller edge $((T_p,S_m),(T_p,S_j))$, label (T_p,S_j) and all processes reachable from (T_p,S_j) along intra-controller edges. If (T_i,S_j) is labelled, declare that (T_i,S_j) is on a black cycle.

Algorithm for a Controller C_m Other Than the Initiator

A2: Upon receiving a meaningful probe along an inter-controller edge directed towards a process (T_i,S_m) label (T_i,S_m) and all processes reachable from (T_i,S_m) along intra-controller edges. If there is an inter-controller edge from a labelled process (T_a,S_m) to any process (T_a,S_b) then send a probe to C_b along edge $((T_a,S_m),(T_a,S_b))$ if such a probe has not already been sent.

Note: Each step A0,A1,A2 of the algorithm, once started, must be completed before the controller can send or receive other messages. Hence the intra-controller edges and outgoing inter-controller edges from processes in S_j cannot change during steps A0 and A1. The analogous condition holds for S_m in step A2.

The proof of the algorithm for the DDB model is exactly the same as for the basic model. The performance issues discussed for the basic model also apply to the DDB model. However, there is one performance issue which arises in the DDB model which does not arise in the basic model. The algorithm presented above requires a controller C_j to initiate a separate probe computation for each of its constituent processes (T_i,S_j). We now show how the number of probe computations can be reduced.

6.7. How to Avoid Initiating a Separate Probe Computation for Each Process

When a controller C_j wishes to determine if any of its constituent processes are on dark cycles it first determines if there is a cycle along intra-controller edges alone. If there is no intra-controller cycle, then any cycle through any constituent process (T_i,S_j) must include an inter-controller edge directed towards a constituent process (T_k,S_j). Hence, it is sufficient for a controller to initiate separate probe computations for processes with incoming (black) inter-controller edges. Hence, when a controller wishes to determine if any of its processes are deadlocked it initiates Q separate probe computations where Q is the number of constituent processes with incoming, black, inter-controller edges.

7. SUMMARY

We have presented a solution to the much-studied problem of deadlock detection in distributed data base systems. A formal model based on coloured graphs was used. For purposes of exposition, the problem was introduced in two stages: in the first stage, a simple model, called the basic model was introduced and in the second stage the Menasce-Muntz distributed data base model was discussed. Our algorithm was proved correct. Details regarding the different modes of resource locking and other features of distributed data bases have not been included here. The reader is referred to [3,6].

A great deal of work remains to be done on evaluating the performance of the algorithm and on developing algorithms for different types of distributed systems.

8. ACKNOWLEDGEMENT

Our work in this general area resulted from reading a seminal paper by Dijkstra and Scholten on termination detection [2] and by later discussions with them. Virgil Gligor showed us that the DDB problem, though apparently simple, was non-trivial and interesting, and led us to the sizable body of work on the subject.

9. REFERENCES

1. Chandy, K. M., J. Misra and L. Haas, "A Distributed Deadlock Detection Algorithm and Its Correctness Proof," submitted to the Communications of the ACM.

2. Dijkstra, E. W. D. and C. S. Scholten, "Termination Detection for Diffusing Computations," Information Processing Letters, 11, 1, August 1980, pp 1-4.

3. Menasce, Daniel and Richard Muntz, "Locking and Deadlock Detection in Distributed Data Bases," IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979.

4. Gligor, Virgil and Susan H. Shattuck, "On Deadlock Detection in Distributed Systems," IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980.

5. Yu, Yao-Tin and Mohamed Gouda, "Deadlock Detection for a Class of Communicating Finite State Machines," TR-193, Computer Sciences Depatment, University of Texas, Austin, Texas 78712.

6. Gray, J. N., "Notes on Data Base Operating Systems," in Operating Systems and Advanced Course, Berlin, Heidelberg: Springer-Verlag, 1978, Ch. 3.F, pp. 394-481.

7. Obermarck, Ron, "Global Deadlock Detection Algorithm," RJ2845, IBM Research Laboratory, San Jose, California 95193, June 1980.

8. Mohan, C., "Distributed Data Base Management - Progress, Problems, Some Proposals and Future Directions," Computer Sciences Department, Working Paper WP-7802, University of Texas, Austin, Texas 78712, May 1979.