Locking and Deadlock Detection in Distributed Data Bases

DANIEL A. MENASCE AND RICHARD R. MUNTZ, MEMBER, IEEE

Abstract—This paper describes two protocols for the detection of deadlocks in distributed data bases—a hierarchically organized one and a distributed one. A graph model which depicts the state of execution of all transactions in the system is used by both protocols. A cycle in this graph is a necessary and sufficient condition for a deadlock to exist. Nevertheless, neither protocol requires that the global graph be built and maintained in order for deadlocks to be detected. In the case of the hierarchical protocol, the communications cost can be optimized if the topology of the hierarachy is appropriately chosen.

Index Terms—Data bases, deadlock detection, distributed data bases, graph theory.

I. Introduction

THIS WORK is concerned with issues of locking and deadlock detection mechanisms in distributed data bases. The problem of system deadlocks in multiprogramming and multiprocessing systems has received considerable attention in the literature and is well understood [1]-[3]. There are three approaches to the treatment of deadlocks: deadlock prevention, deadlock avoidance, and deadlock detection and resolution.

Deadlock prevention requires that all the resources be acquired at once by a transaction. This requirement cannot always be satisfied in a data base environment since the resource needs of a transaction may be data dependent and not precisely known at the start of the transaction. Therefore, it would be necessary for a transaction to acquire all possible resources required, thereby decreasing system concurrency.

Deadlock avoidance requires some advance knowledge of the resource usage of transactions in order to determine at each point in time whether there is a valid sequence of actions of the already initiated but not yet completed transactions such that all of them can be run to completion. Again, this approach is not practical in distributed data bases since the necessary advance information to avoid deadlocks is either absent or is distributed enough to render inefficient any attempt to avoid deadlocks.

Deadlock detection can be done by searching for cycles in a "state graph" [3]. A method for the detection and resolution of deadlocks in a centralized data-base system was presented in [4]. In distributed data bases, however, it is not efficient to maintain a global state graph for the whole system. Two methods for detecting deadlocks in distributed data bases, which do not require that a global graph be built and maintained, are presented in this paper—one hierarchical and one distributed. An outline of the proof that both protocols will detect all existing deadlocks is given here. For the case of the hierarchical protocol, the problem of establishing the hierarchy in a way such that the cost of using the protocol is minimized is introduced.

Manuscript received February 6, 1978; revised November 1, 1978. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-77-C-0211. The work of D. A. Menasce was supported in part by the Conselho Nacional de Desenvolvimento Cientifico e Technologico, CNPq, Brazil. This is an expanded version of a paper presented at the Third Berkeley Workshop on Distributed Data Bases.

The authors are with the Department of Computer Science, University of California, Los Angeles, CA 90024.

II. Formal Model of Transaction Processing

This section introduces the necessary notation and formalism upon which we base the locking protocols and deadlock detection mechanisms presented in this paper. The data base is considered to be distributed among n sites, S1, S2, $\cdots$, Sn, of a computer network. Users interact with the data base via transactions. A *transaction* is a sequence of actions which can be either read, write, lock, or unlock operations. Transactions are assumed to be two phase, i.e., once an unlock operation is issued no other lock can be requested by the transaction. As shown in [6], this is necessary to preserve the data-base consistency. If the actions of a transaction involve data at a single site, the transaction is called *local*, as opposed to a *distributed* transaction which involves resources at several sites. We assume that distributed transactions are implemented as a collection of processes which act on behalf of the transaction. Those processes are called *transaction incarnations.* There may be one or more incarnations of the same transaction at each participating site. A transaction incarnation is responsible among other things for 1) acquiring, using, and releasing resources local to the site at which it is executing, as needed by the transaction, and 2) exchanging messages with remote incarnations of the same transaction for purposes of cooperating with incarnations located at foreign sites. Note that this model of a transaction execution is general enough to accommodate other models.

A transaction can be in two different states, namely, *active* and *blocked.* A transaction is blocked if its execution cannot proceed because a needed resource is being held by another transaction, and the transaction is active otherwise. We now introduce a graphic model which depicts the state of execution of all transactions in the system. This model is in the form of a graph called the *transaction_wait_for* (TWF) graph. The nodes of this graph are associated with transaction in-

0098-5589/79/0500-0195$00.75 © 1979 IEEE

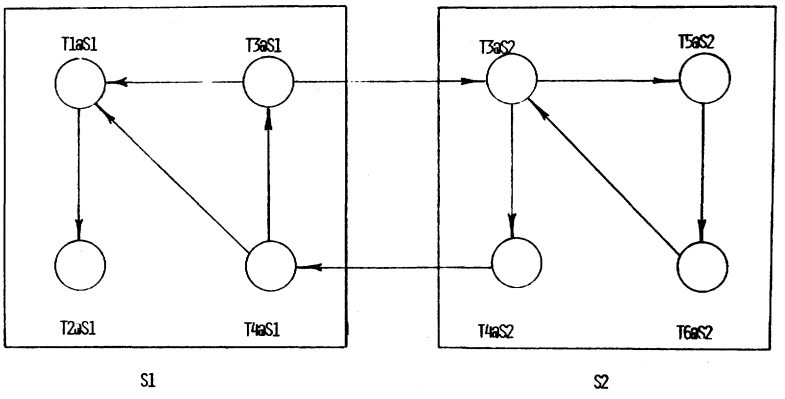

Fig. 1. A transaction_wait_for graph for a network with two sites S1 and S2.

carnations and are labeled by the pair (transaction_name, site_name). Note that labeling transaction incarnations with the pair (transaction name, site name) provides unique global names for these nodes if there is at most one transaction incarnation per site per transaction. If more than one incarnation is to be allowed, then distinct local names should be assigned for them. Since this distinction is irrelevant for the forthcoming discussion, we will assume that there is only one incarnation of any transaction per site.

1) There is a directed arc from node (Ti, S) to node (Tj, S) if the incarnation of transaction Ti at site S is blocked and waiting for the incarnation of transaction Tj to release a resource needed by Ti. (Ti, S) is said to be in "resource wait" for (Tj, S).

2) There is a directed arc from node (T, Si) to node (T, Sj) if the incarnation of transaction T at site Si is blocked and waiting for a message from the incarnation of T at site Sj. (T, Si) is said to be in "message wait" for (T, Sj).

It can be easily seen that the existence of a cycle in the transaction_wait_for graph is a necessary and sufficient condition for a deadlock to occur.

Fig. 1 shows an example of a transaction_wait_for graph for a network with two sites S1 and S2. This graph shows two deadlock cycles. One of them is a local deadlock since it involves only incarnations of transactions at site S2 and, therefore, only resources local to S2. The other deadlock cycle spans both sites and is an example of what we call a global deadlock.

III. Deadlock Detection Approaches

Deadlock detection involves building and maintaining the transaction_wait_for graph and searching for the existence of cycles in the graph. The graph has to be updated every time that a transaction changes state, either from active to blocked or vice versa. It should be noted, however, that new cycles can only potentially arise when a transaction is blocked. Deadlock resolution involves the selection of one or more transactions to be preempted in order for the cycle to be broken. The criteria used in this selection should try to minimize the penalty of preemption by any suitable metric. Examples of deadlock resolution methods are: preempt the transaction which owns less resources, preempt the transaction with the smallest rollback cost, preempt the transaction with the longest expected time to complete, and preempt any transaction in the cycle. The examination of such criteria is beyond the scope of this paper.

A centralized approach for deadlock detection in distributed data bases was suggested by Gray in [5]. In this approach there is a centralized deadlock detector which is responsible for constructing a global graph equivalent to the transaction_wait_for graph considered here. The central deadlock detector builds the graph out of information received from all the participating sites in the network.

While the centralized method may be practical and efficient for local networks, it may impose fairly large communications cost in geographically distributed systems. This observation stems from the fact that the central deadlock detector may be located very "far" from some of the sites in the network. As an example, we certainly do not want deadlocks which only involve resources located at some host computers located in Southern California to be detected at the East Coast. The hierarchical approach to deadlock detection presented in the next section lends itself to an optimization by which deadlocks can be detected by a site which is located as "close" as possible to the sites involved in the cycle.

IV. A Hierarchically Organized Locking and Deadlock Detection

Let the data base (DB) be partitioned into a set of subdata bases BDi's such that DB is the union of all the DBi's, and DBi and DBj are disjoint for $i \neq j$. The locking and deadlock detection mechanism presented here has as its core a hierarchy of lock controllers which interact in a way to be explained in this section. First, we are going to distinguish between the controllers which are at the bottommost level of the hierarchy, called *leaf controllers* or *Lk's*, and the *nonleaf controllers* or *NLK's*.

A leaf controller Lki is assigned to each subdata base DBi. In the example shown in Fig. 2, we have three leaf controllers LK1, LK2, and LK3 and two nonleaf controllers NLK0 and NLK1.

Each leaf controller Lki maintains a transaction_wait_for

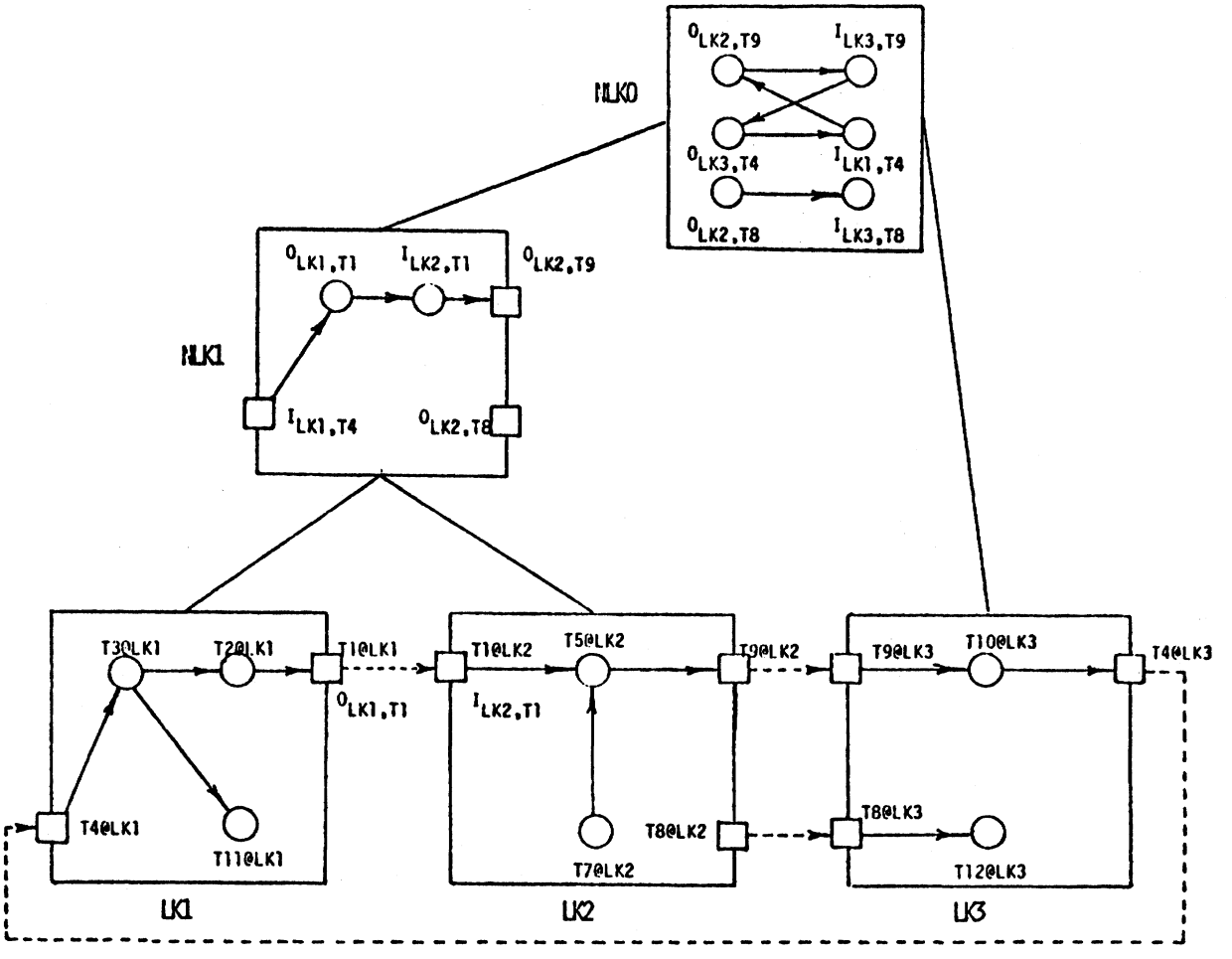

Fig. 2. A hierarchy of lock controllers.

graph, TWF(LKi). This graph contains all the nodes of the global TWF associated with transaction incarnations local to LKi. In addition, two special types of nodes called *output-port nodes* and *input-port nodes* are introduced in the TWF of a leaf controller. These nodes are associated with the arcs of the global TWF which join incarnations in two distinct controllers and are defined as follows.

1) A node in the TWF graph of LKi is called an *output port* and denoted O(LKi, T) if the global TWF contains an outgoing arc from an incarnation of transaction T local to LKi into a nonlocal incarnation of T.

2) A node in the transaction_wait_for graph of LKi is called an *input port* and denoted I(LKi, T) in TWF(LKi) if in the global TWF there is an incoming arc into an incarnation of transaction T local to LKi from a nonlocal incarnation of T.

Note that labels assigned to input and output ports are unique since there is only one transaction incarnation per transaction per site. In the example of Fig. 2, the output port O(LK1, T1) and the input port I(LK2, T1) correspond to an arc in the global TWF from T1@LK1 (the incarnation of Tw at LK1) to T1@LK2. The dashed lines indicate arcs in the global TWF. These arcs are represented explicitly at the upper levels of the hierarchy as will be explained below.

Nonleaf controllers maintain a graph called *input-output-ports (IOP)* graph. Nodes of an IOP are associated with input and output ports of leaf controllers. We will refer to them as *i-nodes* and *o-nodes*, respectively. Some of the i-nodes may be themselves input ports for the IOP, and some of the o-nodes may be output ports for the IOP. The IOP for controller NLKi, denoted IOP(NLKi), is defined by the following rules.

1) Arcs from i-nodes can go only to o-nodes and vice versa.

2) There is an arc from o-node Oa to i-node Ib if Oa is an output port of a leaf controller in the subtree rooted at NLKi and Ib is a corresponding input port of another leaf controller in the same subtree. In the example of Fig. 2, there is an arc from O(LK1, T1) to I(LK2, T1) in the IOP of NLK1 since O(LK1, T1) is an output port of LK1 and I(LK2, T1) is its corresponding input port in LK2. LK1 and LK2 are in the subtree rooted by NLK1.

3) There is an arc from the i-node Ia to o-node Ob in IOP(NLKi) if there is a path from an input port Ia to an output port Ob of a son of NLKi. In the example of Fig. 2 there is an arc from I(LK1, T4) to O(LK2, T9) in NLK0 since in NLK1 there is a path between I(LK1, T4) and O(LK2, T9).

4) An input (output) port of IOP(NLKi) is also an input (output) port of a leaf controller in the subtree rooted by

NLKi. In the example of Fig. 2 the input port of IOP(NLK1) is also an input port of LK1.

V. Hierarchical Protocol—A Description

Before we describe the protocol, let us define the *lowest common ancestor* between controllers K1, K2, · · · , K*n*, denoted LCA(K1, K2, · · · , K*n*), as the common ancestor between them at the lowest level in the hierarchy (the root is at the highest level). Rules 1) through 3) below describe the hierarchical protocol.

Rule 1

(*Transaction incarnation T requests a local resource*): The requested resource R is in the same subdata base as the transaction incarnation T. Let LKi be the controller for resource R.

Rule 1.1: The resource cannot be granted. Let {T1, T2, · · · , Tk} be the set of transactions which currently hold resource R. Add an arc from (T, LKi) to (Tj, LKi) for j = 1 to k. Check the transaction_wait_for graph at LKi for the existence of cycles.

Rule 1.1a: If cycles were formed, then one or more local deadlocks have been detected and an appropriate action is required for deadlock resolution.

Rule 1.1b: The addition of the arcs mentioned in Rule 1.1 may have created one or more paths between input and output ports of LKi. For each such path, send the (input port, output port) pair which delimits the path to the father of LKi.

Rule 2

(*Transaction T requests a nonlocal resource*): The requested resource R is in a different subdata base from the previously requested resource. Therefore, it has to be acquired by an incarnation of T local to R. Let LKi be the controller for the previously requested resource and let LKi be the controller for resource R. The incarnation of T at LKi becomes blocked and waiting for a message from the incarnation of T at LKj. The node (T, LKi) is now an output port of the transaction_wait_for graph at LKi, and the node (T, LKj) is an input port of the transaction_wait_for graph at LKj.

Rule 2.1: An arc from O(LKi, T) to I(LKj, T) is created in the IOP graph of the lowest common ancestor between LKi and LKj.

Rule 2.2: An o-node labeled O(LKi, T) is added to the IOP graph of each controller in the path between LKi and LOA(LKi, LKj). Each such o-node is also an output port of the corresponding IOP graph.

Rule 2.3: An i-node labeled I(LKj, T) is added to the IOP graph of each controller in the path between LKj and LOA(LKi, LKj). Each such i-node is also an input port of the corresponding IOP graph.

The protocol followed by a nonleaf controller NLKi is described by Rule 3 below.

Rule 3

An arc is added to IOP(NLKi).

Rule 3.1: If a cycle is generated by the addition of the new arc, then a global deadlock has been detected and an appropriate action is required to resolve it.

Rule 3.2: If no cycle was generated, check whether any input-output port connection has been generated in IOP(NLKi) and report the endpoints of any such connections to the father of NLKi.

We have not yet mentioned how lock releases are reflected in the appropriate graphs in the hierarchy. Let each controller (LK or NLK) maintain a list of the i-o paths (i.e., the paths which connect input ports to output ports) in its graph. A possible representation for this list could be in the form of a bit matrix where each row corresponds to an i-o path, and each column is associated with an arc in the graph. The value '1' in the i, j entry of this matrix indicates that arc j is in the i-o path i. An unlock operation causes an arc (maybe more) to be deleted from a TWF graph of a leaf controller. All the i-o paths (if any) which contained this arc are broken. This may be reported to the father of the LK. There, the arcs which represented the broken i-o paths will be used to find which i-o paths were broken in the nonleaf controller. This propagation continues up in the hierarchy until the deletion of an arc from a graph does not cause any i-o path to be broken.

The method described above for deadlock detection requires that nonleaf controllers be kept up-to-date continuously. Other variations can be used when appropriate. For example, the information concerning connections between input ports and output ports can be sent periodically. For a sufficiently long period this would reduce the amount of traffic generated but may result in a deadlock existing for too long a period of time. Another method which is intermediate to continuous and periodic deadlock detection is to report connections between input and output ports after they have persisted longer than some threshold. Since, if a deadlock occurs the cycle persists until it is detected and relieved, this method will detect a deadlock after some delay. It appears that a judicious choice for the threshold can result in less traffic being generated than with continuous checking and less delay in detecting a deadlock than with periodic checking.

VI. Hierarchical Protocol—Plausibility Argument

The hierarchical protocol has the following properties:

1) deadlocks which involve resources of a single subdata base DBi are detected by the formation of a cycle in the TWF of the leaf controller associated to DBi;
2) deadlocks which involve resources controlled by the leaf controllers LK1, LK2, · · · , LKi are detected by the formation of a cycle in the IOP graph of the nonleaf controller which is the lowest common ancestor between the LKi's.

Property *a* follows directly from Rule 1 of the protocol. The validity of property *b* is not so straightforward and will be shown to hold by the following theorems.

Let us introduce some notation first. Let the arcs which connect an i-node to an o-node in an input-output-ports graph be called i → o arcs and let o → i arcs be the ones which connect o-nodes to i-nodes. Let *tree*(K) be the subtree of the hierarchy rooted at controller K. Let us now extend the notation TWF(K) to indicate the subgraph of the global TWF obtained by considering only the resources controlled by all the LK's in tree(K).

Let us show that if a cycle is detected in the IOP graph of a nonleaf controller, there is a deadlock.

Theorem 1: If there is a cycle in the input-output-ports graph of a nonleaf controller NLKi then there is a deadlock.

Proof: We need to prove that if there is a cycle C in the IOP graph of a nonleaf controller NLKi, there is an associated cycle C′ in TWF(NLKi). Let us show how to construct the cycle C′. Let C be a cycle in IOP(NLKi). For the purpose of this construction, let us label each i → o arc (Ij, Oj) in C with the label Kj if Kj is an immediate son of NLKi such that the creation of an input-output-port connection in Kj caused the arc (Ij, Oj) to be created in IOP(NLKi)–see Rules 1.1b and 3.2 of the protocol. Arcs of the type o → i in IOP(NLKi) will not be labeled. Using the notation (a, b, c) to indicate an arc labeled *c* from node *a* to node *b* let the cycle C be C = (I1, O1, K1), (O1, I2, -), (I2, O2, K2), · · · , (In, On, Kn), (On, I1, -).

The repeated application of Operation 1 below will transform the cycle C into a cycle in which all the labels indicate leaf controllers.

Operation 1: If Kj in (Ij, Oj, Kj) is a nonleaf controller, replace (Ij, Oj, Kj) by a path connecting Ij to Oj in IOP(Kj).

After this transformation, there is a path in the TWF of an LK in tree(NLKi) associated with each i → o arc in C. There is also an arc between incarnations of transactions in the TWF's of distinct LK's associated with each o → i arc in C. More precisely, for each i → o arc (Ii, Oi, Ki) in C there is a path in TWF(Ki) between the input port Ii and the output port Oi of TWF(Ki). Now, by Rule 2.1 of the protocol each o → i arc (Oi, Ii + 1, -) in C connects two incarnations of the same transaction. These incarnations are local to leaf controllers in tree(NLKi). Finally, the sum of all the paths and arcs thus obtained defines the cycle C′ in TWF(NLKi).

In order to conclude the proof we must show that the arcs in the cycle C′ defined above exist simultaneously, and, therefore, C′ is a deadlock cycle. Assume not. Then there is at least a pair of arcs Ti → Tj and Tm → Tn which did not appear simultaneously in C′ and such that there is a path from Tj to Tm in TWF(NLKi). Consider first the case in which Ti → Tj existed (i.e., appeared and disappeared) before Tm → Tn. Then, transaction Tj released the resource it was holding (which was needed by Ti). This implies that all the transactions in the path from Tj to Tm in TWF(NLKi) must have had released at least one resource also. Since transactions are assumed to be two phase, then none of them (including Tm) can issue any further lock requests. Therefore, the arc Tm → Tn cannot exist. This contradicts our assumption and shows that it is not possible for Ti → Tj to have existed before Tm → Tn. Consider now the case in which Tm → Tn existed before Ti → Tj. This case is perfectly possible as long as these two arcs are not part of the same cycle; otherwise, one would have to conclude that every arc in the cycle existed before itself. Therefore, all the arcs in C′ coexist in TWF(NLKi), and C′ represents a deadlock cycle, and the theorem is proved.

We want to show now that given a cycle in the global TWF there is a corresponding cycle in one of the controllers in the hierarchy. In order to state this property more precisely, some definitions are in order. Let the graph G′ be called an *arc condensation* or simply *condensation* of the graph G if the node set of G′ is a subset of the node set of G, and if (v1, v*n*) is an arc in G′ then there is a path from node v1 to node v*n* in the graph G.

Theorem 2: Given a cycle G in the global TWF there is an arc condensation of C in one and only one controller in the hierarchy. This controller is the lowest common ancestor of all the controllers which manage the resources in the cycle C.

Proof: Let C be a cycle in the global TWF which involves resources controlled by leaf controllers LK1, LK2, · · · , LK*n*. Each of the *n* leaf controllers only knows the portion of the cycle which contains resources local to the controller. Let us consider an initial condensation of the cycle C obtained by substituting every path in the TWF from an input port to an output port of a leaf controller by a single arc connecting these ports. For the purpose of this proof, let us introduce a representation for a cycle which clearly illustrates how the knowledge about portions of the cycle is distributed throughout several controllers in the hierarchy. So, let a cycle be represented by a sequence of labeled arcs (IO, OO, KO), (I1, O1, K1), · · · , (Ij, Oj, Kj), · · · , (In - 1, On - 1, Kn - 1) where (Ij, Oj, Kj) indicates that there is a connection between the input port Ij and the output port Oj in the graph of the controller Kj. Also, the output port Oj is connected to the input port Ij + 1(mod *n*). Let *father*(*K*) be the father of controller K in the hierarchy. The repeated application of operations 1 and 2 below gives us the desired condensed cycle C.

Operation 1: If all the labels in C are the same, then stop. Substitute every maximal path P in C composed solely of arcs labeled with sons of a common controller K by a single arc having as endpoints the endpoints of P and having as label the common father controller K. This operation is illustrated in Fig. 3 and can be thought of as a short circuit between consecutive brothers. Repeat Operation 1 until all the labels in C are different and then do Operation 2.

Operation 2: Find all the arcs in C which have as label a controller whose level in the hierarchy is the lowest of all the controllers which appear as labels in C. Substitute the label Ki in each of these arcs by father(Ki). Do Operation 1. Operation 2 is illustrated in Fig. 4.

In order to show the validity of Operation 1, let us refer to Fig. 3 and consider the following observations.

1) There is an arc from Oi to Ii + 1 for i = 1 to (l - 1) in IOP(K), since the controller K is the lowest common ancestor between K1, K2, · · · , Kl. See Rule 2.1 of the protocol.

2) There is an arc from Ii to Oi, for i = 1 to 1 in IOP(K), since every time a connection between an input and an output port is created it is propagated to its father. See Rules 1.1a and 3.2 of the protocol.

3) Since the controller K0 is not a son of controller K, the input port I1 is also an input port of K which is in the path between K0 and LCA(K1, K0). See Rule 2.3 of the protocol. The same observation applied to controller Kl + 1 and the output port O1 (see Rule 2.2 of the protocol).

The reader should notice that Operation 1 preserves the lowest common ancestor between all the controllers involved in the cycle.

The validity of Operation 2 follows from observations 2) and 3) above. Notice that the choice of a controller at the lowest possible level to substitute during operation 2 preserves the

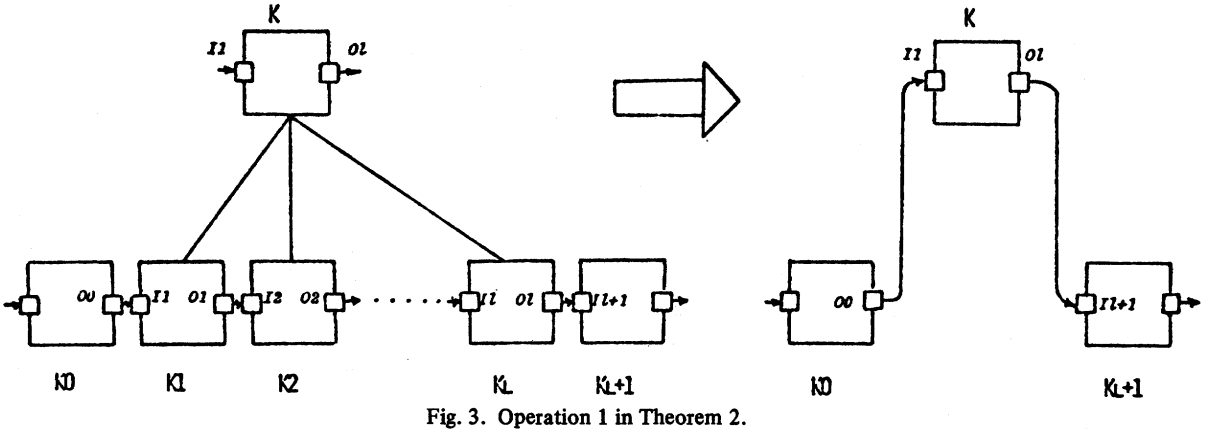

Fig. 3. Operation 1 in Theorem 2.

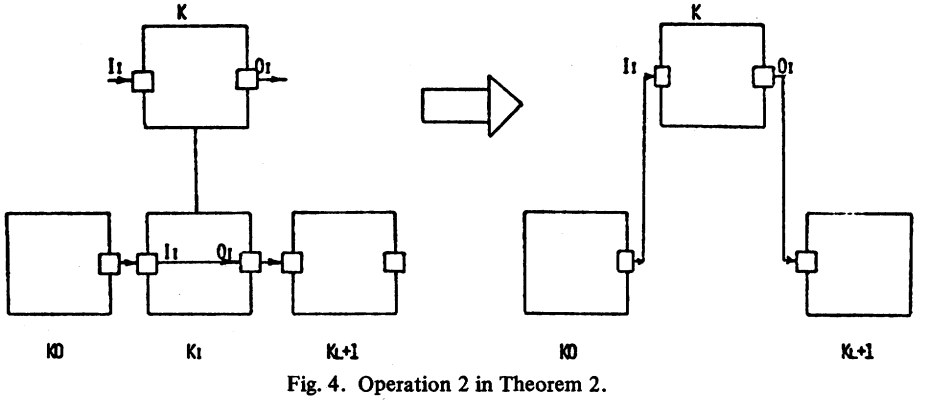

Fig. 4. Operation 2 in Theorem 2.

lowest common ancestor between the controllers involved in the cycle. The reader can easily convince himself that this must be the case.

Therefore, when all the labels are the same we have a representation of a condensed version of the original cycle. The common label is the name of the controller where the cycle is completely represented. Since operations 1 and 2 preserve the lowest common ancestor between the controllers involved in the cycle, the original cycle will be represented in a condensed way in LCA(LK1, LK2, $\cdots$, LKn).

Theorems 1 and 2 together give us a necessary and sufficient condition for a deadlock to be detected in a distributed data base involving resource at different sites. This result will be stated as the following theorem.

Theorem 3: A necessary and sufficient condition for a deadlock involving resources in controllers LK1, LK2, $\cdots$, LKn to exist is that there is a cycle in the IOP graph of ICA(LK1, LK2, $\cdots$, LKn). This cycle is a condensation of the corresponding cycle in the global TWF.

VII. Deadlock Resolution

As pointed out in an earlier section, we are not going to examine in this paper the criteria involved in optimal deadlock resolution since this is mainly a policy issue. This section discusses, however, the mechanisms which are necessary to allow the implementation of any such policy. The reader may have noticed that the condensed cycle in the IOP graph contains less information than the corresponding cycle in the TWF graph. In particular, not all the transactions which participate in the cycle in the TWF graph appear in the IOP graph.

One way to compensate for this loss of information is to require that, whenever an i-o arc is received by an NLK, the name of the controller which generated the arc be stored with the arc. Then, when an NLK detects a deadlock cycle, it can send down the tree to its appropriate sons a message which will continue to propagate down (through the appropriate sons) until it reaches the leaves of the tree. At this point the LK's can report directly to the NLK which detected the deadlock all the necessary information to implement the desired policy for deadlock resolution.

Notice that the additional messages necessary to support the above described mechanism do not substantially increase the total communications cost of the protocol since they must only be sent when deadlocks are detected and not during normal operation.

Another, less flexible alternative is to select the transaction to be preempted from those which appear in the IOP graph only. While no additional messages are required here it is likely that a nonoptimal choice will be taken in resolving the deadlock.

VIII. Hierarachy Establishment

So far we have assumed the existence of the hierarchy without considering how it is established in the first place. The performance of the hierarchical protocol, in terms of the overhead message traffic incurred by it, can be minimized if the hierarchy is appropriately chosen. This choice should consider the pattern of the DB traffic with respect to the locality of access to a controller or group of controllers. For instance, assume that one is able to identify groups or clusters of leaf controllers such that a high percentage of the DB traffic involves controllers in the same cluster while very little traffic is of the intercluster type. One possibility here would be to assign nonleaf controllers to each cluster and try to further cluster the non-leaf controllers or put all of them together under the same root.

The problem in general can be stated as follows. Given a set of leaf controllers assigned to the nodes of a computer network, given the DB traffic pattern, and given the cost of sending messages between every pair of nodes in the network, find a hierarchy which minimizes the total cost incurred in using the protocol.

There are clearly some heuristic rules which, if applied to a given hierarchy, result in another hierarchy of less cost. The general optimization problem, however, is the subject of current research effort.

IX. A Distributed Locking and Deadlock Detection Protocol

A locking protocol which uses distributed control and a distributed deadlock detection mechanism is presented here. Before we describe the protocol, some definitions are in order. Each data-base site controls a set of resources. Transactions request resources by sending their requests to the controller of the resource. Each controller is responsible for

1) processing lock and lock release requests for local resources. Requests may originate from any node in the network.
2) building a simplified version of the transaction_wait_for graph and detecting deadlocks. The TWF maintained by a controller is a subgraph of the global TWF.

For the purpose of stating this algorithm, we will use a simplified version of the transaction_wait_for graph. The reader should be aware that this version is a redefinition of the graph used in the section on the hierarchical protocol although the same name for the graph is retained. In this version, there is no notion of transaction incarnations. Nodes are associated with transactions, and there is a directed arc from transaction T' to transaction T'' if T' is blocked and must wait for T'' to release a resource (not necessarily a resource needed by T') before T' is able to proceed.

Some definitions are in order. A *nonblocked* transaction is a node in the transaction_wait_for graph with no outgoing arcs or a *sink* node. Let us now define *blocking set(T)* as the set of all nonblocked transactions which can be reached by following a directed path in the TWF graph starting at the node associated with transaction T. This is the set of transactions which are ultimately blocking transaction T. The pair (T, T') is said to be a blocking pair of T if T' is in blocking_set(T).

The execution of a transaction can be described as follows. A transaction has a site of origin which is the site where the transaction entered the system. The transaction starts running at this site, performing local operations until operations on nonlocal data are necessary. Then, a lock request in the appropriate mode is built and sent to the controller for the requested resource. This controller will either accept or reject the lock, sending the reply to the site of origin of the transaction. If there are multiple copies of data, lock requests have to be sent to all controllers which keep a copy of the data.

X. Distributed Protocol—A Description

Let *Sorig(T)* be the site of origin of transaction T, and let TWF(K) be the TWF graph at site Sk. The protocol is described by Rules 1 and 2 given below as carried out by controller Sk. Let T be a transaction requesting resource R.

Rule 1

The resource R cannot be granted to T because it is being held by transactions T1, T2, $\cdots$, Tk. Add an arc from transaction T to each of the transactions in the set {T1, T2, $\cdots$, Tk}. If the addition of these arcs caused a cycle to be formed in TWF(K), then a deadlock was detected, and an appropriate action is required for its resolution. For each transaction T' in blocking_set(T), send the blocking pair (T, T') to Sorig(T) if Sorig(T) $\neq$ Sk and to Sorig(T') if Sorig(T') $\neq$ Sk.

Rule 2

A blocking pair (T, T') is received. Add an arc from T to T' in TWF(K). If a cycle was formed, then a deadlock exists, and it must be resolved by an appropriate action. If T' is blocked and Sorig(T) $\neq$ Sk, then for each transaction T'' in the blocking_set(T), send the blocking pair (T, T'') to Sorig(T'') if Sorig(T'') $\neq$ Sk.

Some comments are in order.

1) The arcs of the transaction_wait_for graph considered here may represent one of two types of relationships between transactions, namely, a *direct* wait and an *indirect* wait. Transaction T1 is said to be waiting directly on T2 if the resource needed by T1 for its continued execution is being held by T2. A transaction T1 is said to be waiting indirectly on Tk if there is a set of transactions T2, T3, $\cdots$, Tk - 1 such that Ti is waiting directly on Ti + 1 for i = 1 to k - 1.

2) From the previous observation it can be seen that cycles in a TWF graph may be a condensation (in the sense defined in the section on the hierarchical protocol) of the cycle that would exist in the global transaction_wait_for graph for the whole system.

XI. Distributed Protocol—Plausibility Argument

That the protocol described in the previous section is able to detect all deadlocks is shown in the following theorem.

Theorem 4: The above described protocol detects all possible deadlocks.

Proof: In order to show this result we will consider a global deadlock cycle, as shown in Fig. 5, and we will show

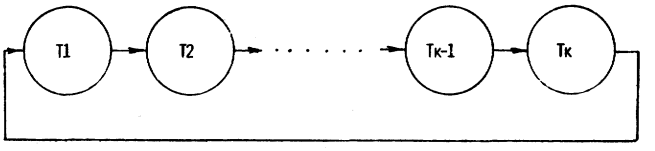

Fig. 5. Deadlock cycle involving transactions T1, T2, · · · , Tk.

that this cycle will appear in the TWF of the site of origin of at least one of the transaction in the cycle.

There are many orderings of resource requests that can lead to the same deadlock cycle. Each time a request is rejected, the newly formed blocking pairs will increase the knowledge that some controllers have about the global graph. Therefore, the more advance knowledge obtained when a request is rejected the more rapidly the deadlock will be detected. The ordering of resource requests used in this proof is such that controllers will get the minimum possible knowledge of the rest of the graph when a request is rejected. The reader should not have any problem in convincing himself that if the theorem holds for this ordering, then it holds for any other. The chosen ordering is the following. Initially, transactions T1 through Tk - 1 are blocked and each of the controllers at the site of origin of these transactions have the knowledge of a single transaction ahead in the cycle. At this point the transaction_wait_for graph at the site of origin of each transaction is shown as follows.

Sorig(T1): T1 → T2
Sorig(T2): T2 → T3
:
Sorig(Tk - 1): Tk - 1 → Tk.

Now, when Tk makes a request and is blocked by T1, the blocking pair (Tk, T2) will be sent to Sorig(T2) where a new blocking pair (Tk, T3) is formed and sent to Sorig(T3). There the blocking pair (Tk, T4) is formed and sent to Sorig(T4) and so on until the blocking pair (Tk, Tk - 1) reaches Sorig(Tk - 1). This causes the arc from Tk to Tk - 1 to be added to the TWF graph at that site. Since this graph already contained an arc from Tk - 1 to Tk, a cycle is formed and the deadlock is detected.

XII. Conclusion

This paper presents two solutions to the problem of deadlock detection in distributed data bases. The first solution consists of a hierarchy of lock controllers and is intended to achieve better performance, in terms of communications cost, than a centralized approach. For this purpose, the hierarchy should be established in a way such that deadlocks can be detected by a site which is located as "close" as possible to the sites involved in the deadlock. The problem of finding a hierarchy of lock controllers which minimizes the total cost incurred in using this protocol is the subject of current research. The design of a distributed protocol for deadlock detection was motivated by the desire to support reliable operation in environments subject to failures.

Both protocols use a graph model to depict the current state of execution of all transactions in the system. A cycle in this graph is a necessary and sufficient condition for a deadlock to exist. The protocols presented here do not require that a global graph be built and maintained in order for deadlocks to be detected. An outline of the proof of the correct operation of the proposed protocols is included in the paper.

The communications cost involved in using the solutions presented here depends on several factors such as data-base traffic pattern, distance between participating sites, hierarchy topology (for the hierarchical protocol only), and others. A detailed analysis of the performance characteristics of these protocols is the subject of further work.

References

[1] A. N. Haberman, "Prevention of system deadlocks," *Commun. Ass. Comput. Mach.*, vol. 12, no. 7, pp. 373-377, July 1969.
[2] A. Shoshani and E. G. Coffman, "Sequencing tasks in multiprocess, multiple resource systems to avoid deadlocks," in *Proc. 11th Annual Symp. on Switching and Automata Theory*, Oct. 1970, pp. 225-233.
[3] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surveys*, vol. 3, no. 2, pp. 67-78, June 1971.
[4] P. F. King and A. J. Collmeyer, "Database sharing–An efficient mechanism for supporting concurrent processes," in *Proc. AFIPS Nat. Comput. Conf.*, 1973.
[5] J. N. Gray, "Notes on database operating systems," in *Operating Systems and Advanced Course.* Berlin, Heidelberg: Springer-Verlag, 1978, ch. 3.F, pp. 394-481.
[6] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The motions of consistency and predicate locks in a database system," *Commun. Ass. Comput. Mach.*, vol. 19, no. 11, Nov. 1976.

Daniel A. Menasce, photograph and biography not available at the time of publication.

Richard R. Muntz (S'62-S'65-M'69), photograph and biography not available at the time of publication.