THE IMPACT OF NETWORK CHARACTERISTICS
ON THE SELECTION OF A DEADLOCK DETECTION ALGORITHM
FOR DISTRIBUTED DATABASES

by

Pamela Dorr Fuller Daniel

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

S. Antoy E. Haddad

May, 1989

Blacksburg, Virginia
THE IMPACT OF NETWORK CHARACTERISTICS ON THE SELECTION OF A DEADLOCK DETECTION ALGORITHM FOR DISTRIBUTED DATABASES

by
Pamela Dorr Fuller Daniel

Committee Chairman: Csaba J. Egyhazy
Computer Science

(ABSTRACT)

Much attention has been focused on the problem of deadlock detection in distributed databases, resulting in the publication of numerous algorithms to accomplish this function. The algorithms published to date differ greatly in many respects: timing, location, information collection, and basic approach. The emphasis of this research has been on theory and proof of correctness, rather than on practical application. Relatively few attempts have been made to implement the algorithms.

The impact of the characteristics of the underlying database management system, transaction model, and communications network upon the effectiveness and performance of the proposed deadlock detection algorithms has largely been ignored. It is the intent of this study to examine more closely the interaction between a deadlock detection algorithm and one aspect of the environment in which it is implemented: namely, the communications network.
## Contents

1 Introduction ............................................. 1

2 Research Methods and Approach ....................... 2

3 Discussion of the Interaction of a Deadlock Detection Algorithm With the Networking Environment .... 4
   3.1 Basic Concepts in Distributed Deadlock Detection .......... 4
   3.2 Communications Profiles of Selected Deadlock Detection Algorithms ... 7
      3.2.1 Path-Pushing Algorithms .................................. 8
      3.2.2 Edge-Chasing Algorithms .................................. 33
      3.2.3 Diffusing Computation Algorithms ....................... 38
      3.2.4 Global State Detection Algorithms ....................... 41
   3.3 Environmental Alternatives Provided by Network Architecture ... 43
      3.3.1 Alternatives for Network Class and Topology .......... 45
      3.3.2 Alternatives for Network Services ....................... 52
      3.3.3 Influence of Network Alternatives on Distributed Applications 57
   3.4 Impact of the Network Environment on the Selection of a Deadlock Detection Algorithm .......... 61
      3.4.1 Environmental Preferences of Individual Deadlock Detection Algorithms .......... 61
      3.4.2 Selection of a Deadlock Detection Algorithm Given a Network Environment .......... 75

4 Conclusions about the Influence of the Network Environment on the Selection of a Distributed Deadlock Detection Algorithm ........ 85

5 Summary .................................................. 87
1 Introduction

The problem of deadlock among concurrent processes plays an important role in many areas of computing, including data base management systems. As a possible solution to this problem, one that can maximize both resource usage and process throughput, deadlock detection has been the subject of considerable research. The implementation of distributed deadlock detection, involving the coordination of agents at multiple sites, is significantly more complex than that of a centralized solution.

The degree of difficulty involved in implementing distributed deadlock detection is reflected in the number and diversity of the algorithms that have been proposed. The focus of these proposals has been the development of algorithms that will successfully handle any interprocess dependency with minimal overhead in terms of number of messages sent and internal storage requirements. The vast majority of the research to date has been purely theoretical in nature, and the algorithms have generally been presented as independent processes rather than as one component of an integrated system. Few of these algorithms have actually been implemented, and little effort has been made to assess the effect of working environment on their operation.

The purpose of this study is to examine more closely the interaction of distributed deadlock detection algorithms with the networking environment, to determine what aspects of that environment are most likely to affect the performance of the algorithms, and to assess the impact of specific changes in those areas on selected deadlock detection algorithms.
2 Research Methods and Approach

A four part investigation into the relationship of distributed deadlock detection algorithms with the supporting network environment was undertaken in pursuit of this subject. The initial effort concentrated on the identification of a small set of specific algorithms that would represent the full variety of approaches to distributed deadlock detection featured in the current literature. The timely publication of Knapp’s 1987 survey [13] of deadlock detection algorithms for distributed databases, reviewing research published between 1980 and 1986 as well as some of the older classic contributions to the area, provided a fortuitous starting point. The algorithms discussed in that survey, augmented by a few more recent proposals, formed the research base for this project.

A detailed analysis of each of the selected algorithms was then conducted, to identify the ways in which those algorithms interact with the communications environment. Particular attention was paid to the degree of distribution of the deadlock detection functions, the location of the nodes involved in the communications, the frequency and duration of the transmissions, and, the quantity and format of the individual messages used.

The resulting profile of communications requirements provided valuable insight into the aspects of network architecture having the greatest impact on the functionality of the selected algorithms. The Defense Advanced Research Projects Agency (DARPA) Internet model of network architecture was selected for this step because
of its widespread use and popularity in the United States. Two broad areas of impact were distinguished, and a range of configurational options available in each of those areas identified.

Finally, a representative subset of the deadlock detection algorithms originally analyzed were selected for a detailed examination to determine the specific operational effects of different network configurations. The environmental preferences of the individual algorithms and the relative benefits of implementing each algorithm in a specific environment were studied.
3 Discussion of the Interaction of a Deadlock Detection Algorithm With the Networking Environment

3.1 Basic Concepts in Distributed Deadlock Detection

The possibility of a deadlock arises in a system any time two or more concurrent processes can compete for the use of two or more resources. The following sequence of events is typical of that producing a deadlock:

- transaction $t_1$ is granted use of resource $r_1$
- transaction $t_2$ is granted use of resource $r_2$
- transaction $t_1$ requests use of resource $r_2$
- transaction $t_2$ requests use of resource $r_1$

Neither transaction will ever be granted the use of the additional resource it has requested and, consequently, neither transaction will run to completion.

There are currently three approaches to the problem of deadlock: *deadlock prevention*, in which all resources needed by a process must be reserved in advance and allocated to that process for the duration of its execution; *deadlock avoidance*, in which all but one of the processes involved in a conflicting request for a resource are immediately aborted, to be restarted at some future time; and, *deadlock detection*, in which processes are allowed to wait for others to release desired resources, the outstanding resource requests are periodically examined to determine whether a deadlock exists, and one of the processes involved in a deadlock is aborted to resolve the deadlock situation.
It is obvious that deadlock detection is the approach most likely to maximize the concurrent usage of resources in a system; and there is a corresponding degree of interest in the development of robust procedures for its implementation. The distribution of processes and/or resources among multiple physical machines, whether geographically proximate or distant, adds a significant degree of difficulty to the problem. Multi-site as well as local dependencies must be identified, and the information obtained from inherently asynchronous processors must be synchronized to construct a consistent global view of the system state. The difference between the centralized and distributed solutions is exemplified by an article entitled “Deadlock Detection Is Really Cheap” [11], containing an algorithm that can be used to detect the existence of a deadlock and identify the processes involved. The author makes one critical implicit assumption in his assessment: that the state information concerning all processes in the system has already been collected and integrated. The extent of published material describing and criticizing various approaches to precisely this aspect of the problem is evidence of its difficulty.

The asynchronous nature of distributed processing gives rise to the two major problems encountered in the implementation of any distributed deadlock detection algorithm: the detection of a deadlock that does not exist; and, conversely, the failure to detect a deadlock that does exist. Either occurrence may significantly reduce the degree of concurrency in the system: by the needless abortion and restart of processes due to false deadlock detection and, worse, by the prolonged unavailability of resources due to the failure to detect a deadlock situation. The importance of these problems is
reflected in the definition of the two properties of correctness for a deadlock detection algorithm [13]: progress, meaning that if a deadlock exists, it must be detected; and, safety, meaning that if a deadlock is detected, it must exist.

The major tool used in deadlock detection is the wait-for-graph, which models the outstanding resource requests of transactions in the system. Some algorithms, particularly earlier contributions to the field, construct and analyze explicit wait-for-graphs; the properties of an implicit global graph are reflected in the design of other algorithms, although the graph itself is never actually constructed. A wait-for-graph consists of a node (vertex) for each transaction in the system, and a directed arc between two nodes for each outstanding resource request. An actively processing transaction is represented in the graph by a node with no outgoing edges. An active transaction becomes blocked when it makes a resource request that cannot be immediately granted because that resource is held in a conflicting mode by another transaction. The requesting (blocked) transaction is then said to be dependent on the holding transaction; the dependency is represented by the addition to the graph of a directed edge from the node representing the former to that representing the latter. The edge is removed from the graph when the requested resource is granted to the blocked transaction.

A path is the sequence of nodes encountered by traversing the directed edges in the wait-for-graph. A transaction is directly dependent on the transaction represented by the node following it in a path, and transitively dependent on the transactions
represented by the remaining nodes. A cycle is a path in which the initial and terminal nodes are the same. The presence of a cycle in the wait-for-graph indicates that a deadlock exists in the system; the nodes in the cycle represent the transactions involved in that deadlock.

A distinction is sometimes made between dependency on local and remote resources in a distributed system. A transaction requesting a remote resource may spawn a transaction agent at the remote site, and suspend its local operation until a message has been received from that agent indicating that the request has been granted. In such a system, a transaction blocked on a local resource is said to be in resource wait, while a transaction waiting for communication from a remote agent is said to be in message wait. The wait-for-graph in such systems will often adopt a method of distinguishing between nodes involved in local and remote dependencies, using a specific node to represent the remote world, for example.

3.2 Communications Profiles of Selected Deadlock Detection Algorithms

The organization of this discussion reflects the second of two classification schemes proposed by Knapp [13] in his survey of recent developments in distributed deadlock detection. Knapp observed that deadlock detection algorithms use four approaches to the collection of process state information: path-pushing, edge-chasing, diffusing computations, and global state detection. The characteristics of each of the four classes of algorithms will be discussed separately, beginning with an introduction to the class and followed by a more detailed discussion of the specific requirements of
several individual algorithms belonging to that class.

3.2.1 Path-Pushing Algorithms

The *path-pushing* algorithms transmit portions of an explicit wait-for-graph from site to site, gradually integrating the subgraphs to develop a global graph, which can then be used to determine whether or not a deadlock exists. Specifically, each site periodically develops a wait-for-graph and transmits it to a set of remote sites. Depending on the degree of distribution incorporated in the deadlock detection algorithm, this set of remote sites may include all of the remote sites with which the processes in the graph interact or a single remote site that has been designated as the controller for the site in question. The receiving sites integrate the transmitted graphs with their own graphs and transmit the result to a similarly constructed set of remote sites. At each iteration, the composite wait-for-graph is analyzed to determine whether or not a deadlock exists, as evidenced by a cycle in the graph. Eventually, one or more sites will receive enough composite information to construct a graph showing a distributed (multi-site) deadlock.

The algorithms in this category present by far the widest variety of approaches to the problem of deadlock detection; in terms of the topology of the distributed database, the location and functionality of the processes involved in deadlock detection, and the type, destination, and frequency of the messages to be sent. For this reason, a relatively large number of algorithms in this category have been selected for analysis. The selected algorithms are discussed in chronological order, beginning with
two algorithms proposed by Menasce and Muntz [14] in one of the original papers published in the field. These algorithms are followed by another classic contribution by Obermarck [18], which has the distinction of having actually been implemented in a distributed database system, System R [13]. The algorithms proposed by Ho and Ramamoorthy [10] are included because of their contribution to the problem of constructing an internally consistent global view of a system in spite of the asynchronous nature of and communications delays inherent in distributed processing. Similarly, the work of Wuu and Bernstein [25] is important because it identifies a method for avoiding false deadlock detection regardless of whether or not the underlying transaction processing follows a two-phase locking protocol. The discussion of path-pushing algorithms is concluded with two recent contributions, by Elmagarmid, Sheth, and Liu [7] and Elmagarmid, Soundararajan, and Liu [8], in which process and resource state tables are transferred dynamically between sites as dependencies are added and removed from the wait-for-graph.

Menasce and Muntz, Hierarchical Model

One of the earliest proposals for distributed deadlock detection using path-pushing algorithms was made by Menasce and Muntz [14]. They outline two algorithms: one based on a hierarchical model and the other on a fully distributed model. In the hierarchical model, the database is distributed across a number of sites, at each of which a controller is responsible for the maintenance of a disjoint partition of the database. The connectivity of these sites is unspecified. Superimposed on this database model is
the deadlock detection model, consisting of a hierarchy of controllers dedicated to the construction of transaction wait-for-graphs and the detection of cycles within them. The tree formed by these deadlock detection controllers is not required to be either balanced or binary. The controllers at the deepest level of the hierarchy, the leaf controllers, are coincident with the database controllers. The remaining controllers, the non-leaf controllers, construct incrementally more global views of the system.

Both the leaf and non-leaf controllers maintain explicit wait-for-graphs, as shown in Figure 1. Two special types of nodes represent transactions involved in remote dependencies: an output-port represents a local transaction in message wait, while an input-port represents the local agent of a remote transaction. A leaf controller's graph contains a node for each local transaction and agent, while a non-leaf controller's graph contains a node for each output-port and input-port of its subordinate controllers.

Two types of inter-controller deadlock detection messages are directed up the tree, from the leaf controllers toward the root. When a transaction becomes blocked on a local resource, the leaf controller updates its wait-for-graph and then identifies the set of new (input-port, output-port) transitive dependencies created by the addition of those arcs. This set of dependencies is sent to the leaf controller's parent, which will update its graph accordingly. While not explicitly covered in the algorithm, one can assume that a similar process is undergone when a previously blocked transaction is reactivated by the grant of a local resource.
Figure 1: Menasce and Muntz, Hierarchical Model
The second type of message is generated when a transaction becomes blocked on a remote resource and consists of one \((output-port, input-port)\) pair representing the transaction making the request and its agent at the site of the resource, respectively. The leaf controller at each site will send this pair to its parent; both parents will update their wait-for-graphs and forward the message until it reaches the root of the subtree in which they are both located, called the lowest common ancestor of the two leaf controllers. A similar sequence of events is expected to occur when the remote resource request is granted and the inter-site dependency is removed.

The basic algorithm assumes that the wait-for-graphs at non-leaf controllers are updated continuously. Inter-controller traffic can be cut substantially, however, by delaying the transmission of update messages, either until a dependency reaches some existence threshold or until a specific period of time has passed since the last update. The effects of such a delay are not completely fortuitous, however. As the period between updates is extended, the likelihood of additional \((input-port, output-port)\) dependencies becoming eligible for transmission is increased, resulting in a longer message than would originally have been the case. At some point, the length of an update message may exceed the amount that can be transmitted in an individual packet, actually increasing the number of messages that must be sent.

Obviously, the design of the hierarchy has a significant effect on the amount and cost of detection-oriented message traffic. The authors summarize this problem as follows:

Given a set of leaf controllers, assigned to the nodes of a computer network,
given the [database] 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 [deadlock detection] protocol.

The emphasis in the discussion is on the identification of groups of controllers with a high degree of interaction and the assignment of a parent controller to each such group. The effect of altering the topology on the timeliness and cost of message transmission could be profound.

Menasce and Muntz, Distributed Model

The distributed model proposed by Menasce and Muntz [14] is quite different from the hierarchical model, both in organization and functionality. In this model, there are no distinctions between database and deadlock detection controllers; the controller at each site is responsible for the construction and analysis of a transaction wait-for-graph as well as the control and allocation of the resources located at that site. Both the actual resource request/grant messages and the deadlock detection messages are communicated directly between the sites involved.

The wait-for-graph contains both direct and indirect (transitive) dependencies involving local transactions. When a resource request cannot be satisfied, a direct dependency is added to the local wait-for-graph at the sites of both the blocking and the blocked transactions. The addition of that dependency may identify one or more transitive dependencies, which must be forwarded to the sites of the transactions involved. An example is shown in Figure 2.
Figure 2: Menasce and Muntz, Distributed Model
The situation is exacerbated if resource sharing is supported by the underlying database model. When a resource is held exclusively, only one direct dependency is created when a transaction becomes blocked. If that resource is shared by a number of transactions, a direct dependency must be created for each of those transactions. The total number of messages generated when a transaction becomes blocked can increase dramatically in this situation.

This algorithm involves the transmission of a number of very short messages between sites whenever a transaction becomes blocked. In some cases, if a dependency is created that involves multiple transactions at a single site, one message can be used to convey several dependencies. In general, however, each dependency will be transmitted by a separate message. As with the hierarchical algorithm, the transmitted messages contain updates to an existing wait-for-graph. These updates can be delayed according to some threshold criteria or periodicity at the expense of increasing the individual message length as well as delaying the detection of a deadlock, since updating the local wait-for-graph triggers the analysis of that graph for cycles.

Obermarck

In the algorithm proposed by Obermarck [18], the deadlock detection functionality is fully distributed among the network nodes, and the deadlock detection communications take place along the same communications links as the resource management messages. The network is assumed to be reliable. Distributed transactions are represented by agents at the sites of remote resources. The agents are identified in such
a way that they can be ordered; by \((\text{site}, \text{process})\) or \((\text{process}, \text{site})\), for example. The communications link between two agents of the same transaction is directed; one agent sends and the other receives messages across that link.

Each site maintains a wait-for-graph of all dependencies known by that site. Instead of identifying certain nodes as input-ports and output-ports, Obermarck's graph contains an explicit External node, labeled 'EX', which represents the unknown portion of the global wait-for-graph. The External node is considered to have the lowest identifier value of any node, ensuring that it appears first in any ordered list of nodes. The EX node may appear at the head or the tail of an arc in the graph.

The intersite deadlock detection messages are composed of Strings, which identify a potential deadlock cycle as an explicit path beginning and ending at the External node. For example, the cycle ‘EX \(\rightarrow t_1 \rightarrow t_2 \rightarrow EX\)’ would be represented by the String \((EX, t_1, t_2)\). It is immediately apparent that the messages employed by this algorithm will be substantially longer than those in the previous two models, where a path was represented by its endpoints rather than by naming each node explicitly.

In contrast to Menasce and Muntz's approach of continuously updating the wait-for-graphs maintained by the deadlock detection controllers, Obermarck proposes a distinct deadlock detection cycle, consisting of the following steps:

1. Build a local wait-for-graph. First construct a graph containing a node for each local transaction agent and an External node. Then, add to the graph all nodes and dependencies in the Strings received from other sites.

2. Analyze the graph to identify all cycles, both those consisting completely of specific transaction nodes and those containing the External node. Represent
each cycle by an ordered list of nodes, beginning with the node having the lowest identifier (which is the External node if it appears in the cycle).

3. Select one or more victims to break those cycles that do not contain the External node.

4. Each remaining cycle contains the External node. Select those cycles in which the identifier of the node on which the External node waits is greater than that of the node waiting for the External node. Construct a String for each of these cycles and transmit it to the site from which the last transaction in the String is waiting to receive.

5. Discard the wait-for-graph.

Several important points are made here. Obermarck's model may involve both the transmission of a large number of messages and the transmission of long deadlock detection messages. This is due to a combination of the following factors. First, the wait-for-graph is constructed anew for each deadlock detection cycle. This requires that all Strings be transmitted during each cycle, not just those that were created since the last cycle. Second, there may be a number of Strings with the same initial and terminating node, rather than one String representing the collective transitive dependency of the first node on the last. And, third, each String consists of a number of transaction identifiers, rather than just those of the initial and terminating nodes. The proliferation of deadlock detection messages is partially mitigated by transmitting Strings only in the direction of the dependency, and by transmitting only those Strings in which the identifier of the first transaction is lexically greater than that of the last.

Obermarck's algorithm also makes an explicit attempt to synchronize the deadlock detection cycles in the distributed sites, by defining the periodicity of the cycles and by timestamping intersite messages. During each iteration, only the latest set
of messages from a particular site are used; earlier sets are discarded as being out of date. The effects of this synchronization are completely negated by the effects of integrating the results of previous iterations at remote sites into the local graph of the current iteration, ignoring the fact that processing has continued at the remote sites and that the Strings received from them are increasingly likely to contain a mixture of valid and invalid information. Obermarck might be better off augmenting the current graph until it stabilizes from one iteration to the next, instead of reconstructing the graph at each iteration. To counteract this problem, Obermarck suggests validating a detected cycle by sending a message containing the cycle to each site represented in the cycle, in order (either with or against the direction of the dependencies). Receipt of the verification message at the originating site would constitute verification of the deadlock. This additional step introduces the concept of edge-chasing, which is the basic approach used in the next class of algorithms.

One disadvantage of Obermarck's algorithm is that a particular deadlock cycle can be detected asynchronously by several sites during a single iteration of the detection algorithm. Furthermore, each of the detecting sites might choose a different victim to resolve the deadlock. Iteration 3 of the example given by Obermarck [18, Figure 6, page 202] is duplicated in Figure 3. In this example, the cycle (2, 3, 4, 2) is detected at sites A and C. Site A chooses transaction 3 as a victim, while site C chooses transaction 2. In addition to the needless abort/restart of transactions, this situation points out the duplication of effort by the individual sites. While a certain amount of such duplication is useful to prevent loss of information during adverse
Figure 3: Obermarck
conditions, it also entails the transmission of redundant deadlock detection messages.

**Ho and Ramamoorthy**

The three algorithms published by Ho and Ramamoorthy [10] are primarily concerned with the construction of a global wait-for-graph for a distributed system that is internally consistent despite the communication delays inherent in distributed processing. In each of these algorithms, local status information is collected at each site and periodically transmitted to some control site in response to a broadcast request. The control site constructs a global wait-for-graph and analyzes that graph for deadlock cycles. The method used to select the control site is not discussed. None of these algorithms presupposes two-phase locking.

In the first algorithm, each site maintains a process status table identifying the resources that the process holds and those for which it is waiting. Periodically, the control site broadcasts a message to all sites requesting their status tables. When all of the individual tables have been received, the control site constructs and analyzes the resulting global wait-for-graph. If one or more cycles are found, the control site broadcasts another request for status tables and constructs a second wait-for-graph, this time including only those transactions that are common between the two graphs. If analysis of the second graph shows a cycle, deadlock is declared. While this double detection cycle will catch the inconsistencies created by asynchronous reporting during the first detection cycle, similar conditions can occur during the second cycle, with the same result: false deadlock detection.
The second algorithm employs a one-pass detection scheme, as opposed to the two-pass scheme of the first algorithm. Each site maintains two status tables: a resource table, containing lists of the transactions holding and waiting for each local resource; and, a process table, containing the transactions spawned by each local process. In this algorithm, a transaction is uniquely identified by the pair \((P, t)\), where \(P\) is the process that spawned it and \(t\) is its timestamp. The sequence of events involved in a resource request, grant, and release are shown in Figure 4. The key component of this algorithm is that all of the sites involved in a dependency maintain the complete status of the dependency in their tables.

As in the first algorithm, a control site periodically broadcasts a request for status to all sites in the network and builds a global wait-for-graph based on their responses. In this case, however, the global graph is constructed using only those transactions for which the resource and process table entries from the involved sites agree. The fact that they do agree means that there are no outstanding messages concerning the dependency.

The maintenance of the local process and resource tables adds at most one message per transaction to the normal process load: the message acknowledging a request that cannot yet be granted. All other updates can be accomplished by the normal request, grant, and release messages. The periodic deadlock detection cycle requires \(2n\) messages, where \(n\) is the number of sites: one message from the control site to each remote site requesting the status tables; and one message from each
Event Process Site Resource Site

Request
Add Ptab(P,t,R,w)
Grant
Chg Rtab(P,t,R,a)
Release
Del Ptab(P,t,R,a)

Add Rtab(P,t,R,w)
Acknowledge
Chg Ptab(P,t,R,a)
Delete
Del Rtab(P,t,R,a)

r 2
r 2
t 1
r 2
t 1
Grant
Delete

Figure 4: Ho and Ramamoorthy
remote site to the control site containing the requested tables. The status request messages are quite short; the replies, however, must convey the contents of the status tables and may be quite long.

The third algorithm proposed by Ho and Ramamoorthy uses the same method of maintaining the process and resource tables, but groups the database sites into a hierarchy of clusters for the purpose of deadlock detection. In this algorithm, the central control site periodically broadcasts a status request to its subordinate control sites. Each of these control sites, in turn, broadcasts a status request to its subordinates. As the status information is collected, each control site constructs and analyzes a wait-for-graph using the approach identified in the second algorithm. Only information relevant to inter-cluster transactions is forwarded up to the next level in the hierarchy.

Wuu and Bernstein

The work of Wuu and Bernstein [25] is important because it identifies two methods for avoiding false deadlock detection, which has proven to be one of the major deficiencies in the path-pushing algorithms. Incorporation of either of these methods in a path-pushing algorithm will increase its effectiveness and may make it a viable deadlock detection scheme.

The first method is to use a two-phase locking protocol for the request and release of all resources by a transaction. Wuu and Bernstein submit a proof that the use of the two-phase locking protocol for all transactions will prevent false deadlock
detection. This is a characteristic of the underlying database concurrency control method rather than of the deadlock detection algorithm, however, and will not be discussed further here.

The second method proposed by Wuu and Bernstein employs timestamping of the edges in a wait-for-graph to determine whether the dependencies they represent actually coexisted in the database system. The following scenario defines the timing of significant resource management events. Given that transaction \( T_1 \) requests a resource from resource controller \( R_1 \), and that that resource is currently held by transaction \( T_2 \), define the following sequence of events:

- \( t_r \) \( T_1 \) sends a request message to \( R_1 \).
- \( t_a \) \( R_1 \) receives the request message.
- \( t_s \) \( T_2 \) sends a release message to \( R_1 \).
- \( t_v \) \( R_1 \) receives the release message.
- \( t_n \) \( R_1 \) sends a grant message to \( T_1 \).
- \( t_g \) \( T_1 \) receives the grant message.

Either of events \( t_a \) and \( t_r \) can be used to mark the creation of a dependency, as long as the usage is consistent throughout the implementation. Event \( t_s \) marks the destruction of a dependency. If event \( t_a \) (\( t_r \)) occurs before event \( t_s \), then the dependency exists from time \( t_a \) (\( t_r \)) to time \( t_s \). The existence interval of the related edge, \((T_1, T_2)\), in the wait-for-graph is \([t_a, t_s]\) ([\( t_r, t_s \)]). Two edges coexist in the wait-for-graph if their existence intervals overlap; i.e., when \( \max(t_{a1}, t_{a2}) < \min(t_{s1}, t_{s2}) \) \( \) \( \max(t_{r1}, t_{r2}) < \min(t_{s1}, t_{s2}) \).

This algorithm employs a central deadlock detector, to which messages concerning the creation and destruction of dependencies are sent by the resource controllers
distributed throughout the network. These sites are connected by a reliable network, ensuring that messages are not lost, duplicated, or delivered out of order. Upon receiving a request for an unavailable resource, the resource controller informs the deadlock detector of the creation of an edge at time $t_a$ ($t_r$). When the resource controller receives the release of a resource for which a transaction is waiting, it informs the deadlock detector of the destruction of the relevant edge at time $t_s$. The deadlock detector must know the maximum time, $\Delta$, for message transmission between any two sites in the network, including its own site. If the resource controller sends creation and destruction messages to the deadlock detector immediately upon receipt of the relevant request and grant messages, the deadlock detector will know of the creation or deletion of any edge within time $2\Delta$ of the actual event ($\Delta$ for the request/grant to arrive at the resource controller, and $\Delta$ for the related message to arrive at the deadlock detector).

Therefore, if an edge has existed in the wait-for-graph for longer than $2\Delta$, the related dependency must have existed in the database at time $t_c - 2\Delta$, where $t_c$ is the current time at the deadlock detector. To determine whether a deadlock exists in the database, the deadlock detector must create a wait-for-graph using all edges created before time $t_c - 2\Delta$. These edges coexisted at time $t_c - 2\Delta$. If this wait-for-graph contains a cycle, then a deadlock exists among the related transactions.

The key to this algorithm is the determination of the maximum transmission delay, $\Delta$. This delay must be both calculable and reasonable in terms of the delay.
in detection of a deadlock, because no deadlock will be detected in the system for at least this amount of time. The messages used to collect the deadlock information are neither long nor complex. Each consists of two transaction identifiers, a timestamp, and some indication of whether the message represents the creation or destruction of an edge. Only the resource controllers and the detector are involved in this communication. The fact that one central detector is used will increase the value of $\Delta$ to accommodate any congestion that may be experienced due to the use of a central site.

Elmagarmid, Sheth, and Liu

The algorithms proposed by Elmagarmid, Sheth, and Liu [7] take a quite different approach to deadlock detection than the path-pushing algorithms discussed previously. In the CDDMOR and PDDDA algorithms, the database resources are distributed across a number of sites in the system. Each of these resources can be held exclusively by a single process or shared by a number of processes. The processes themselves can be composed of a number of transactions, each of which can independently request and release resources. In this manner, a single process can have multiple outstanding requests for resources. At some site, which may or may not be coincident with a partition of the database, there is a central controller, which is responsible for the maintenance of the process-resource interaction database (PRIDB) used for both resource allocation and deadlock detection. The PRIDB consists of a resource table and a process table. The resource table contains, for each resource, the mode (shared or exclusive) in which it is currently held and a list of processes by which it is held.
The process table contains, for each process, a list of the resources for which each process is waiting and the modes in which each of those resources is requested.

The first algorithm, the Centralized Deadlock Detection with Multiple Outstanding Requests (CDDMOR) [7], is mentioned only for completeness, since no messages other than simple resource request, grant, and release messages are transmitted across the network. All request and release messages are sent to the central controller, which updates the PRIDB accordingly. The deadlock detection algorithm is initiated under three conditions:

- when a process requests a resource that is unavailable (held in a conflicting mode);
- when a resource is allocated to some subset of the processes waiting for it (because the waiting processes request the resource in conflicting modes); and,
- when a resource is allocated to a process which has one or more requests still outstanding.

In each of these cases, the detection process constructs a wait-for-graph, beginning with the waiting process and incrementally adding the resources and processes on which it (transitively) waits. A cycle in this graph denotes a deadlock condition.

The second algorithm, the Partially Distributed Deadlock Detection Algorithm (PDDDA) [7], attempts to reduce the congestion at the central controller by performing some resource allocations at the sites of the individual resources rather than at the central site. A local controller at the site of each resource maintains the same type of process and resource tables maintained by the central controller. When a local controller detects the possibility of a multi-site deadlock in a subgraph for which it
is responsible, it transmits to the central controller the resource and process table entries for all nodes in that subgraph. While the possibility of a global deadlock exists in the subgraph, all resource allocation and deadlock detection for the involved processes and resources takes place at the central controller. When the possibility of global deadlock disappears, after all of the processes holding the involved resource complete, control of that resource is transferred back to the appropriate local controller.

A subgraph is classified in one of three ways: local, global, or central. The processes and resources involved in a subgraph share its classification. If the processes involved in a subgraph are holding or waiting only for local resources, and all of those resources are held or waited for only by local processes, the subgraph is classified as local. A local subgraph is reclassified as global when one of its processes is granted access to a remote resource; in this class of subgraph all multi-site requests have been satisfied, although some processes might still be waiting for resources local to themselves. The process-resource interactions contained in local and global subgraphs are safe interactions; there is no possibility of a multi-site (global) deadlock produced by such interactions. The process and resource table entries concerning safe interactions are maintained by the local controllers at the sites of the resources involved.

A local or global subgraph is reclassified as a central subgraph due to an unsafe process-resource interaction; one that may create a global deadlock. An unsafe interaction occurs when a process is denied access to a remote resource, or to a lo-
cal resource involved in a global subgraph. The process and resource table entries concerning these interactions are transferred to the central controller at this time. Control of all resources and transactions involved in this subgraph will be retained by the central controller while the possibility of global deadlock exists.

The communications overhead created by the PDDDA consists of two types of messages: the transmission of the process and resource tables between a local controller and the central controller when a subgraph is classified as central or declassified to local; and, the forwarding of resource requests from a local controller to the central controller for any resource involved in a central subgraph. As in Ho and Ramamoorthy’s second algorithm [10], messages of the first type may be quite long.

Elmagarmid, Soundararajan, and Liu

The algorithm published in Elmagarmid, Soundararajan, and Liu [8] carries the above concepts to a fully distributed level. Transactions and resources are distributed across the sites in the system, and each has a corresponding transaction or resource controller at that site. The controllers communicate among themselves to accomplish all tasks involved in resource allocation and deadlock detection. Such communication is synchronized: both the sending and receiving controllers must be ready to communicate with each other for a message to be transmitted; if one is not ready the other must wait.

Interacting transactions and resources form disjoint subgraphs in the system. One transaction controller in each subgraph assumes charge of all the resources and
transactions of which the subgraph is comprised. The remaining transaction and resource controllers forward all communications to this controller. As resources are requested, the subgraph is augmented by the requesting transaction and all of the resources and other transactions with which it is interacting. Control of all such resources and transactions is transferred to the managing transaction controller. This arrangement ensures that all of the information about the state of a particular resource or transaction is kept in one place, avoiding the issue of conflicts due to message delay. As resources are released by one transaction, they are assigned to some subset of the waiting transactions and deadlock detection is performed. If, at this time, the subgraph can be partitioned into disjoint subgraphs, control of the members of each disjoint partition is transferred to one transaction controller in that partition. This ensures the highest possible degree of concurrency in the system.

Each transaction controller maintains a resource table and a transaction table, containing entries for each resource and transaction it is currently managing. Each resource table entry is a quintuple consisting of the resource identifier, the mode in which the resource is currently held, the set of transactions currently holding the resource, the set of transactions waiting for shared access to the resource, and the set of transactions waiting for exclusive access to the resource. Each transaction table entry is a triple consisting of the transaction identifier, the set of resources it is currently holding, and the set of resources for which it is waiting. The resource controller, on the other hand, only has to keep track of the controller currently managing the resource; either itself or one of the transaction controllers.
There are a number of components to the communications overhead involved in this algorithm. Each request for a resource that is not currently idle will have to be forwarded by its resource controller to the transaction controller by which the resource is currently managed. Since each resource controller keeps track of its current manager, the request can be forwarded in one hop. The request itself is a simple quadruple identifying the resource and mode of access being requested, the transaction on behalf of which the request is made, and the originating controller (the manager of that transaction, which actually makes the request).

Upon receipt of such a forwarded request, the managing transaction controller requests the originating controller to transfer its transaction and resource tables to the managing controller. The originating controller must not only transmit these tables to its new manager, but also inform each of the transactions and resources that it was managing of the identity of their new controller. This involves transmission of a short request and a long response between the managing and originating controllers, and a variable number of short messages between the originating controller and the controllers it had been managing.

When a subgraph is partitioned due to the release of a resource, the managing controller must send the resource and transaction table entries corresponding to the members of each partition to its new managing controller. Furthermore, each of the resources and transactions in the related partitions must be informed of its new manager. Again, there are one long and several short messages per partition.
Summary

The path-pushing algorithms as a class have largely been dismissed as both incorrect and inefficient, due to the difficulty of constructing an internally consistent global view of a system from inherently asynchronous sources. This reputation is due mainly to the problems encountered in early contributions to the class, and may well be revised based upon the corrective methods proposed in later algorithms.

Within this class can be found a wide variety of approaches to the collection and transmission of state information. The ten algorithms referenced here are evenly divided between the location of the deadlock detector at distributed, hierarchical, and central sites. They are similarly divided about whether database management and deadlock detection operations are to be collocated: three of the algorithms ([14, Distributed Model], [18], and [8]) involve no separate sites for deadlock detection, while two algorithms ([25] and [7, CDDMOR]) completely separate the deadlock detection and database management functions. Fully half of the algorithms incorporate some combination of local (to the site of the database management functions) and remote deadlock detection. Perhaps the most significant aspect of this distribution is that seven of the ten algorithms require at least some communications links to exist solely for the purpose of deadlock detection.

Only the algorithms proposed by Ho and Ramamoorthy [10] make explicit use of broadcast technology; while Obermarck also attempts to synchronize the deadlock detection cycles at the separate sites, the remaining six algorithms prefer continuous
and asynchronous update of the state information. Most of the messages involved in deadlock detection are short; several algorithms ([7, PDDDA] and [8]) also periodically transmit long messages containing the process and resource tables. The algorithms proposed by Ho and Ramamoorthy [10] transmit such long messages exclusively, but this occurs only in response to the broadcast request for information, not during regular processing. Singularly, the [7, CDDMOR] algorithm involves no deadlock detection messages at all.

3.2.2 Edge-Chasing Algorithms

While the path-pushing algorithms collect state information by transmitting portions of explicit wait-for-graphs from site to site in the system, the edge-chasing algorithms transmit special messages called probes between interacting processes. These probes are transmitted in one direction only, either following a dependency (i.e., if process $t_1$ requests a resource from process $t_2$, the probe would be sent from $t_1$ to $t_2$) or in the opposite direction of the dependency (i.e., from $t_2$ to $t_1$). A process receiving a probe that it initiated has discovered a deadlock.

Knapp [13] references two edge-chasing algorithms in his survey, one by Chandy and Misra [13, Section 4.3], and the other by Mitchell and Merritt [13, Section 4.2]. Both algorithms are considered correct by Knapp and Elmagarmid [6]. The salient characteristics of those algorithms, as presented in Knapp’s article, will be briefly discussed here. A more detailed look is taken at an algorithm proposed by Sinha and Natarajan [22] and modified by Choudhary et al. [4] based upon their actual imple-
mentation of the original algorithm. As in the discussion of path-pushing algorithms, the edge-chasing algorithms are presented here in chronological order.

Chandy and Misra

The edge-chasing algorithm referenced by Knapp [13, Section 4.3] was first proposed by Chandy and Misra in 1982 and later corrected by Chandy, Misra, and Haas in 1983. In this algorithm, there is a transaction controller at each site in the distributed system. Probes are initiated by the controllers on behalf of idle (blocked) transactions and forwarded with the direction of the transaction dependencies. Each probe consists of a triple \((i, j, k)\) representing the transaction \(t_i\) on behalf of which the probe was initiated, and the dependency \((t_j, t_k)\) that is currently being traversed by the probe. A probe \((i, j, k)\) received by the controller of transaction \(t_k\) is forwarded if that transaction is blocked and the dependency of the \(t_i\) on \(t_k\) was not previously known to this controller. This last clause eliminates the duplication of probes, and represents the most complicated part of the algorithm. Receipt of a probe of the form \((i, j, i)\) by the controller of \(t_i\) constitutes detection of a deadlock situation.

Mitchell and Merritt

Knapp also references a 1984 algorithm proposed by Mitchell and Merritt [13, Section 4.2]. In this algorithm, each process maintains two state variables, called its public and private labels. The private label is always unique throughout the distributed system; its public label may or may not be unique. The edge-chasing probes transmit
a single public label in the opposite direction of the inter-process dependencies. To do this, a process must also keep track of the processes which are blocked on it.

Initially, the public and private labels of a process are equal: $u_n = v_n$. When a process $p_1$ becomes blocked on a resource held by another process $p_2$, $p_1$'s public and private labels are changed to a unique value greater than either of the previous public labels of the processes involved: $u_1$ and $v_1$ are set to the same value $z$, where $z > \max(u_1, u_2)$ and $z$ is unique throughout the system. The newly blocked process will initiate a probe containing its public label, and send it to all of the processes which are, in turn, blocked on it. Upon receipt of a probe, a process $p_n$ must compare the probe value to its own public label. If the probe value $p$ is greater than the public label, the public label is reset to the probe value, $u_n = p$, and the probe is forwarded to any processes blocked on $p_n$. The private label of $p_n$ remains unchanged, it will now be unequal to the public label. If the probe value $p$ is equal to both the public and private labels of process $p_n$ ($p = u_n = v_n$), a deadlock has been detected and process $p_n$ can be aborted. If the probe value $p$ is less than the public label, or if it equals the public label but not the private label, the probe is discarded.

Sinha and Natarajan

The algorithm proposed by Sinha and Natarajan [22] was corrected and implemented by Choudhary et al. [4]. The underlying system consists of a database distributed among a number of sites, which are connected by a reliable point-to-point network. The reliability attribute ensures that all messages arrive at their destination in the
order in which they were sent, within a finite amount of time, and without duplication or errors. The transactions in the system obey the two-phase locking protocol, and a transaction can have only one outstanding data request at a time. The transaction identifiers are based upon a combination of site identifier and timestamp, and are used to assign a unique priority to each transaction. Each data item is controlled exclusively by a data manager. Deadlock detection messages are always sent between transactions and data managers, never between two transactions or two data managers.

A deadlock detection probe is initiated by a data manager that detects an antagonistic conflict. An antagonistic conflict is one in which a transaction of higher priority (earlier timestamp) waits for a transaction of lower priority (later timestamp). This may occur in any of the following circumstances: when a data manager receives and cannot grant a resource request, when a data manager can satisfy only some of the outstanding requests for a particular resource, or when a data manager reassigns a resource following the completion or abortion of the transaction by which it was previously held. In the last case, the data manager must send a request to each transaction still waiting on that resource to retransmit all of its probes to the data manager. A probe consists of an ordered pair of transaction identifiers, \((initiator, junior)\), in which the initiator is the identifier of the transaction waiting in antagonistic conflict and the junior is the identifier of the transaction with the least priority among those encountered so far in the deadlock cycle. Obviously, the value of \(initiator\) is set only by the originating data manager and remains unchanged throughout the life of the
probe, while the value of *junior* may be changed by any transaction receiving the probe. The probe is sent by the data manager to the transaction(s) which hold the resource in question.

A transaction receiving a probe will place its own identifier in the *junior* part of the probe if it has a lower priority than the current junior transaction. If the transaction is blocked on an outstanding data request, it will forward the probe to the data manager involved. A data manager receiving a probe will: declare deadlock, if the initiator of the probe is the current holder of the resource; forward it, if the initiator of the probe is in antagonistic conflict with the current holder of the resource; or, discard the probe.

**Summary**

There is much less variation in the approaches taken by algorithms proposed in this category. In each case, the deadlock detection functionality is implemented at each site in the distributed system; there are no separate controllers dedicated to this function, as there were in many of the path-pushing algorithms. Furthermore, the communications involved in the deadlock detection process occur along the same communications links as the routine resource request, grant, and release messages; there are no communications links existing solely for the purpose of deadlock detection. Finally, the deadlock detection probes consist entirely of very short messages (one, two, or three identifiers per message in the algorithms referenced here), distinct from the resource management messages, that are sent continuously during the normal
processing. The major communications-related variation in these algorithms appears to be whether the probes are sent with or against the direction of the transaction dependencies.

3.2.3 Diffusing Computation Algorithms

The diffusing computation algorithms also use special messages to perform their deadlock detection. In this case, however, there are two types of messages: queries, which are sent in the direction of a dependency; and, replies, which are sent in the opposite direction. The computation is initiated by a blocked process and constitutes a depth-first walk of the virtual tree of resource and process dependencies. A key component of the algorithm is that only blocked processes will send queries and replies; an executing process will discard all such messages. When a blocked process receives the first query on behalf of a particular process, called the engaging query, it changes from an inactive to an active state; an active process forwards the query to the nodes on which it is blocked, and immediately replies to any subsequent queries. An active process replies to the original query only when it has received replies to all of the queries it has sent; the process returns to the neutral state at this time. If an initiating process receives a reply to every query that it has sent, a deadlock has been detected.

Only the first of the two algorithms referenced by Knapp [13, Section 4.4] will be discussed here. The other algorithm, by Hermann and Chandy [13, Section 4.5] extends the application of the algorithm to a more complicated processing model in-
volving nested transactions. Both of the algorithms referenced in [13] are considered correct. Several other publications by Misra and Chandy ([16] and [17]) discuss the application of this class of algorithm under specialized circumstances: in communicating sequential processes and for knot detection, respectively.

Chandy, Misra, and Haas

The diffusing computation algorithm referenced by Knapp [13, Section 4.4] was published by Chandy, Misra, and Haas in 1983. The query and reply messages used for deadlock detection are triples of the form \((p_i, p_s, p_r)\), where \(p_i\) is the initiating process, \(p_s\) is the process sending the message, and \(p_r\) is the process to whom the message is being sent. Processes at any two sites in the system must be able to communicate. Only blocked processes take part in deadlock detection. An executing process will discard all queries and replies it receives, even if the replies are in response to queries initiated by that process while in a previous blocked state.

Two local variables are maintained by each process for each engaging query: \(num(i)\) and \(wait(i)\), where \(i\) represents the initiating process, \(p_i\). The variable \(num(i)\) contains the number of outstanding queries (queries for which no reply has yet been received) sent on behalf of \(p_i\); the variable \(wait(i)\) is a boolean value indicating whether this process has been blocked continuously since receipt of the engaging query. In any executing process, the values of \(wait(i)\) will be false for all \(i\).

When a process \(p_i\) becomes blocked on a resource, it initiates a diffusing computation by sending a query \((p_i, p_i, p_r)\) to each process on which it is blocked. At this
point, \(\text{wait}(i)\) becomes true and the total number of queries sent is placed in \(\text{num}(i)\).

A blocked process \(p_k\) receiving a query \((p_i, p_j, p_k)\) first determines whether this is the engaging query for \(p_i\), the first time it has received a query on behalf of that process. If it is an engaging query, process \(p_k\) follows the same procedure used to initiate a computation: it sets \(\text{wait}(i)\) to true, sends a query \((p_i, p_k, p_m)\) to each process on which it is blocked, and sets \(\text{num}(i)\) to the number of such processes. If this query is not an engaging query, process \(p_k\) will send a reply \((p_i, p_k, p_j)\) immediately to process \(p_j\). In this case, the local variables \(\text{wait}(i)\) and \(\text{num}(i)\) are not changed.

A blocked process \(p_j\) receiving a reply \((p_i, p_k, p_j)\) first checks the value of \(\text{wait}(i)\) to determine if it has been blocked continuously since receiving the engaging query for \(p_i\). If it has not been blocked, the reply is discarded. If process \(p_j\) remains blocked, it will decrement the number of outstanding queries contained in \(\text{num}(i)\). If the result is non-zero, indicating that process \(p_j\) is still awaiting replies to some queries, no further action is taken. If there are no outstanding queries, and \(p_j\) is not the initiator of the computation, \(p_j\) will send a reply to the process from which it received the engaging query for \(p_i\). If \(p_j\) is the initiating process for the computation, it will declare deadlock.

**Summary**

As noted in the previous discussion of edge-chasing algorithms, there is relatively little variation among the diffusing computation algorithms in terms of the characteristics of their communications requirements and usage. The deadlock detection function-
ality must be implemented at each site in the distributed database. The communications involved in deadlock detection, while distinct from the resource management (request, release, and grant) messages, use the same inter-site links. In contrast to the unidirectional requirements of the edge-chasing algorithms, the diffusing computations require two-way communications to enable the transmission of both queries and replies. The basic query and reply messages are quite short. Some variation is found in the differentiation of coexisting diffusing computations: the algorithm of Chandy, Misra, and Haas [13, Section 4.4] uses a simple sequence number unique for each initiating process, while the algorithm of Hermann and Chandy [13, Section 4.5] places an explicit path from the initiator to the current node in each message.

3.2.4 Global State Detection Algorithms

The *global state detection* algorithms depend on the ability to capture a global snapshot of the system and then manipulate that snapshot offline to determine whether or not a deadlock exists. A snapshot is taken of the state (outstanding dependencies) of all processes at a particular site, $s_1$. This snapshot is saved or transmitted to some final processing site. At the same time the snapshot is taken, a *marker* is sent from site $s_1$ to some other site, $s_2$. Upon receipt of the marker, site $s_2$ takes a snapshot of its processes, disregarding any messages received from site $s_1$ that were sent from $s_1$ after it propagated the marker. This procedure is repeated until a snapshot has been taken at every site in the system. All of the snapshots are then collected and integrated to produce a consistent global view of the system. The global view can
then be processed offline in any of a number of ways to determine whether a deadlock exists.

The definition of the class of global state detection algorithms in Knapp is based upon the 1985 work of Chandy and Lamport [13, Section 3.4]. No details of their method of implementing this procedure are given, and no algorithms using this procedure are explicitly discussed. Knapp does cite a 1983 algorithm by Bracha and Toueg [13, Section 4.6] that uses the concept of global state detection to collect its state information, and then applies a variation of the diffusing computation techniques to the global view to perform the actual deadlock detection. However, even in this discussion no details of the collection of state information are given.

In the absence of further information, the following attempt is made to identify some of the communications requirements inherent in the implementation of an algorithm of this class. A global state detection algorithm is, by definition, both central and periodic: the individual snapshots are collected at some single site and processed offline to determine whether a deadlock exists. The offline nature of the processing dictates that the detection of any deadlock will be delayed until all local states are collected and the detection algorithm run; this delay may be significant.

During the time that the snapshots are being taken, each site must be able to temporarily suspend communications with selected remote sites in the network. To do so effectively, each site must know the complete membership of each of two disjoint sets: the set of sites that have already submitted their snapshots; and, the
set of sites that have yet to do so. The membership of both sets changes dynamically
during this period as each site takes its snapshot and forwards it to the central site.
Communications with other members of the same set is allowed; communications
between sets prohibited. Each site must know when it is due to take a local snapshot,
and to whom the snapshot is to be sent.

The messages involved in the collection of state information are of two types. Each site
must inform all remote sites that it is taking its snapshot. These messages
are most likely short, containing a site identifier and a timestamp, and must be used by
the receiving sites to terminate or restart communications with the sending site. The
messages used to transmit a site's snapshot to the site at which it will be processed
must contain all the local state information, and is more likely to be long. If the offline
processing occurs at a site collocated with a database partition, no communications
links are required other than those used for routine data management.

3.3 Environmental Alternatives Provided by Network Architecture

Historically, each networking developer defined its own architectural model and proto-
cols. The increasing emphasis on inter-network communications has promoted interest
in the specification of a common model. One of the first applications of this concept
is the TCP/IP protocol family used by the Internet. More recently, the International
Standards Organization has proposed a Reference Model for Open Systems Inter-
connection (OSI), toward which networking technology is expected to migrate. The
correspondence between the TCP/IP protocols and the OSI model is very loose. The
Internet model of network architecture and TCP/IP protocols have been chosen as the framework for the following discussion because of their availability and popularity in the United States.

The TCP/IP model is comprised of five layers: hardware, network interface, internet, transport, and application. The hardware layer implements the physical connection between the machines on the network; its unit of transmission is the bit. The network interface layer is the functional equivalent of the OSI data link layer; it is hardware-specific and may consist of either a device driver or a link level protocol implementation. The unit of transmission between the network interface and hardware layers is the frame. The internet layer supervises all machine-to-machine transmission, including routing, basic flow control, and synchronization. The protocols through which this functionality are implemented are the Internet Protocol (IP), which supervises the transmission of process-to-process datagrams, and the Internet Control Message Protocol (ICMP), which implements the coordination between the instantiations of the IP on each host in the network. The service provided by the IP is usually described as connectionless (each datagram is treated as an independent unit) and unreliable (delivery is not guaranteed). The unit of transmission between the internet and network interface layers is the datagram. These three layers are sometimes collectively referred to as the ‘communications subnet’. The functionality of these layers is dependent in many respects on the topological design of the network, the physical communications media connecting the nodes, and low level access and control mechanisms used for their management. They must be present at all
intermediate nodes of the network, as well as at the source and destination hosts.

The transport layer provides end-to-end (process-to-process) services to applications running on the hosts attached to the network. The transport layer must implement any network services requested or required by an application that are not met by the connectionless, unreliable, datagram service supplied by the internet layer. Such services might include buffered transmission, reliable message delivery, (a)synchronous connections, and simplex, full- or half-duplex transmission. Of these, the most frequently requested service is reliability: the guaranteed, ordered, error-free delivery of messages. Most networks, the Internet included, implement several transport layer protocols providing different levels of service. The unit of transmission between the transport and internet layers is the packet.

The application layer sits above the transport layer; an implementation of a deadlock detection algorithm would reside at this layer. The unit of transmission between the application and transport layers is the message.

3.3.1 Alternatives for Network Class and Topology

There are two basic classes of networks: point-to-point and broadcast. The topological choices available for an installation are determined by the network class; they, in turn, dictate many of the functions that must be provided by the communications subnet.
Point-to-point Networks

Point-to-point networks can be configured as a tree, loop, or mesh, as shown in Figure 5. In a point-to-point network, a pair of nodes is connected by a single dedicated communications channel. Messages passing through the network are routed from node to node until they reach their destination. At each intermediate node, the next leg of the journey is determined and the message is stored until the corresponding outbound channel becomes available. There is no contention for use of a channel because each is dedicated to a specific path; network management takes the form of controlling congestion at the nodes: to reduce the number of queued messages, to evenly distribute them among the queues for outgoing channels, and to avoid the problem of running out of storage space for messages in transit.

In a point-to-point tree network, the nodes are connected to form a hierarchy. The tree need not be either balanced or binary. The hierarchy is generally designed to take advantage of locality of data and function by clustering nodes that frequently exchange messages. Control and supervisory functions can be concentrated at the root node or distributed across the non-leaf nodes in the hierarchy. There is only one path between any pair of nodes in the network: from the source to the lowest common ancestor of the two nodes, and from there to the destination. Consequently, the number of segments traversed by a message depends on the depth of the source and destination nodes in their subtree. The maximum number of concurrent messages in the network approaches the number of distinct internodal segments, although a non-
Figure 5: Point-to-Point Topologies
leaf node with a large number of descendents will not be able to service all incoming lines at once. Congestion at the non-leaf nodes of the network can be a problem, most severe when the tree is unary or binary, and diminishing as the number of descendents per non-leaf node increases.

In a point-to-point loop network, the nodes are connected in a circular fashion. Each node is connected to exactly two others. There is no obvious location for control and supervisory functions. A message travels from one node to another in a variable number of hops, depending on the number of segments between the source and destination nodes. Loops generally provide two routes between each pair of nodes: one in each direction of the loop. One of these routes will generally be shorter than the other. Given \( n \) nodes and \( n - 1 \) segments, the number of messages travelling can equal the number of segments. Congestion management consists of monitoring the length of the queue for a particular segment.

A point-to-point mesh network can best be described as any network that does not qualify as either a tree or a loop. A node can be connected to any or all of the other nodes in the network. A mesh in which each site is connected to all other sites is said to be *fully connected*. There is no obvious location for control and supervisory functions. A message travels from one node to another in a variable number of hops depending on the number of segments between source and destination in the mesh. There can be any number of paths from source to destination, depending on the degree of connectivity. The routing algorithm for mesh networks can be very complex, and
must balance the length of a path from source to destination against the degree of 
congestion at each of the intermediate nodes. A path that may be the shortest might 
also incur the most delay. Given \( n \) nodes and \( n - 1 \) segments, the number of messages 
travelling can equal the number of segments. Congestion control takes the form of 
queue management for a particular segment.

Broadcast Networks

Broadcast networks are usually configured as a bus or ring, as shown in Figure 6. In 
a broadcast network, a single communications channel is shared by all of the nodes 
on the network. The individual nodes compete for control of the channel in order 
to send a message; there are a number of different algorithms for channel allocation, 
both central and distributed, static and dynamic. Once sent, a message is available to 
all of the nodes in the network. There is no routing or congestion control necessary, 
since the message is read and stored only at its destination(s) and since only one 
message can be in transit on the channel at a time.

The class of broadcast networks can be further subdivided into local and wide 
area networks (LANs and WANs), according to the magnitude of the geographical 
distances between the nodes. The technology involved in the physical implementation 
of the two are quite different: LANs generally use cable (e.g., Ethernet or proNET- 
10) for their physical connections, while WANs use telephone lines, radio, or satellite 
for transmission. The access control algorithms can also be quite different, due to 
the delays inherent in long distance transmission. This discussion will concentrate on
Figure 6: Broadcast Topologies
LAN technology.

In a broadcast bus network, all nodes tap into a common linear cable. By definition, each site is directly connected to all other sites on the bus and there is no obvious location for control and supervisory functions. Only one message can be on the bus at a time and each node is capable of receiving every transmission. While there is no routing or congestion control associated with a broadcast network, contention for use of the bus becomes a significant network management function. There are a number of protocols used for channel allocation, the most well known among them is that used in Ethernet networks: carrier sense multiple access with collision detection (CSMA/CD). In this protocol, a node waits until it can sense that the bus is idle before beginning to transmit. The node then listens to its own transmission to determine whether or not there is a collision. Upon detecting a collision, the node stops transmitting and waits for a predetermined interval before attempting retransmission. There is a maximum packet length and a minimum delay between transmissions by a single node to ensure equal access to the channel by all nodes.

A broadcast ring network shares some of the characteristics of a bus: each site is directly connected to all other sites and there is no obvious location for control and supervisory functions. However, the physical configuration of the network is a unidirectional loop rather than a bus, and the channel allocation function is also very different. There are several types of ring networks, differentiated by the channel
allocation protocol used. The most commonly known are the token and slotted rings. In a *token ring* network, a token circulates continually on the ring. A node wishing to transmit must seize the token and remove it from the ring. A node verifies its transmission by receiving the transmitted message at the completion of its circuit of the ring. When transmission is complete, the token is placed back into circulation. As is true with a broadcast bus, only one message can be in transit on a token ring at a time, and the packet length is limited to ensure equal access to the channel. The token passing mechanism takes the place of the inter-transmission delay. A *slotted ring* network, on the other hand, is divided into a number of fixed-size slots. A node wishing to transmit waits until it sees an empty slot, and inserts its message in that slot. In this configuration, many packets can be in transit at a time (as many as there are slots) but each packet must fit in a single slot.

### 3.3.2 Alternatives for Network Services

The Internet recommends implementation of two transport layer protocols: the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). These represent the minimum and maximum extremes, respectively, in the degree of service available over the Internet. A number of other transport protocols, mostly elective or experimental, are available to provide special purpose services. There is growing recognition of the need for a third type of general transport protocol to provide the intermediate level of service appropriate for many transaction processing applications. Two candidates for this level of service are the Internet Reliable Transaction Proto-
col (IRTP) and the Versatile Message Transaction Protocol (VMTP). The services provided by the VMTP will be discussed here.

**Transmission Control Protocol (TCP)**

The Transmission Control Protocol (TCP) defines a reliable end-to-end full-duplex virtual circuit delivery vehicle that makes virtually no assumptions about the level or type of service provided by the layer below it. Comer [5, Section 12.3] identifies the following characteristics of the service provided by the TCP. The TCP provides an *unstructured stream* oriented service; meaning that the data is transferred as a stream of octets, that the octets are delivered in the same order as they were sent, and that the format and meaning of the sequence of octets in the stream is known only to the applications on either end. The TCP provides a *virtual circuit* service; meaning that an end-to-end connection is established between the source and destination hosts before data transfer can begin, and that this connection must be explicitly terminated when transfer is complete. The TCP provides a *buffered transfer* service; meaning that the message lengths used by an application are completely independent (smaller or larger) of the packet sizes used by the TCP. Finally, the TCP provides a *full duplex* connection between two hosts; transfer of data can take place simultaneously in both directions on the virtual circuit.

One of the consequences of providing a reliable service is the additional message overhead associated with the TCP. The TCP provides an end-to-end virtual circuit service; therefore it exchanges packets to open and close a connection as well as
to transfer data. Establishing a connection requires the exchange of three packets (SYN, SYN-ACK, and ACK). Closing a connection requires a separate close of each channel of the full-duplex connection; closing each side involves the transmission of two packets (FIN and FIN-ACK). Every data packet sent by the TCP must be acknowledged, and those acknowledgements that cannot be piggybacked onto other packets must be transmitted separately. All of this is in addition to the overhead involved in the retransmission of packets that have been lost or damaged in transit.

The TCP uses a 20-octet header for each packet, except when options are present. When combined with the 20-octet IP datagram header, this results in at least 40 octets of overhead transmitted for each packet of data. The packet sizes are negotiable between the TCP implementations on the source and destination hosts. Most implementations try to negotiate a packet size that is a small multiple of the frame size for the data link layer interface involved. A small packet size results in an inefficient overhead-to-data ratio because of the 40-byte header size. On the other hand, a large packet size will cause fragmentation by the IP layer because the packet will not fit into a single datagram. The problem with fragmentation is that retransmission takes place by packet: if any fragment fails to arrive or contains errors, all fragments in the packet must be retransmitted.

**User Datagram Protocol (UDP)**

At the other end of the spectrum is the User Datagram Protocol (UDP). The UDP provides a minimal level of transport service to an application; using it is virtually
the same as transmitting directly through the Internet Protocol (IP), which provides a connectionless packet-switching datagram service. There is no guarantee that the datagram will not be duplicated in transit, or that it will even be delivered. The datagram does include a checksum, however, so that the packet can be verified to be error-free if it does arrive at its destination. However, it is up to the application to take action if the transmission was not correct.

The major service provided by the UDP is the ability to distinguish between different protocol ports on the source and destination hosts. A protocol port is an abstract representation of an application process; instead of using process numbers, which might change over time and will certainly be different for instantiations on individual machines, a particular application will register for a protocol port, which will be the same on any network host at all times.

The UDP is a transaction oriented protocol, meaning that each packet is assumed to be complete and self-contained. As its name implies, a UDP packet is the same as a datagram. The length of a UDP packet can be any multiple of two octets, with a minimum length of 8 — the size of a UDP header. The size of the maximum UDP packet effectively depends upon the MTU of the underlying network: 1500 for Ethernet, 2044 for proNET-10, negotiable for point-to-point networks. When combined with the 20-octet header added to the datagram by the IP layer, the total amount of overhead in a UDP packet is 28 octets.
Versatile Message Transaction Protocol (VMTP)

The Versatile Message Transaction Protocol (VMTP) proposes a third, intermediate level of transport service, based on the transmission requirements of transaction processing applications. These applications generally employ a client-server model (one side requests, the other responds), involving a brief (one request, one reply) simplex (one direction at a time) exchange of short (one or a very few) packets per independent transaction [1, Section 2].

The VMTP provides a reliable (ordered, error and duplicate free, guaranteed delivery of messages) service that is not encumbered by some of the overhead required for the TCP. The transaction-oriented nature of the VMTP eliminates the need for much of this overhead: there are no connection establishment and connection closing procedures, packet acknowledgement is implicit in the receipt of a response to a request, and there is no need to keep state information unless a transaction is actively taking place. The protocol includes some additional features that are very useful for some deadlock detection algorithms [3]: The VMTP supports multicasting with multiple responses, meaning that a client can send (broadcast) a single request to a number of servers, each of which will respond exactly once if at all (Section 2.7); a request can be forwarded up to 15 times, with the last server responding directly to the client (Section 2.9); and, each request or response can consist of multiple packets, insulating the application from having to know or function within the constraints of the underlying datagram size (Section 2.14).
The basic unit of user data is a 512-octet segment data block. Each segment is prefaced in a VMTP packet by a header (24 octets) and a message control block (40 octets), and followed by a checksum (4 octets). This results in a packet size of 580 octets, of which 68 are overhead. The underlying IP layer will add another 20 octets of header to each packet, resulting in a total packet length of 600 octets, 88 octets of which are overhead. The VMTP provides a facility for transmitting larger amounts of data using packet groups (up to 32 packets per group, for a maximum of 16K octets of user data), which can themselves be grouped to form a packet run (up to 256 groups per run, for a maximum of 4096K octets). In a less portable variation of the protocol, an implementation with implicit knowledge of the underlying frame size can place more than one segment in a packet. For example, the VMTP running over an Ethernet can transmit two segment blocks per packet and still meet the limit of 1500 octets of user data per frame.

3.3.3 Influence of Network Alternatives on Distributed Applications

Both the network topology and services have a visible effect on the performance of a distributed application. The influence of the former is more critical for a number of reasons. First, the cost of establishing physical connections is often a significant part of the installation of a network. Once installed, the network is a lot less likely to change than the software. Second, as is true with the TCP/IP protocols, a network package usually provides a number of transport level services. Third, the transport protocols are implemented in software, and those not provided by the network package
can be implemented in the application. While this is not necessarily a trivial task, it is feasible.

Several aspects of the difference between point-to-point and broadcast class networks can have a significant impact on the performance of a deadlock detection algorithm. The first concerns the implementation of a multicast capability. Multicasting is a technique that allows a single message to be received by multiple destinations [5, page 344]. Broadcast networks such as the Ethernet provide hardware support for multicasting. Point-to-point networks may provide software support for multicasting in the network layer, although this service is inherently more complicated and less efficient. Network service protocols that provide virtual circuit connections preclude the use of this technique. In algorithms such as Ho and Ramamoorthy’s [10], where a central deadlock detector periodically requests state information from all of the other nodes on the network, the difference in number of messages sent and elapsed time for receipt of the requests at all nodes can be significant.

A second area of importance is the propagation time of a single message between two hosts. In both point-to-point and broadcast networks there is a propagation delay associated with the distance between hosts. This delay varies with the communications media used for the physical connection. However, in a point-to-point network an additional delay is introduced for any message travelling across more than one segment: at each intermediate node the message must be read and stored until the next channel on its path is available. A message sent across several segments of a
network will experience a noticeable delay.

The segmented nature of point-to-point networks results in another area of significant difference between the two classes of networks. In a point-to-point network, a separate transmission can take place on each segment at any point in time. This means that in a network of $n$ nodes, $n - 1$ messages can be in transit at once. In contrast, a broadcast network usually consists of a single segment, and only one message can be in transit on that segment. The positive effects of simultaneous transmission at least partially counteract the negative effects of the store-and-forward delay in point-to-point networks.

One of the major effects of choosing the class of service for the network is the definition of frame size. A frame is the unit of transmission between the data link layers of the network. In the Internet community, the maximum size of a frame is referred to as the maximum transfer unit (MTU). The MTU can vary widely from network to network, and may be either fixed or negotiable. In general, the MTU for local area networks is a fixed size determined by the network standard, while the MTU for a point-to-point network is negotiated by the networking software [5, page 344].

For example, the Internet requires that all connected hosts and gateways be prepared to accept packets of up to 576 octets [5, page 69]. This frame size includes 20 octets of header data from the Internet Protocol (IP) layer of the network, as well as any framing overhead associated with the communications medium.
Many point-to-point connections over short distances are made using dedicated serial lines. The protocol for transmitting TCP/IP packets over such lines is called the Serial Line IP, or SLIP. Romkey [21] has proposed a standard for the data link layer that is very simple. There is no maximum frame size defined, although Romkey recommends adherence to a user data maximum of 1006 octets, which is what is used for the standard Berkeley UNIX distributions of the SLIP software. Added to the user data are one or two bytes of framing, marking the start (optional) and end of text.

The CCITT Recommendation X.25 [2] defines separate maximum frame sizes for its datagram and virtual circuit services. Each X.25 frame contains 7 octets of overhead. The datagram service (Section 5.1.2) provides a variable length user data field with a fixed upper limit of 128 octets. The actual data length may or may not be an integral number of octets, depending on the requirements of the underlying network service. The virtual circuit service (Section 4.3.2) defines as standard a maximum user data length of 128 octets. Again, the length may or may not be an integral number of octets, according to the support of the underlying service. Other user data sizes (of 16, 32, 64, 256, 512, 1024, 2048, and 4096 octets) can be negotiated for each connection or selected for a period of time as the default for all transmissions across an interface.

The Ethernet supports a variable length frame up to a maximum of 1526 octets. Each frame must completely identify both its source and destination as part of the
26 octets of frame packaging overhead. This leaves up to 1500 octets of actual user data (including the headers from higher protocol layers).

The proNET-10 also supports a variable length frame, in this case with a maximum of approximately 2052 octets. The data portion of the frame contains from 0 to 2044 octets. The overhead portion of the frame is 61 bits, between 7 and 8 octets. The frame overhead length reflects the distinction between the units recognized by the physical communications device and the host: the physical transmission takes place in bits, so any number of bits is valid; the data portion of the frame must be transferred to and from a host computer, which uses the octet as its unit.

3.4 Impact of the Network Environment on the Selection of a Deadlock Detection Algorithm

The purpose of this discussion is to assess the relative merits of the deadlock detection algorithms discussed earlier in the context of a particular network topology and transport service. Six of the fifteen deadlock detection algorithms discussed earlier, representing the full range of approaches and attributes, have been selected for this exercise.

3.4.1 Environmental Preferences of Individual Deadlock Detection Algorithms

Each algorithm is analyzed individually to identify the attributes that make it suited or unsuited to implementation using a particular transport service protocol or network topology. The results of this analysis are displayed in two tables, Network Topology...
Obermarck

Obermarck [18] proposed a fully distributed path-pushing algorithm in which the deadlock detection cycles are periodic and synchronized among the sites in the network. At the end of each cycle, each node transmits Strings, representing the transitive dependencies in the local composite wait-for-graph that include an external node, to the remote sites of the transactions on which the Strings are dependent. In each new cycle, the local wait-for-graph is augmented by the nodes and dependencies contained in the latest set of Strings received from remote sites.

The attributes of the algorithm that are key to the determination of network topology include: the periodicity and synchronization of the transmissions; and, the full distribution of both the data collection and deadlock detection functions. Because the deadlock detection cycles are synchronized, all sites will attempt to transmit at once. The simultaneous contention for a single resource by all sites will greatly degrade the performance of both the bus and ring topologies of broadcast networks. The fully distributed nature of the algorithm requires that each site be able to communicate with any other site in the network, as both a sender and a receiver. The degree of concurrency provided by the point-to-point topology is significant for precisely these reasons: at a single point in time, it is theoretically possible that every site will be trying to send to every remote site. The periodicity of the cycle becomes very signif-
Table 1: Network Topology Preferred by Deadlock Detection Algorithm

<table>
<thead>
<tr>
<th></th>
<th>Point-to-point</th>
<th>Broadcast</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mesh</td>
<td>Tree</td>
</tr>
<tr>
<td>Obermarck</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Ho and Ramamoorthy</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Wuu and Bernstein</td>
<td>—</td>
<td>1</td>
</tr>
<tr>
<td>Elmagarmid, Soundararajan, and Liu</td>
<td>3</td>
<td>—</td>
</tr>
<tr>
<td>Sinha and Natarajan</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Chandy, Misra, and Haas</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2: Network Protocol Preferred by Deadlock Detection Algorithm

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>VMTP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obermarck</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Ho and Ramamoorthy</td>
<td>2</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>Wuu and Bernstein</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Elmagarmid, Soundararajan, and Liu</td>
<td>2</td>
<td>1</td>
<td>—</td>
</tr>
<tr>
<td>Sinha and Natarajan</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Chandy, Misra, and Haas</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
icant in this respect. It is not necessary for the Strings to arrive at their destination in a minimal amount of time, just that they arrive before the next deadlock detection cycle begins. The combination of these factors provides some flexibility in the requirements for the degree of connectivity in the network, and all three point-to-point topologies are probably acceptable.

There are several aspects of the algorithm that recommend its implementation using the TCP transport service protocol. Because transmissions take place during a specific period of time, and a site is likely to receive as well as to send messages in this period, the algorithm can take advantage of a full duplex connection. Since each message is directed to a specific destination, the algorithm can make use of virtual circuits between pairs of machines. Because all transmission takes place during a well-defined period, the overhead of setting up and tearing down a connection becomes less significant. The algorithm also assumes the reliability of the connection.

The VMTP and UDP are also acceptable candidates for the implementation of this algorithm. While none of the features of the VMTP are likely to be taken advantage of by the algorithm, it does provide a reliable connection between two sites with minimal overhead. Even for the datagram service provided by the UDP, the length of the Strings is not likely to be a factor. If identifiers are assumed to be four octets in length (for example, the Internet site address), each transaction identifier, composed of a (site, process) pair, would be eight octets in length. A String contains a minimum of two transaction identifiers (external node, transaction), resulting in
a minimum String length of 16 octets. Even the minimal frame size (128 octets of user data) of the X.25 protocols will hold a String of 12 transaction identifiers and a UDP/IP header.

The assumption of reliability imposes some extra work on an implementation over the UDP, but using this service is not out of the question. Given that a String will fit completely into a frame, and that each frame contains an integral number of Strings, the messages are independent and unordered within a single deadlock detection cycle. The algorithm already contains an explicit provision for using only the latest set of Strings from each remote site, this functionality can also be used to eliminate duplicate Strings. order. The remaining two attributes of reliability (guaranteed and error-free delivery) can easily be implemented by an application using acknowledgements, timeouts, and the checksum already provided by the UDP.

In summary, then, Obermarck’s algorithm is a likely candidate for implementation over networks of the mesh, loop, and tree topologies using the TCP, VMTP, and UDP protocols. Both are listed in descending order.

Ho and Ramamoorthy, Hierarchical Model

The path-pushing hierarchical algorithm of Ho and Ramamoorthy [10] also employs a cyclical, synchronized deadlock detection cycle. In this algorithm, there is a hierarchy of deadlock detection controllers. A detection cycle is begun by the root controller broadcasting a request for state information to its descendants. This request is forwarded at each intermediate level of the hierarchy until it reaches the leaf nodes.
These, in turn, respond by transmitting their process and resource state tables back up the chain.

The hierarchical model of Ho and Ramamoorthy [10] is ideally suited for implementation using a point-to-point tree topology. The interior nodes of the tree provide an obvious location for the deadlock detection controllers. The upper levels of the tree do not form a potential bottleneck, as they do in Obermark's algorithm, because all communication is directed from the root to the leaves and vice versa.

The fact that the algorithm uses multicasting to request transmission of the deadlock state information points toward the broadcast networks, which provide superior support of this facility than do the point-to-point networks. However, this algorithm shares the attributes of Obermark's algorithm that led to the disqualification of the broadcast networks: the deadlock detection cycle is both periodic and synchronized, and the simultaneous attempt to transmit the lengthy responses back up the tree causes heavy contention for the use of the single channel.

Given a network level software implementation of multicasting, both the mesh and the loop point-to-point networks are still available. This is particularly true since the structure of the hierarchy can be developed to make the best possible use of the network connectivity. The major problem is to prevent a node from receiving and/or responding to more than one request per cycle.

There are several aspects of the VMTP protocol that make it an ideal candidate for this algorithm. The deadlock detection cycle is a perfect example of both the
request/response and client/server models of communication. Communications are always directed, outbound in one phase and inbound in the other, requiring a half duplex or simplex channel rather than full duplex. The VMTP also implements a multicasting service in which a request can be sent to multiple destinations, each of which can return at most one reply. Furthermore, the algorithm corresponds perfectly to the file server model, which the VMTP was designed to support. Finally, the transport service provided by the VMTP is reliable, which is particularly critical for the transmission of large amounts of data.

This last aspect of the algorithm eliminates the UDP service as a possibility. The length of the response messages, which contain the process and resource tables from the leaf nodes, is likely to greatly exceed the size of a datagram on any type of network. The amount of work and overhead of ensuring the guaranteed, ordered, unduplicated, error-free delivery of the individual datagrams is not to be underestimated or undertaken lightly.

The TCP service can be effectively used for this algorithm, though. Its service is reliable and, because the transmission is both periodic and synchronized, the overhead involved in opening and closing connections is not as much of a burden as it might be for a continuous model. Although the TCP does not support multicasting, the algorithm can effectively employ virtual circuits for its transmissions because of the periodicity of the transmissions.
Wuu and Bernstein, Timestamping Model

The timestamping algorithm of Wuu and Bernstein [25] employs a single central controller for its path-pushing deadlock detection. The resource managers notify this controller whenever a resource dependency is created or destroyed. The central controller uses an estimated maximum time delay to determine the most recent point at which the dependency can be guaranteed to have existed to eliminate conflicts due to transmission delay.

The basic topological requirement of this algorithm is that the transmission delay between the resource controllers and the central controller have very little variance. If the algorithm is extended to maintain a separate $\Delta$ value for each resource controller, the variance can be minimized per path, rather than for all paths. The only topology that is not suited for this algorithm is the mesh point-to-point topology, because there are a number of routes (of potentially different lengths) between any two pairs of nodes. The actual degree of the delay is not particularly important, because the deadlock detection function is run offline, and does not impinge upon the normal activity of the resource managers. Because the resource controllers update the central controller continuously, there is no inherent contention for use of the channel.

Both the TCP and VMTP transport services are capable of providing a reliable, connection between the resource and central controllers. Using either of these protocols for this algorithm entails a large amount of overhead for what is really a minimal set of requirements. The communications involved in this algorithm are
in one direction only, from the resource controllers to the central controller. Each message is self-contained and short enough to easily fit into virtually any frame size. The level of service required is more compatible with that provided by the UDP. The only drawback of using the UDP is that it does not provide a reliable connection.

Two aspects of reliability (ensured and error-free delivery) can be easily implemented at the application level. The timestamp included in every message can be used to discard duplicate transmissions and correctly order the packets. The ordered delivery of datagrams is also promoted by the fact that, outside of a mesh topology, there is only one route between any two nodes.

Elmagarmid, Soundararajan, and Liu

The path-pushing algorithm of Elmagarmid, Soundararajan, and Liu [8] proposes a fully distributed deadlock detection scheme in which a distributed system is dynamically partitioned into disjoint groups of interacting processes and resources. One transaction controller in each partition assumes control of all interacting members of the partition, performing resource management functions as well as the maintenance of the state information used for deadlock detection. The control functionality and state information are transferred between sites as the system is repartitioned. There are two particularly significant aspects to this arrangement. First, since all of the state information regarding interacting processes and resources resides at one site, there is no inherent inconsistency due to message loss or delay. Second, resource request and release messages must be forwarded from the resource controller to the managing
controller, with the result that a communications delay or failure will impair routine processing and resource management as well as deadlock detection.

This configuration makes it imperative that communications be completed as quickly as possible, to avoid loss of operational concurrency as well as the delay in deadlock detection. The requirement for timely communication exists between every pair of sites in the network, because the deadlock detection functionality is fully distributed. The optimal network configuration for this algorithm is a fully connected mesh, since each pair of nodes can communicate directly without contention for channels or congestion at intermediate nodes. However, since the costs associated with a high degree of connectivity reduce the likelihood of a network being well connected, and the performance of this algorithm degrades both quickly and severely as the degree of connectivity decreases, the choice of a mesh topology is, in general, less than advantageous. This algorithm is not at all suitable for implementation over a minimally connected mesh such as the point-to-point loop. The tree topology can also be dismissed from consideration, due to both the relatively low degree of connectivity and the likelihood of congestion at the non-leaf nodes. The bus and ring broadcast topologies provide more suitable environments, since each pair of nodes can communicate directly and there are no store-and-forward delays or failures due to lack of internal storage at intermediate nodes.

There are several aspects of the algorithm that make it ideally suited for implementation over the VMTP. A resource request is forwarded by the resource controller
to the transaction controller by which the resource is currently being managed; this transaction controller eventually responds to the originating controller with a grant or refusal. In the meantime, the transaction controller sends a request for state information to the originating controller; which responds immediately with the desired information. Both of these exchanges follow the request/response and client/server models for which the VMTP was designed. The former exchange involves the forwarding of a request from server to server, while the latter models the operation of a file server; both types of communication are specialties of the VMTP. There are a number of additional short messages used to notify a resource or transaction manager of its new controller. All communications require a reliable connection.

The necessity of transmitting large amounts of data between sites effectively eliminates the use of the UDP for this algorithm, as it did for the algorithm of Ho and Ramamoorthy. The TCP service remains an effective option, the connection overhead being outweighed by the requirements of bulk data transmission.

Sinha and Natarajan

The edge-chasing algorithm proposed by Sinha and Natarajan [22] explicitly assumes the existence of a reliable point-to-point network. When a process becomes blocked on a resource, the manager of that resource initiates a probe and sends it to the transaction(s) holding the resource. They, in turn, forward the probe to the managers of resources for which they hold outstanding requests. Messages are sent only in response to an antagonistic conflict, defined as a dependency of a transaction on
one with a lesser identifier. The identifiers are composed of a \((site-id, timestamp)\) pair, with the consequence that transactions are prioritized by site. The probes themselves consists of two identifiers, that of the initiator and that of the lowest priority transaction in the cycle.

Despite its assumptions, the algorithm can function equally well on a broadcast as well as a point-to-point network, and can be as effective using the unreliable UDP service as the reliable TCP and VMTP services. This algorithm can be implemented on any of the five network topologies; there is no synchronization of communications to eliminate broadcast networks, and no requirement for minimal or consistent transmission time to rule out some of the point-to-point topologies. The probes are transmitted asynchronously, are very short, and can be somewhat delayed without overly degrading the performance of the algorithm.

The use of site order for determining the existence of an antagonistic conflict provides some incentive to implement the algorithm over a topology such as a broadcast ring. Probes are always sent in the direction of the antagonistic conflict, so implementation over the unidirectional ring is ideal. The point-to-point loop can also be organized to take advantage of this characteristic. The careful assignment of site identifiers can help minimize both contention and congestion in these networks.

If a resource can be held in shared as well as exclusive mode, or if a transaction can be simultaneously blocked on more than one resource, the availability of a multicasting capability will improve performance. A hardware or data link level im-
plementation of multicasting (broadcast networks) is more effective than a network level implementation (point-to-point networks).

With respect to the network protocol, the unreliable UDP service actually appears superior for this application than either the VMTP or the TCP. The probes are short enough to easily fit into a datagram, each probe is an unordered complete entity, and the algorithm incorporates elimination of duplicate probes into its normal processing. The only two remaining requirements for reliability (guaranteed and error-free transmission) are easily incorporated into an application.

Chandy, Misra, and Haas

In the diffusing computation algorithm of Chandy, Misra, and Haas [13, Section 4.4], a blocked process sends out a query to each process holding a resource on which it is blocked. Upon receiving the first such query from a particular process, called the engaging query, a blocked process will forward it to all of the processes upon which it is itself blocked. Subsequent queries from the initiating process will receive an immediate reply. Upon receiving a reply to all outstanding queries, a blocked process will send a reply to its engaging query.

This algorithm has many similarities to the edge-chasing algorithm of Sinha and Natarajan: there is no synchronization of communications to eliminate broadcast networks, and no requirement for minimal or consistent transmission time to rule out some of the point-to-point topologies. The queries and replies are transmitted asynchronously to other processes, are very short, and can be somewhat delayed
without overly degrading the performance of the algorithm. Like the algorithm of Sinha and Natarajan, it can be implemented on any of the five network topologies; although this algorithm does not use site ordering and can not derive an advantage from implementation over the ring or loop topologies. This algorithm makes explicit use of multicasting capabilities, and will definitely benefit from the superior efficiency of the broadcast network implementation of this function.

This class of algorithm is best suited for implementation using the VMTP service, since it adheres to the query/response model for which the VMTP was designed, and can use the multicasting functions provided by the VMTP. Of particular interest is the VMTP support of a multicast query, in which at most one response is expected from each destination.

The reliability of the network service is more of an issue in this algorithm than it was for the edge-chasing algorithms. In particular, the duplication of queries can cause the detection of a false deadlock, since the process receiving the duplicate will reply immediately. Message loss is not as critical, although it will at least delay the detection of a deadlock, and may prevent it altogether. On the other hand, both the queries and replies are short and will fit into any size datagram, error-free transmission can be easily implemented by the application software, and the order in which the queries/replies are received is immaterial. Since the UDP itself will not retransmit, the application can insert a sequence number in the packet and use that to eliminate duplicate queries. Combining this with a simple time-out/retransmission function
will effectively eliminate the problem and facilitate the use of the UDP service.

In terms of network services, those provided by the TCP more than adequately support the diffusing computation algorithms. The virtual circuit service provided by the TCP, however, does not support the multicasting functionality and thereby makes the implementation over that service more complex than in the other two cases.

3.4.2 Selection of a Deadlock Detection Algorithm Given a Network Environment

As a final exercise, the merits of the algorithms judged in the last section to be candidates for implementation over a particular network topology and service are examined to determine which are the three most appropriate algorithms for each environment. Within each category, the algorithms are first ordered by the sum of their ratings for the individual topology and service. This composite ranking is then modified by an analysis of the attributes of the individual algorithms to identify the final first, second, and third choices. The results of the exercise are summarized in Table 3.

Mesh Topology

The composite ranking of the candidates for implementation using the TCP service is: Obermarck (2), Ho and Ramamoorthy (4), Elmagarmid, Soundararajan, and Liu (5), Chandy, Misra, and Haas (6), and Sinha and Natarajan (7). While this combination of topology and service is probably the best for Obermarck's algorithm, the correctness of that algorithm is recognized to depend on an inherently invalid assump-
Table 3: Deadlock Detection Algorithms by Network Protocol and Topology

<table>
<thead>
<tr>
<th></th>
<th>TCP</th>
<th>VMTP</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh</td>
<td>Ho and Ramamoorthy</td>
<td>Ho and Ramamoorthy</td>
<td>Sinha and Natarajan</td>
</tr>
<tr>
<td></td>
<td>Sinha and Natarajan</td>
<td>Chandy, Misra, and Haas</td>
<td>Chandy, Misra, and Haas</td>
</tr>
<tr>
<td></td>
<td>Chandy, Misra, and Haas</td>
<td>Sinha and Natarajan</td>
<td>Obermarck</td>
</tr>
<tr>
<td>Tree</td>
<td>Wuu and Bernstein</td>
<td>Ho and Ramamoorthy</td>
<td>Wuu and Bernstein</td>
</tr>
<tr>
<td></td>
<td>Ho and Ramamoorthy</td>
<td>Wuu and Bernstein</td>
<td>Sinha and Natarajan</td>
</tr>
<tr>
<td></td>
<td>Sinha and Natarajan</td>
<td>Chandy, Misra, and Haas</td>
<td>Chandy, Misra, and Haas</td>
</tr>
<tr>
<td>Loop</td>
<td>Wuu and Bernstein</td>
<td>Wuu and Bernstein</td>
<td>Wuu and Bernstein</td>
</tr>
<tr>
<td></td>
<td>Sinha and Natarajan</td>
<td>Sinha and Natarajan</td>
<td>Sinha and Natarajan</td>
</tr>
<tr>
<td></td>
<td>Chandy, Misra, and Haas</td>
<td>Chandy, Misra, and Haas</td>
<td>Chandy, Misra, and Haas</td>
</tr>
<tr>
<td>Bus</td>
<td>Sinha and Natarajan</td>
<td>Chandy, Misra, and Haas</td>
<td>Wuu and Bernstein</td>
</tr>
<tr>
<td></td>
<td>Wuu and Bernstein</td>
<td>Wuu and Bernstein</td>
<td>Sinha and Natarajan</td>
</tr>
<tr>
<td></td>
<td>Chandy, Misra, and Haas</td>
<td>Sinha and Natarajan</td>
<td>Chandy, Misra, and Haas</td>
</tr>
<tr>
<td>Ring</td>
<td>Wuu and Bernstein</td>
<td>Wuu and Bernstein</td>
<td>Wuu and Bernstein</td>
</tr>
<tr>
<td></td>
<td>Sinha and Natarajan</td>
<td>Chandy, Misra, and Haas</td>
<td>Sinha and Natarajan</td>
</tr>
<tr>
<td></td>
<td>Chandy, Misra, and Haas</td>
<td>Sinha and Natarajan</td>
<td>Chandy, Misra, and Haas</td>
</tr>
</tbody>
</table>
tion. Furthermore, Obermarck’s algorithm can result in the detection of a deadlock at multiple sites, at each of which a different victim process may be selected. The combination of these two factors effectively eliminates this algorithm from contention for the top three spots.

The algorithm proposed by Elmagarmid, Soundararajan, and Liu is complex and involves the transfer of resource management as well as deadlock detection functions. The potential effect of communication delays and failures on the concurrency and overall performance of the complete system is significantly more severe than in the other algorithms available. This algorithm, too, is judged to be less effective than the remaining three.

The TCP’s lack of multicast support fuels a certain amount of debate about the final ranking of the remaining three algorithms for this category. Sinha and Natarajan’s algorithm is the only one of the three that does not employ this facility, and might therefore be moved above the other two. However, the periodicity of the Ho and Ramamoorthy’s algorithm substantially mitigates the effect of having to establish multiple virtual circuits. As a result, the final ranking is: (1) Ho and Ramamoorthy, (2) Sinha and Natarajan, (3) Chandy, Misra, and Haas.

The composite ranking of the candidates for implementation using the VMTP service is: Obermarck (3), Ho and Ramamoorthy (3), Elmagarmid, Soundararajan, and Liu (4), Chandy, Misra, and Haas (4), and Sinha and Natarajan (6). The algorithms of Obermarck and Elmagarmid, Soundararajan, and Liu can be eliminated.
from contention for the reasons given in the discussion of the TCP service. There is no compelling reason to adjust the relative ranks of the three remaining algorithms: (1) Ho and Ramamoorthy, (2) Chandy, Misra, and Haas, (3) Sinha and Natarajan.

The composite ranking of the candidates for implementation using the UDP service is: Obermarck (4), Sinha and Natarajan (5), and Chandy, Misra, and Haas (5). For reasons discussed earlier, Obermarck’s algorithm is the least favorable choice of these three algorithms. Although their composite scores are the same, the algorithm proposed by Sinha and Natarajan has some advantages over the one proposed by Chandy, Misra, and Haas. Conceptually, the edge-chasing algorithm employs only half the messages of a diffusing computation algorithm, with the result that the unreliable service has only half the chance to fail. Furthermore, the treatment of problems caused by unreliability is integrated more completely into Sinha and Natarajan’s algorithm than that of Chandy, Misra, and Haas. Consequently, the final ranking of the three algorithms is: (1) Sinha and Natarajan, (2) Chandy, Misra, and Haas, (3) Obermarck.

Tree Topology

The composite ranking of the candidates for implementation using the TCP service is: Ho and Ramamoorthy (3), Obermarck (4), Wuu and Bernstein (4), Sinha and Natarajan (8), and Chandy, Misra, and Haas (8). As before, Obermarck’s algorithm is dismissed as an unlikely candidate. The algorithms of both Wuu and Bernstein and Ho and Ramamoorthy have a natural affinity for this topology; the former ranks
slightly higher, by virtue of its simplicity and because it does not employ multicasting. Sinha and Natarajan's algorithm is somewhat better suited to the tree topology than that of Chandy, Misra, and Haas for two reasons: it does not employ multicasting; and, it (conceptually) requires fewer messages, so there is less likelihood of a bottleneck. As a result, the final ranking is: (1) Wuu and Bernstein, (2) Ho and Ramamoorthy, (3) Sinha and Natarajan.

The composite ranking of the candidates for implementation using the VMTP service is: Ho and Ramamoorthy (2), Wuu and Bernstein (3), Obermarck (5), Chandy, Misra, and Haas (6), and Sinha and Natarajan (7). As before, Obermarck's algorithm is dismissed as an unlikely candidate. In this case, the VMTP support of multicasting enables the algorithm of Ho and Ramamoorthy to retain its place above that of Wuu and Bernstein, and reinforces the edge that the simplicity of the algorithm of Chandy, Misra, and Haas has over that of Sinha and Natarajan. There is no compelling reason to adjust the relative ranks of the top three of the remaining four algorithms: (1) Ho and Ramamoorthy, (2) Wuu and Bernstein, (3) Chandy, Misra, and Haas.

The composite ranking of the candidates for implementation using the UDP service is: Wuu and Bernstein (2), Obermarck (6), Sinha and Natarajan (6), and Chandy, Misra, and Haas (7). As before, Obermarck's algorithm is dismissed as an unlikely candidate. Sinha and Natarajan's algorithm is ranked over that of Chandy, Misra, and Haas for the same reasons discussed under the mesh topology. Consequently, the final ranking of the three algorithms is: Wuu and Bernstein, (2) Sinha
and Natarajan, (3) Chandy, Misra, and Haas.

**Loop Topology**

The composite ranking of the candidates for implementation using the TCP service is: Obermarck (3), Ho and Ramamoorthy (5), Sinha and Natarajan (5), Wuu and Bernstein (7), and Chandy, Misra, and Haas (7). As before, Obermarck's algorithm is dismissed as an unlikely candidate. The large amounts of data transmitted by the algorithm of Ho and Ramamoorthy make it less attractive than the remaining algorithms, due to the potential impact of the large transmissions on the temporary storage at the intermediate nodes of the network. Since the connectivity of a loop network is fairly low, and messages travel across an average of one quarter of the segments in the network, an algorithm employing a minimal number of messages per cycle should be favored in the selection process. The algorithm of Wuu and Bernstein fits this specification, and becomes the first choice. Sinha and Natarajan's algorithm is better suited to the loop topology than that of Chandy, Misra, and Haas for three reasons: it uses site order to limit the number of messages sent, with the added effect of reducing the number of segments crossed when the loop is connected in site order; it does not employ multicasting; and, it (conceptually) requires fewer messages, so there is less likelihood of a bottleneck. As a result, the final ranking is: (1) Wuu and Bernstein, (2) Sinha and Natarajan, (3) Chandy, Misra, and Haas.

The composite ranking of the candidates for implementation using the VMTP service is: Obermarck (4), Ho and Ramamoorthy (4), Sinha and Natarajan (4),
Chandy, Misra, and Haas (5), and Wuu and Bernstein (6). As before, the algorithms of Obermarck and Ho and Ramamoorthy are dismissed as unlikely candidates. Wuu and Bernstein's algorithm maintains its rank for the reasons given in the discussion of the TCP service. The relative ranks of the algorithms of Sinha and Natarajan and Chandy, Misra, and Haas remain the same. As a result choices are: (1) Wuu and Bernstein, (2) Sinha and Natarajan, (3) Chandy, Misra, and Haas.

The composite ranking of the candidates for implementation using the UDP service is: Sinha and Natarajan (3), Obermarck (5), Wuu and Bernstein (5), and Chandy, Misra, and Haas (6). As before, Obermarck's algorithm is dismissed as an unlikely candidate. Wuu and Bernstein's algorithm is given a boost because of its minimal number of messages. Sinha and Natarajan's algorithm is ranked over that of Chandy, Misra, and Haas for the reasons discussed under the mesh topology. Consequently, the final ranking of the three algorithms is: (1) Wuu and Bernstein, (2) Sinha and Natarajan, (3) Chandy, Misra, and Haas.

Bus Topology

The composite ranking of the candidates for implementation using the TCP service is: Elmagarmid, Soundararajan, and Liu (3), Chandy, Misra, and Haas (4), Wuu and Bernstein (6), and Sinha and Natarajan (6). The algorithm proposed by Elmagarmid, Soundararajan, and Liu is significantly more complex than the others in this list, and can be effectively eliminated from contention. The randomizing effects of contention on the maximum transmission delay and the lack of multicasting support by the TCP
service lower the rankings of the algorithms of Wuu and Bernstein and Chandy, Misra, and Haas, respectively. As a result, the algorithm of Sinha and Natarajan claims the top spot. The simplicity of the algorithm proposed by Wuu and Bernstein gains for it the second place. The final ranking is: (1) Sinha and Natarajan, (2) Wuu and Bernstein, (3) Chandy, Misra, and Haas.

The composite ranking of the candidates for implementation using the VMTP service is: Elmagarmid, Soundararajan, and Liu (2), Chandy, Misra, and Haas (2), Wuu and Bernstein (5), and Sinha and Natarajan (5). As before, the algorithm of Elmagarmid, Soundararajan, and Liu is dismissed. There is no reason for altering the sequence indicated by the composite rankings of the remaining three algorithms, despite the effect of bus contention on the algorithm of Wuu and Bernstein. The resultant rankings are: (1) Chandy, Misra, and Haas, (2) Wuu and Bernstein, (3) Sinha and Natarajan.

The composite ranking of the candidates for implementation using the UDP service is: Chandy, Misra, and Haas (3), Wuu and Bernstein (4), and Sinha and Natarajan (4). Wuu and Bernstein's algorithm takes the top rank in this environment, as it has the fewest messages to be mismanaged by the unreliable delivery service. As has been seen before, Sinha and Natarajan's algorithm is ranked over that of Chandy, Misra, and Haas because of its treatment of unreliability. Consequently, the final ranking of the three algorithms is: (1) Wuu and Bernstein, (2) Sinha and Natarajan, (3) Chandy, Misra, and Haas.
Ring Topology

The composite ranking of the candidates for implementation using the TCP service is: Elmagarmid, Soundararajan, and Liu (4), Sinha and Natarajan (4), Wuu and Bernstein (5), and Chandy, Misra, and Haas (5). The algorithm proposed by Elmagarmid, Soundararajan, and Liu is significantly more complex than the others in this list, and can be effectively eliminated from contention. The simplicity of the algorithm proposed by Wuu and Bernstein, combined with the constancy of the maximum transmission delay between each pair of nodes, provides a powerful recommendation for this algorithm. Because the ring topology implicitly supports a site-order operation, and because the TCP service does not support multicasting, the algorithm of Sinha and Natarajan outranks that of Chandy, Misra, and Haas. The final ranking is: (1) Wuu and Bernstein, (2) Sinha and Natarajan, (3) Chandy, Misra, and Haas.

The composite ranking of the candidates for implementation using the VMTP service is: Elmagarmid, Soundararajan, and Liu (3), Sinha and Natarajan (3), Chandy, Misra, and Haas (3), and Wuu and Bernstein (4). As before, the algorithm of Elmagarmid, Soundararajan, and Liu is dismissed. The algorithm of Wuu and Bernstein again ranks first, for the reasons discussed above. Despite the support of site order processing, the algorithm of Chandy, Misra, and Haas edges out that of Sinha and Natarajan, due entirely to the functionality provided by the VMTP service. The resultant rankings are: (1) Wuu and Bernstein, (2) Chandy, Misra, and Haas, (3) Sinha and Natarajan.
The composite ranking of the candidates for implementation using the UDP service is: Sinha and Natarajan (2), Wuu and Bernstein (3), and Chandy, Misra, and Haas (4). Wuu and Bernstein’s algorithm occupies the top rank in this environment, as it has the fewest messages to be mismanaged by the unreliable delivery service. As has been seen before, Sinha and Natarajan’s algorithm is ranked over that of Chandy, Misra, and Haas because of its treatment of unreliability. Consequently, the final ranking of the three algorithms is: (1) Wuu and Bernstein, (2) Sinha and Natarajan, (3) Chandy, Misra, and Haas.
4 Conclusions about the Influence of the Network Environment on the Selection of a Distributed Deadlock Detection Algorithm

Several conclusions can be drawn from the preceding discussion of the interaction of a deadlock detection algorithm for distributed databases and the networking environment in which it must operate. First and foremost, the characteristics of the underlying network do have an effect on the performance of a deadlock detection algorithm. Not only does the network affect the efficiency and effectiveness of an algorithm, but also its basic suitability for implementation in that environment.

The aspects of a communications environment that are most significant to this discussion are the network topology and services. Included in the topology are the class of network, whether it is broadcast or point-to-point, the degree and form of connectivity of the individual nodes in the network, and the low level protocols that govern both the access to the communications channels and the routing of individual message packets from node to node within the network. The network services, on the other hand, are concerned with the attributes of the end-to-end logical connections between a pair of nodes, the communications overhead involved in establishing and maintaining those connections, and the format and content of the messages transferred across them.

Second, a deadlock detection algorithm for which the network topology and services are perfectly suited is not necessarily the best choice for implementation in
that environment. While the network characteristics may eliminate from contention some algorithms, and enhance or detract from the appeal of others, the relative merits of the deadlock detection algorithms themselves are likely to alter or completely rearrange a ranking based upon environmental preference. Neither the attributes of the algorithm nor the characteristics of the network are sufficient in and of themselves to justify a particular choice.

Finally, the more simple the communication requirements of a deadlock detection algorithm, in terms of the frequency, length, and destination of the messages being sent, the more versatile that algorithm is, and the more likely to be effective in environments other than those it prefers.
5 Summary

The communications requirements of a number of deadlock detection algorithms for distributed databases, selected from the four major categories of such algorithms, were identified. The resultant usage profile served to guide an investigation of the characteristics of networking solutions provided by the current technology. Two broad areas of influence were found; in the topology of the physical interconnections, and in the services provided by the network software. A number of options within each of these two areas were further discussed.

The interaction of a representative subset of the deadlock detection algorithms with the topological and service options available in a network were examined, resulting in an ordered list of the environmental preferences of each algorithm. Finally, the relative merits of the individual algorithms were used to adjust their composite ranking for each combination of network topology and service, and to identify the first, second and third choices for implementation within a particular environment.
References


The vita has been removed from the scanned document