ACCESS CONTROL FOR DISTRIBUTED DATABASES

ANTHONY McGUIRE

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the requirements
for the degree of Master of Computer Science at
Concordia University,
Montreal, Quebec, Canada

November 1978

ABSTRACT

Access Control for Distributed Databases

Anthony McGuire

A mechanism for access control in a distributed database is presented. The mechanism is composed of separate access controllers, one of which runs in each station of the computer network. A wait graph model is used to describe the waiting relationships between processes and the files of the database. The access control mechanism keeps, for each file, only the unstructured list of all files which precede that one in the wait graph. Except for simultaneous requests, this suffices to detect deadlock. In the case of simultaneous requests, deadlock may occur, but is automatically detected and recovered.

A variation of this access control mechanism is also presented. In this mechanism, deadlock is completely avoided by allowing only those edges which do not cause a loop, to be introduced into the wait graph.

Both mechanisms are compared on the basis of message traffic overhead in the network, CPU time requirements and storage requirements. It is shown that both mechanisms require low message traffic overhead, particularly for a database with high locality of reference.

An implementation of a Distributed Database Access

Control System for a network of PDP/11 minicomputers, is presented. Error handling in the access control model underlying this implementation is discussed.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my advisor, Professor Juern Juergens. His guidance, constant encouragement and our many discussions proved invaluable in bringing this thesis to fruition.

To Terry, go my special thanks for her unfailing support, and for the many unrewarded hours spent typing this thesis.

My gratitude is due to the Royal Bank of Canada, where the idea for this thesis originated, and who graciously afforded me time to pursue this research for several weeks, while in their employ.

My thanks go to the management of the Biomedical Engineering Unit, McGill University, for the use of their computing facilities during the implementation phase of this work.

Financial support of this research through the Natural Sciences and Engineering Research Council Canada (grant no. A3575) and the Quebec Ministry of Education (grant no. F.C.A.C. EQ-21) is gratefully acknowledged.

CONTENTS

INDEX OF FIGURES

CHAPTER 1

Introduction

With the introduction of computer network technology, many applications have evolved for which the distribution of a database is a natural approach. Although the distribution of data over a computer network allows for the efficient implementation of applications which are themselves naturally distributed, it exaggerates the problems of file consistency and makes control of the file access for concurrent users more complex than when the complete database is maintained at one central station.

1.1 Distributed Databases

Throughout the literature there is a lack of consensus on the definition of a distributed database [56]. For the purpose of this work, we define a distributed database as data which are distributed among the stations of a computer network. The data are split into components called files, such that all the files in the computer network form a unique collection of data.

The word file may also be replaced by segment or dataset or any other convenient word which describes a subset of the database.

1.2 Main Design Problems of Distributed Databases

The main problems in the design of distributed database systems have been: (1) the architecture of the network; (2) the allocation of copies of files to the stations of the network; (3) the control of access to the files of the database.

The architecture of the computer network is closely related to the applications which will be implemented on the network and the type of file access control which will be used [50]. The two main network structures are hierarchical, also called vertical, where one or more stations in the network exert control over other stations, and horizontal, where all stations in the network have equal control [5].

Historically, the problem of file allocation in a computer network has been the focus of the majority of theoretical research in the area of distributed databases [21]. Chu [12] developed a model for the allocation of copies of files among the stations of a computer network. The main objective of the model was to minimize the overall storage and transmission costs. Casey [8] investigated the relationship between the optimum number of file copies and the rate of update and query traffic to the database. Levin and Morgan [36,37] have extended the models developed by both Chu and Casey, by introducing an interdependency between the data and the programs which reference the data.

Recently, an increasing amount of research has been

done in the area of access control for distributed databases. However, many of the problems have yet to be satisfactorily solved [2]. Because of the problems associated with the distribution of control, centralised control has been the dominant strategy in distributed systems. IBM's System Network Architecture (SNA) is an example of a commercially available distributed system which uses centralised control [28,47].

Lately, work has begun on solutions to the problems of file consistency and deadlock prevention and detection in distributed databases [10,14,15,39,42,52,53]. Many of the proposed solutions have employed theoretical models for the deadlock problem in single computer systems and applied these models to the distributed control problem. This has tended to result in solutions which have a high network traffic overhead [14,39].

1.3 Motivation of This Work

Because research in the area of access control for distributed databases has only begun in recent years, satisfactory solutions have yet to be found for many of the problems. Solutions which have been proposed for the deadlock problem require either centralised control or need high message traffic overhead. For this reason, the aim of this work is to develop a distributed database access control mechanism which correctly maintains the database and, in particular, keeps it deadlock-free without incurring

high network message traffic overhead.

1.4 Outline of This Work

In chapter 2, a review of the literature on access control for distributed databases is presented. Then, in chapter 3, a distributed database access control algorithm is developed. This algorithm avoids deadlock in the majority of cases, and detects and recovers from deadlock when it does occur. A variation of this algorithm, which avoids deadlock in all cases, is developed in chapter 4. In chapter 5, we describe an implementation of the access control mechanism which was developed in chapter 3. Using the model which underlies this implementation, we present in chapter 6, a discussion of errors which may occur in the access control mechanism. We also describe mechanisms which may be used to reduce the occurences of these errors, and recover from them when they do occur. Chapter 7 contains a comparison of the two access control algorithms developed in chapters 3 and 4. These algorithms are also compared with a distributed database access control algorithm which is described in the literature [39,40].

CHAPTER 2

Literature Review

Below we present a review of the literature in the area of distributed databases. Because this area is vast, ranging from resource allocation in the network to data translation, the review is focused on those topics which are pertinent to this work. In particular, this chapter contains an exhaustive discussion of the literature on access control for distributed databases.

2.1 Distributed Databases

Schreiber [54,55] defines a distributed database as a set of files, distributed among the nodes of an information network, which are logically related in such a way as to constitute a unique collection of data. He reviews in detail the main problems of distributed database system design: network architecture, file distribution, file directory allocation and file access control. In a first step towards the solution of these problems, a multilevel model for a distributed database system is proposed. The model itself is large and cumbersome. However, Schreiber feels that the structure is necessary to provide full logical and physical data independence and to model both heterogeneous and homogeneous systems.

In the discussion of the file allocation problem,

Schreiber examines the major criteria for splitting a file into subfiles, so that the majority of references to a subfile originate at the station where the subfile is located. The criteria are high geographic and functional locality of reference of the database. The distribution of the data which results in high geographical locality of reference, referred to by Schreiber as horizontal distribution, is required by applications where the data is related to its geographic location. The vertical distribution, which results in a high functional locality of reference, is characteristic of distributed systems where different stations in the network perform different functions.

Together with Paolini and Pelagatti, Schreiber [50] uses the concept of horizontal and vertical distribution to examine distributed database systems on the basis of applications requirements. Among the applications which are reviewed are manufacturing control systems, inventory systems, banking systems, and computer aided design systems. The review of the various applications shows that a vertical partitioning of the data along with centralized access control are the most common distribution and control requirements.

2.2 Distributed Databases in Homogeneous and Heterogeneous Networks

Peebles, in his dissertation [52], is concerned with the problem of integrating diverse data access systems in a network. He proposes an Inter-Process Communication Facility (IPC) and an Access Process (AP) which can perform data access on behalf of a user process. A user process in one station of the network, uses the IPC to communicate with its AP. A user process has an AP in each station of the network from which it requests data.

In co-operation with Manning, Peebles continues his work on access control for distributed data [42]. However, in this paper, the mechanism which is described has been designed specifically for a homogeneous network. Using a criterion for the partitioning of data similar to that formulated by Schreiber, Manning and Peebles partition the data so that it exhibits a high geographic locality of reference. This criterion was used because the system is intended for transaction processing from commercial applications where most of the references to a particular component of the database originate in a particular geographic region. Examples of such applications are banking and retail credit card sales.

The objective of this system is to provide efficient data sharing among the stations of the network, with minimum CPU and communications overhead. The solution which is

described in this paper, employs two primitives; segments and tasks. All data objects, including messages, are segments. A task is an object which processes a message segment. All inter-task communication is achieved by passing message segments. A switch mechanism, resident in each station of the network, is dedicated to the passing of these segments. The protocols for intra- and inter-station communications are kept uniform so as to make the distribution of the data transparent to higher level software. A description of the implementation of this system on a two-host network of PDP-11 minicomputers, is given.

In a companion paper [43], Manning and Peebles, with Labetoulle, detail the analysis of the system described above, by simulation and modelling using queuing theory techniques. They found that agreement was good between the model and the simulation, particularly for high geographic locality of reference.

Chang [10] describes the design and implementation of a distributed database system. This system is based on the work done by Manning and Peebles in [42]. Chang's system is designed for a medical database which exhibits a high geographic locality of reference. This system is also implemented on a homogeneous network of PDP-11 minicomputers. Each station of the network has a database Machine (DBM), which has access to the files residing in this station. EAch DBM consists of a set of User Machines (UM), a File Machine (FM), and a Network Access Machine

(NAM). The user at a station of the network, is regarded as operating his own UM at that station. Data requests originating from a UM are directed by the NAM to the FM in the station where the requested data is stored. The FM responds to the UM via the NAM.

Chang points out that so far the system has only been implemented for one transaction type, viz., data retrieval. However, he notes that it will be feasible to extend the range of transactions to two or three classes.

The objective of the system described by Chupin [15] is to allow a collection of computing facilities to appear to the user as a single network facility. In this paper, the particular concern is with the control functions specific to a data bank application. The particular data bank system which is discussed, is SOCRATE [1].

The concept of a Logical Network Machine whose function is to coordinate user specified functions and processes, is discussed. In the particular data bank case it is called the Logical SOCRATE Network Machine (LSNM). The LSNM is functionally layered into a compiler level, an abstract memory level and a data-set level. Chupin notes that one of the main objectives of the data access control method is distributed control. Generalized network semaphores are defined to be used for locking purposes. A network semaphore is described by its location, sharing degree and name. However, Chupin indicates that the problems associated with

semaphore naming, and semaphore-to-object binding have not yet been satisfactorily solved.

An interesting approach to concurrency control in a distributed database on a heterogeneous network, is described by Bernstein et al [4]. In this system, which is being implemented on the ARPA network, the database is fully redundant to enhance reliability and responsivity. However, redundancy can increase the cost of updating data because of the extensive inter-station communication necessary to lock all copies of the data being updated. Global locking of data is avoided in this mechanism, by identifying transaction types where it is not necessary. The predefinition of transaction classes forms the basis for the identification of these transaction types. In an example of an inventory control system which is given, the authors show that 99.9% of the transactions do not require global locking of data.

2.3 Centralised and Distributed Access Control

Mahmoud and Riordon [38,39,40] specifically address the problems of centralised and distributed access control. Their main concern, however, is with the efficiency of the access control system itself rather than the specific requirements of any particular distributed application. Solutions are proposed for both the centralised control and the distributed control cases. In the centralised case, the access control mechanism, which is called the "Distributed Data Base Management Facility" (DDBMF), resides in one

designated station. This station is then recognized by all the other stations in the network, as the central control station. All file requests issued in the network are first sent to this central DDBMF. In the distributed control case, a DDBMF resides in each station of the network. A file request is first sent to the DDBMF in the same station as the requesting process. In order to avoid deadlock in file allocation, the solution proposed for the distributed control case employs a synchronised broadcast of file and queue status information from each DDBMF. This results in a high network message traffic overhead.

The two approaches are evaluated by simulation. The evaluation is based on message traffic overhead in the network, CPU time requirements and storage requirements. The solution for centralised control requires less CPU time and storage. This is due to the duplication of effort of each DDBMF in the distributed control case. The message traffic overhead varies with the locality of reference of the database for both cases, becoming lower as the proportion of local file requests increases. In the distributed control case, the message traffic overhead is higher than in the centralised case, when the proportion of remote access requests is higher than that of local access requests. However, the message traffic overhead decreases faster with the increasing locality of reference for the distributed control case than for the centralized control case. For high locality of reference, the distributed control case requires

the lower message traffic overhead.

2.4 Deadlock in Distributed Databases

Mahmoud and Riordon, together with Hutchinson, extend their concern with the deadlock problem in a distributed system to a technique for deadlock pre-emption [30]. The algorithm which they develop is aimed at the minimal cost of deadlock recovery by pre-emption. The author's claim that this algorithm is suitable for computer networks, is based primarily on its low CPU and storage requirements.

Chu and Ohlmacher [14] also examine the problem of deadlock in distributed databases. However, unlike Mahmoud and Riordon, a complete system design is not discussed. The focus of attention in the paper is a deadlock prevention and detection mechanism for a distributed access control system.

The authors propose three mechanisms - two deadlock prevention mechanisms and a deadlock detection mechanism. Both prevention mechanisms use a fixed examining path among the stations of the network, for the examination of remote file requests. The simple mechanism which they propose, prevents deadlock by requiring a process to declare in advance all the files which it intends to reference, prior to the initation of the process. The process is only allowed to begin when all its requested files have been allocated to it. In the second prevention mechanism a variation of Habermann's technique [26] of granting only safe requests,

is used. In order to make this mechanism efficient for the case of distributed control, the process set concept is introduced. All processes which have a pending request for the same file belong to the same process set. The progress of one process is independent of the processes which do not belong to its process set.

The deadlock detection mechanism which is proposed, maintains lists of processes and files and pointers between the lists denoting file requests and/ allocations. Deadlock is detected by scanning the pointers for the existence of a loop. This scheme is similar to the deadlock detection scheme used by Mahmoud and Riordon. Both schemes are based on work done by Murphy [49]. In comparing the three deadlock mechanisms, the authors point out that the simple prevention mechanism, although restrictive, is superior for most applications, since it is easily implemented and requires least system overhead.

Aschim [2] proposes a deadlock mechanism similar to the simple prevention mechanism noted above. Although Aschim's mechanism requires a process to declare all its files in advance, confirmation that a file is to be allocated to the process may be retracted if a process of a higher priority requests the same file. Aschim also proposes a fixed reservation scheme which prevents deadlock. This scheme gives a unique number or priority to each file. Then, processes are required to request files in the order of their unique numbers.

A different approach to the problem of access control in a distributed database is outlined by Rosenkrantz et al [53]. In this approach the user processes, which reference the database, "move" from station to station in the network according to their data requirements. Each station in the network has a local access controller of its own. The objective of the access controller is to ensure that the user process running in its station will eventually terminate; thus, deadlock is eliminated. It is also the objective of the access controllers to ensure that the database as a whole remains consistent.

To achieve the elimination of deadlocks, the access controllers in two remote stations communicate whenever a user process moves from one station to another, when a process terminates or aborts which has previously visited the other station, or when a proces which is involved in a database conflict, has previously visited the other station. The paper shows that the control concepts employed, work correctly. However, it does not indicate that this system has been implemented.

CHAPTER 3

The Access Control Model

In this chapter we discuss our philosophy of access control in a distributed database. We examine the behaviour of a graph model which describes the waiting relationships between processes and files in a computer system. We employ this graph model in the design of the basic algorithm of a distributed access control mechanism, which avoids deadlock in the majority of cases. This algorithm is then refined so that it detects and recovers from deadlock whenever it does occur.

3.1 Background

For the purpose of this study we make some assumptions concerning the general organisation of the distributed data base. We assume that the distributed database consists of a static population of files; that each file has a unique name in the network and resides at exactly one station, so that there are no multiple copies of files.

We also assume that a process may make only one file access request at a time. A process, which requires access to more than one file simultaneously, must make the requests individually, one request being granted or refused before another is made.

We assume that once a file has been allocated to a process, no other process may gain access to that file until it has been released. That is, we assume file access to be exclusive.

Although we envisage a distributed database with a high locality of reference, we make no restriction on the files to which a process may request access. A process running in an arbitrary station of the network may be granted access to any file of the database, regardless of the location of that file. However, a process executes in exactly one station of the network. Processes may be created and deleted at any time, but we do not allow a process to be deleted while it is in possession of a file or waiting for a file.

We also assume for processes, as for files, that each process has a name which is unique throughout the network. We do not make any particular assumption on how many processes execute in parallel in any one station, or on the nature of the support software in the various stations. We only assume that there is, in each station, a supervisory system which controls the process or processes, which execute in that station, on behalf of users, and which can implement such system functions as communication with other stations in the computer network.

3.2 Distributed Access Control

An access control mechanism processes all file access

requests. This mechanism runs in each station of the network. If a process, executing in station S, requires access to a particular file, then this process makes the file access request to the access controller which is running in station S. The process is then suspended. At some later time the process is resumed. Then the file access request has either been granted or refused. An access controller refuses a file access request if the requested file is currently allocated to another process and if allowing the requesting process to wait for the file would cause a deadlock. An access controller in a station maintains both a local and global file directory. The global directory indicates the station at which the remote files are stored. The local directory names all the files stored at the station in which the access controller is running and contains a pointer to the head of the queue for each local file.

When an access controller receives a file access request, it ascertains the location of the file by examining its local and global file directories. In a database which exhibits a high locality of reference, the majority of requests will be for local files. If the request is, in fact, for a local file, the access controller determines whether that file is currently available. If so, the file is allocated to the requesting process and that process is resumed. If, however, the file is currently allocated to some other process, the access controller determines if

allowing the requesting process to wait would cause a deadlock. If it would, then the request is refused. Otherwise, the request is placed on the queue for that file. If, at some later time, it is discovered that further waiting by the process for that file forms part of a deadlock, it may then be decided to refuse the access request.

If a process resquests a remote file, the access controller passes the request to the station where the file is stored. The access controller at that station then processes it as it would a local file request. When it grants or refuses the request, it communicates this to the requesting process via the access controller in the station in which that process executes.

When a process has no further need for a particular file, it communicates this fact to the file's access controller. This is accomplished via the access controller in the same station as the process, if the file is remote. The file is then free to be allocated to any other process which may be waiting for it.

3.3 Network Independence

In the design of the access control mechanism, we are striving for independence of the processes from the structure of the network, as well as from the distribution of the database in the network. For requesting and releasing

files, a process always communicates with the access controller executing in the same station as itself. The access controller is aware of the structure of the network and the distribution of the database, since it must be able to locate files in it. A process, however, need not be aware of these aspects of the network. A single process would behave in the same manner regardless of whether the access control mechanism of the database is distributed or centralized.

For the purpose of granting file access to processes, it is again not necessary to know the distribution of the database. Rather, it is sufficient to know, for each station, which processes currently possess which files and which processes currently wait for which files. This means that the maintenance of complete global file status information is not necessary. Indeed, it is only necessary that an access controller should have sufficient information in order to make decisions on granting its own locally stored files. We will see below that the amount of this information is considerably less than the complete global file status information.

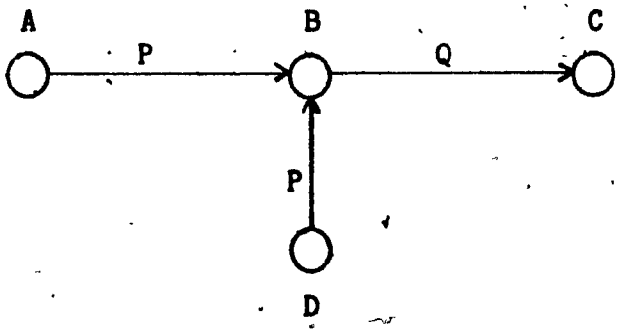
3.4 The Wait Graph

Shoshani and Bernstein [57] have investigated the deadlock problem in the specific context of databases, and they use a particular graph model for describing the status of the database. This "wait graph" model is very useful for

the present investigation.

A node of the wait graph represents a resource which is currently allocated to some process. A labelled directed edge represents a process which is in a wait state, because its resource access request cannot yet be granted. The source node of the edge represents a resource currently owned by the process, while the destination node of the edge is the resource for which the process waits. Each resource is represented in the graph by at most one node. For our purposes, the resources are the files of the data base (figure 3.1). It should be noted that a process which does not possess any files but is waiting for access to some file, is not represented in the wait graph. We do not need to represent such a process since its waiting cannot contribute to a deadlock.

FIGURE 3.1. The Wait Graph

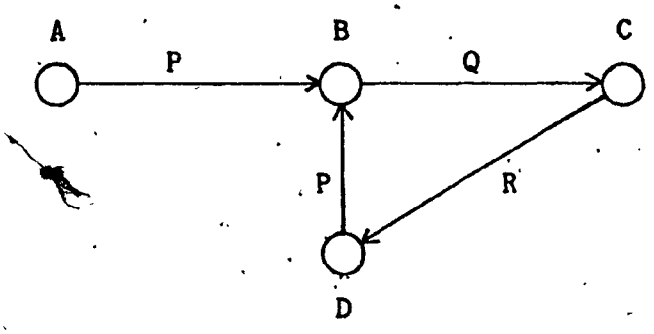


Process P, which possesses files A and D, waits for access to file B. Process Q, which posseses file B, waits for access to file C.

In this graph model, a directed loop is a necessary and sufficient condition for the existence of a deadlock [10].

Hence, deadlock can be avoided by examining the graph to see whether the introduction of a new edge, that is, allowing some process to wait, would cause a loop in the graph. Deadlock can be detected by examining the graph for the existence of a loop. If detected, a deadlock can be broken by removing one edge from the loop, corresponding to refusing one access request (figure 3.2).

FIGURE 3.2. A Loop in the Wait Graph



Files B, C and D are in a deadlock. This deadlock can be broken by removing one of the edges P, Q or R, in the loop, i.e., by rejecting P's request for B, Q's request for C, or R's request for D. (If P's request for B is rejected, then both the edges P(A,B) and P(D,B) are removed.)

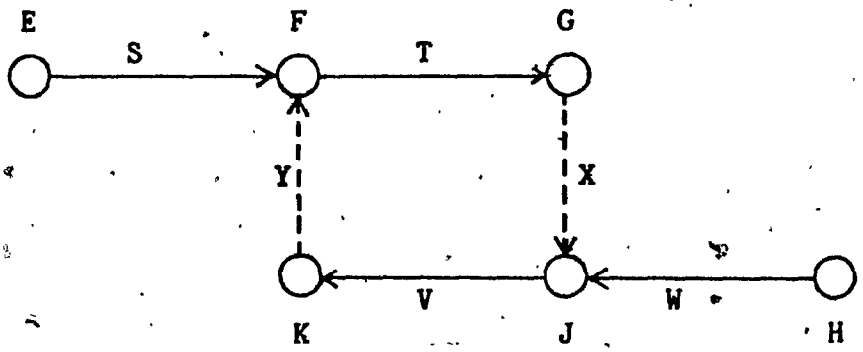
Since a process may only request one file at a time, there can be at most one edge emanating from each node in the graph. However, a node may be the destination of many edges.

For each connected component in the wait graph, there is at most one node from which no edge emanates. If such a node exists, we call it the "end node" of the connected component. If no such node exists, the connected component

contains a loop. By contracting the loop to a single node, we obtain a connected component which has an end node, and this end node is the node which we created by the contraction.

As a result of this, a loop may form only at the end of a connected component in the graph or by the joining of two or more connected components at their ends (figure 3.3). This also implies that there cannot be more than one loop in each connected component, and that any one file can be part of only one loop at a time. In any one connected component, the edges which are incident on the end node are the only ones which can be removed by the granting of the corresponding access request. This is because the owner of that file is the only process in the connected component which is not waiting and so the only one which can release a file.

FIGURE 3.3. A Loop Caused by "Simultaneous" Actions in the Wait Graph



The loop (F,G,J,K) is formed by the simultaneous introduction of the edges X(G,J) and Y(K,F). Nodes G and K are the only nodes in the two connected components, (E,F,G) and (H,J,K) respectively, from which new edges can emanate.

3.5 Deadlock Avoidance and Detection

The task of avoiding deadlock can be rephrased in terms of the wait graph model by saying that no process may be allowed to enter a wait state such that any one of the edges representing this wait state is part of a loop in the graph. To obtain an operational version of this principle, we use the concept of predecessor: a node D is a predecessor of a node C in the wait graph, if there is a directed path from D to C. Then the node D is in a loop if and only if D is its own predecessor, (Figure 3.2). Consequently, whenever the access controller has to decide if a particular process should be allowed to wait for a file, it checks whether the introduction of the new edge or edges in the graph would make the requested file its own predecessor in the wait graph.

This method enables the access controller to avoid deadlocks, provided the file requests are processed one at a time. This means that one request is decided and all resulting changes in the wait graph completed before the next request is processed. In a distributed system, however, this is not easily guaranteed. It may well happen that two or more processes executing in different stations, request file accesses in such a way that the processing of these requests by the various access controllers overlaps in time, and that therefore the decision on each request is based on information which is no longer accurate. If we do not wish to centralize the wait graph information and, thereby, the access control function, and if we do not wish to introduce a global synchronization of the distributed access controllers, then we have no simple way of guaranteeing that such deadlocks will not occur. In chapter 4 we discuss an access control algorithm which avoids deadlock completely, but which requires the introduction of some additional data structures. For the time being, however, we examine the algorithm in which deadlock may occur. Therefore, we must be able to detect deadlock, break it, and recover from it.

A deadlock can be detected if, after the wait graph has been updated to reflect all new waiting processes, there exists a loop in the graph. If the check for a loop is performed after the introduction of each new edge, then no deadlock can go undetected.

3.6 Basic Data Structures

In order to be able to allocate files and avoid deadlock, the access controller maintains the following information:

(1) For each process, a list of all the files currently in the possession of that process; a variable which points to the file for which the process is currently waiting, if any. If the process is not currently waiting for a file, this pointer has the value nil. This data structure will be referred to as the process descriptor.

(2) For each file, a list of all processes which currently wait for access to that file; a variable which points to the process which currently possesses the file. If the file is currently not allocated to a process, this pointer has the value nil. This data structure will be referred to as the file descriptor. Each access controller in the network maintains a process descriptor for each process executing in its own station, and a file descriptor for each file stored at its own station.

Information on the wait graph, as described above, is also maintained by the access control mechanism. However, as noted, it is sufficient to know the predecessors of each file in the wait graph in order to avoid and detect deadlock. The list of predecessors is maintained by the access controller for each file stored at its own station.

This data structure will be referred to as the predecessor list.

The predecessor list of a file need not reflect the structure of the wait graph. In order to avoid or detect a deadlock, it is sufficient to know the set of all those files which precede a given file in the wait graph. This observation, along with the assumption that the population of files in the network is static, enables us to implement the predecessor list of each file as a bit list.

3.7 Maintenance of Predecessor Lists

Whenever a new edge is introduced into the wait graph, the predecessor list of the destination node of this edge must be augmented by that of the source node. Moreover, if the destination node is itself a predecessor of one or more other nodes, then the predecessor lists of these nodes must also be augmented by that of the source node. This task is greatly facilitated by the fact that, for each node F in the wait graph, there can be at most one emanating edge. The destination of such an edge will be called the immediate successor of F. Therefore, in order to allow the update of the predecessor lists, we maintain an immediate successor pointer for each file. If the file is an end node of a connected component in the wait graph, this pointer has the value nil.

We have noted already that removal of an edge from the

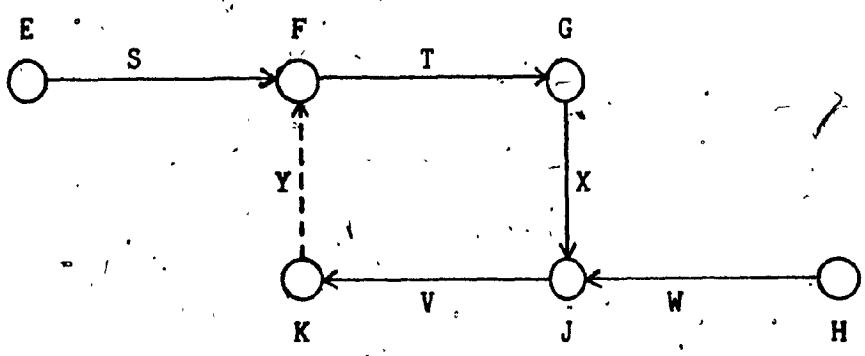
wait graph, due to the granting of a file access request, may only happen at the end of a connected component. More precisely, an edge can be removed from the graph only if its destination node is an end node of a connected component. Consequently, the removal of such an edge affects only the predecessor list of the destination node of that edge. The change to such a nodes's predecessor list consists of erasing the source node of the removed edge, as well as the members of that node's predecessor list.

If the access controller detects a loop in the wait graph, that is, a deadlock, it breaks it by rejecting one of the file access requests which form the edges of the loop. This corresponds to removing an edge from the wait graph. But this edge is not one whose destination is an end node. On the contrary, its destination node is a predecessor of all the other nodes in the loop, and the removal of the edge must be properly reflected in the predecessor lists of all successor nodes in the loop.

This update of predecessor lists after the removal of an edge from a loop, is complicated by the fact that we do not have any structural information on the wait graph: if file E is in the predecessor list of file F we know that there is a directed path from E to F in the wait graph. However, we do not know whether the removal of an edge which is incident on F, should be reflected in F's predecessor list by the deletion of E, (figure 3.3).

We therefore propose that, whenever a loop is broken by the removal of an edge, the predecessor lists of all files in the loop are reset to empty, and, starting at the node which was the destination of the removed edge, these predecessor lists are rebuilt in turn. This is possible if it is known, for every file, which files are its predecessors via a path of length one in the wait graph, that is, which files are its immediate predecessors. The new predecessor list is then rebuilt by forming the union of the predecessor lists of the file's immediate predecessors and the immediate predecessors themselves. This includes the immediate predecessor, in the loop, which we know to have been already rebuilt (figure 3.4). Consequently, we maintain, for each file in the wait graph, a list of immediate predecessors.

FIGURE 3.4. Breaking a Loop in the Wait Graph



The loop (F,G,J,K) is broken by removing the edge Y(K,F). The predecessor lists of files F,G,J, and K must be rebuilt as follows: from {E,F,G,H,J,K,} to {E} for file F; to {E,F} for file G; to {E,F,G,H} for file J; to {E,F,G,H,J} for file K. The predecessor lists of files E and H remain unchanged.

In summary, the data structures which represent the wait graph are predecessor list, immediate predecessor list and the immediate successor pointer.

3.8 The Basic Access Control Algotithm

We first present a simple algorithm which does not allow for simultaneous requests for files. In the next section, we will refine this algorithm in such a way that it detects and recovers from deadlocks due to simultaneous requests.

When a process makes a file access request, the access controller locates the file in the network. The access controller in the station where the file is stored, examines the requested file's descriptor. If the file is currently free, the process is granted access to it immediately. The process and file descriptors are updated accordingly.

If the file is not currently free, then the access controller checks whether the process has already been granted other files. If this is not the case, then the process is entered into the queue of the requested file, and the process and file descriptors are updated accordingly.

If the requesting process already possesses one or more files, then it must be ascertained whether allowing the process to wait causes a deadlock. The process's access controller does this by examining the predecessor lists of all files currently in the possession of the process, to see

whether the requested file is already a predecessor of one of these files, in the wait graph. If so, then allowing the process to wait would cause a deadlock and, hence, the file access request is refused. If not, the process is allowed to wait and is entered on the requested file's queue. The process and file descriptors are updated accordingly. The immediate successor pointer of each of the process's current files is updated to point to the requested file.

Once a process is allowed to wait for a file, that file's predecessor list must be updated. If the process has no files currently in its possession, then there is no change to this list. Otherwise, all files in the possession of the process are added to the immediate predecessor list of the requested file and to its predecessor list. The requested file's predecessor list is further augmented by those of the process's current files, that is, of the new immediate predecessor files, as well as these new immediate predecessors themselves. Since immediate predecessor and predecessor lists are implemented as bit lists, updating these lists amounts to an OR operation.

Once a predecessor list has been updated, it is used to update the predecessor list of that file's immediate successor in the wait graph. This "predecessor list propagation" stops at the file which is the end node of the connected component, that is, the file which has no immediate successor in the wait graph. (The case where there is no end node is discussed below.)

3.9 The Refined Access Control Algorithm

The basic algorithm described above works correctly if files are requested sequentially. However, as noted above, several independent requests may be processed simultaneously, which may lead to a deadlock. We refine the above algorithm to detect and recover from such deadlock.

We first assume that there is, associated with each file in the database, a component of the access control mechanism, which maintains the predecessor list, immediate predecessor list, immediate successor pointer and file descriptor for this file. We call such a component a local access controller (LAC). In this way we develop the algorithm in terms of a virtual network of LACs which may be abstracted from the distribution of the files in the database. Since there is a one-to-one correspondence of LACs and files, we will occasionally use the terms "LAC" and "file" interchangeably. Moreover, for files represented in the wait graph, we will sometimes use the term "node" instead of "file" or "LAC".

We then assume that the files of the database are in some fixed static order such that we may say some one file is "higher" or "lower" than some other file in that order. One such order is given by the representation of the predecessor lists as bit lists.

In order to detect deadlock, we modify the above algorithm in the following way: each time a LAC receives a

list of files which is intended as an update to its predecessor list, it does not perform the update immediately. Instead, it checks whether its own file is a member of the received list. If this is the case, then its file is a member of a loop in the wait graph, and that means it is deadlocked. If, however, the LAC's file is not a member of the list received, then the LAC updates its predecessor list as usual and passes the updated list on to its immediate successor, if any, which then acts in the same fashion. If no deadlock is detected, then the predecessor list propagation will terminate at an end node's LAC.

When a LAC detects a deadlock, it switches to "deadlock mode": the LAC's normal functions of predecessor list updating and propagation are suspended, and only file descriptor and immediate predecessor list updating still continues. The updating of predecessor lists can be safely suspended, since a file will remain deadlocked until the deadlock recovery routine has been performed. The deadlock recovery routine will correctly rebuild all these predecessor lists.

When a LAC discovers a deadlock, it assumes responsibility for breaking it. However, several LACs, whose files are nodes in the loop, may discover the existence of the loop. If each of them decides on its own to break the deadlock, more than one edge may be removed from the loop, i.e., more pending requests than necessary may be rejected. Therefore, it is desirable to have some mechanism by which

one LAC will be selected. This LAC will then break and recover from the deadlock. To accomplish this, we use the static order among files, which was introduced above.

A LAC which detects a deadlock sends a message to its file's immediate successor. This message indicates that a deadlock has been detected and contains the name and static order number of the originating LAC's file. The LAC which receives this message will either have detected the same deadlock, since its file is part of it, or will not have detected the deadlock and be functioning normally. If it has already detected the deadlock, it will have suspended its normal functions, as described above, and sent a "deadlock" message of its own to its file's immediate successor. Otherwise, it switches to "deadlock mode" on receipt of the message.

In either case, the LAC, when it receives the message, compares its own file's static order number with that in the message. If its file's number is lower, it passes the message to its file's immediate successor, which is also a file in the loop. Otherwise, it destroys the message and originates a "deadlock" message of its own, if it has not already done so. It then continues processing in deadlock mode.

By this mechanism, only one message will travel completely around the loop. That is the one originated by the LAC whose file has the highest number in the loop. This

LAC recognizes its own "deadlock" message when it receives it back. It then assumes the responsibility for breaking and recovering from the deadlock. Every other LAC, which has a file in the deadlock, will have passed on at least this one "deadlock" message and, so, knows not to assume this responsibility, but to remain in deadlock mode until it receives a deadlock recovery message. We will refer to the LAC, which has thus been selected to break and recover from the deadlock, as the "boss".

The boss will break the deadlock by refusing the pending request for its own file, which forms an edge in the loop. There can be only one such edge, since, as noted above, a file can be in only one loop at a time. But before this edge is removed from the loop, the boss must prepare for rebuilding the predecessor lists of all files in the loop, to reflect the rejection of this request. The boss first removes from its immediate predecessor list the source node of the edge it wishes to cancel. The boss knows this node as the one from whose LAC it received its own deadlock message. Also, the boss requests this LAC to invalidate its immediate successor pointer. If the process, whose request was represented by the cancelled edge, possesses other files as well, then all these files will also be removed from the boss's immediate predecessor list, and all their immediate successor pointers will be reset to nil (figure 3.4). When this has been done, the boss rebuilds its own predecessor list by forming the union of the predecessor lists of its

immediate predecessors and the immediate predecessors themselves.

The rebuilt predecessor list now reflects the removal of the one request from the graph. The boss then sends a "deadlock recovery message" to its own file's immediate successor. The LAC which receives this message will, as a result, rebuild its file's predecessor list using those of its file's immediate predecessors. This includes the immediate predecessor in the loop whose predecessor list has also been rebuilt, and, so, a predecessor list rebuilt in this way reflects the removal of the one request from the graph. The LAC then passes on the deadlock recovery message to its own file's immediate successor, and returns to normal functioning.

The deadlock recovery message is passed completely around the loop, and each predecessor list of a file in the loop, is, in turn, correctly rebuilt. The boss then receives back and recognizes its own deadlock recovery message. It then refuses the pending request for its own file. At this point the deadlock has been broken and recovery completed. The boss returns to normal functioning.

During the recovery from a deadlock, all processes which access the database may proceed as usual. The processes in the deadlock are, by the very fact that they are in a deadlock, suspended. Therefore, they can neither release nor request a file. The process whose request is

rejected in order to break the deadlock, is also kept waiting until the status information for all files in the loop has been correctly rebuilt.

A process which is not in a deadlock may request one of the deadlocked files. If this process possesses any files, then the immediate successor pointers of these files, as well as the immediate predecessor list of the requested file, are updated in the usual fashion. If the LAC of the requested file is still in deadlock mode, it will have a correct immediate predecessor list when it later receives the deadlock recovery message. If the LAC in question has already received and honoured the deadlock recovery message, it will update its predecessor list and pass it on to its immediate successor in the usual fashion.

In some networks, we could not exclude the possibility that a "predecessor update", when travelling around the loop, overtakes the deadlock recovery message. As a result, a LAC, which is still in deadlock mode, may receive an "update predecessor list" message. Operating in deadlock mode, the LAC will ignore such a message. When it later processes the deadlock recovery message, it will automatically generate a correct predecessor list.

It should be noted, that the algorithm will usually not have to perform deadlock recovery at all. In the proposed mechanism, deadlocks cannot occur unless access requests are issued simultaneously and then conflict in the rather

special way described above (figure 3.3). Usually, the algorithm will avoid deadlock and will do so without incurring high overhead. However, if a deadlock should happen, it will be detected and recovered from correctly.

The algorithm which we described above will be referred to in subsequent chapters as the "deadlock avoidance and detection" algorithm.

CHAPTER 4

Complete Deadlock Avoidance

The deadlock avoidance and detection algorithm, described in chapter 3 above, will avoid deadlock in the most usual cases. When a deadlock does occur, the algorithm will detect it and recover from it. Described below is an alternate approach to the access control problem. This approach uses a varation of the wait graph structure as described in section 3.4 above, to depict the waiting relationships between the processes and files in the network. However, in this algorithm we do not allow a new edge to be introduced into the graph until it is certain that it will not form part of a loop. Thus, deadlock is completely avoided.

4.1 Assumptions

All the assumptions concerning the database and the access controller which were described for the avoidance and detection algorithm, are also used in this algorithm. These assumptions are summarised as follows: the database exhibits a high locality of reference; the files of the database are static in both number and location; each file has a unique name; there are no multiple copies of files; file access is exclusive; a process may make only one file access request at a time; there is an access controller in each station of

the network; a process always addresses its file access requests to the access controller running in its own station.

4.2 The Modified Wait Graph

To distinguish the wait graph which was introduced in chapter 3, from other graphs which we will employ, we call that wait graph the W-graph. A variation of the W-graph is used in this approach to the problem. In fact, the W-graph is used in its entirety, but with the addition that a file which is allocated to a process and which has no process waiting for access to it, is also represented as a node in the graph. Such files form nodes in the W-graph which have no incident or emanating edges. These nodes may be regarded, in the usual sense of the W-graph, as end nodes, since they have no immediate successor. We call this wait graph the MW-graph.

4.3 Deadlock Avoidance

The initial processing of a file access request is identical to that described in section 3.8, the initial processing of the avoidance and detection algorithm, up to the point where it must be determined whether allowing a process to wait for a file would result in a deadlock. This deadlock check, however, is different in this algorithm. Effectively, the access controller determines whether the introduction of the new edge or edges into the MW-graph,

representing the waiting process, would cause a loop in the graph. It does this by 'walking' the connected component of the MW-graph which contains the node representing the requested file, until it reaches the end node of the connected component. The 'walk' starts at the requested file's node and proceeds in the direction of the graph.

Since a connected component of the MW-graph has at most one end node, a new edge can be made incident on any node in that component without the possibility of causing a loop, if it is known that no new edge is being created at the same time, emanating from that end node. Hence, the access controller determines if the process which owns the end node file, has a file request currently being processed. If it has not, then it is safe to introduce the new edge into the graph. Thus, a process is allowed to wait for a file only after it has been determined that this waiting does not result in a deadlock. This is done by ensuring that there is no other "simultaneous" action which might interfere with allowing the process to wait. However, if the process which owns the end node has an outstanding file request currently being processed, then the access controller waits until the processing of this request has been completed.

Because the access controller waits for the completion of a process which itself may be suspended, there exists the possibility of cyclic waiting. This waiting results from the fact that we do not allow an end node in the MW-graph to become a predecessor of another end node which is about to

become the source of a new edge. For this reason we regard such an end node as a 'shut' node. A shut node in the MW-graph represents the intended introduction of the new edge in the graph, of which that node will be the source. The deadlock avoidance scheme can be rephrased in terms of these shut nodes; a new edge will not be introduced into the graph as long as it would be incident on a connected component which contains a shut node. The result of this is that the access controller must wait until the shut node becomes open again.

Because of the possibility of cyclic waiting, the access controller determines if its waiting would result in a deadlock. If it would, then the access controller rejects the file request which it is processing. Otherwise, it suspends the processing of the request until the shut node on which it waits has become open again. At that time, the access controller resumes the processing of the request. If the end node has remained an end node, the new edge can now be introduced into the MW-graph because it is now safe to do so. However, if the node has now an immediate sucessor in the MW-graph, the access controller repeats the process of finding the end node and determining whether the new edge can be introduced or not. It does this until the edge is introduced and the request placed on the file queue or until the request is rejected due to a potential deadlock.

4.4 The P-graph Model

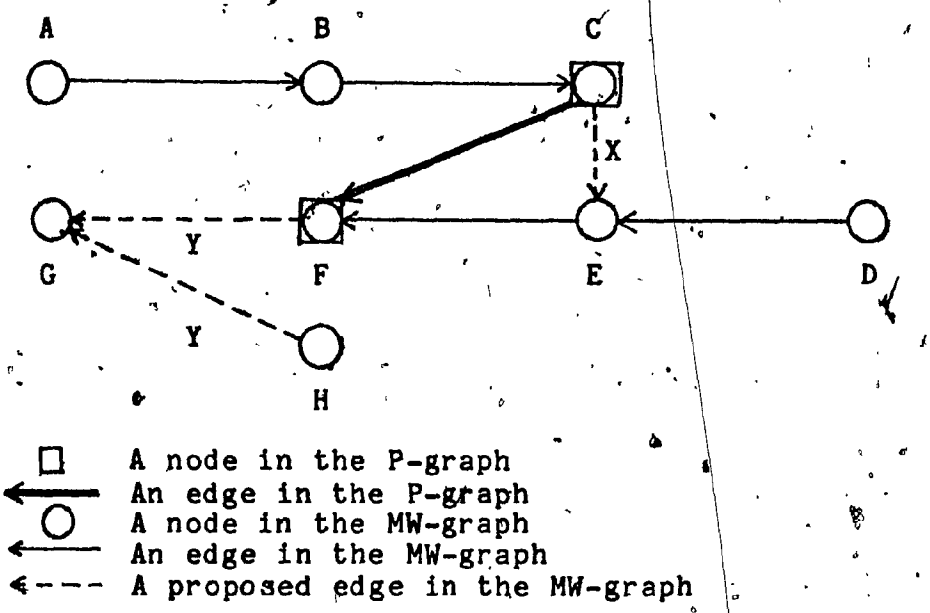
In the avoidance and detection algorithm, our version of a wait graph, the W-graph, was sufficient to describe the waiting relationships between the processes and files in the network. In this approach also, a variation of the W-graph, the MW-graph, describes the waiting relationships between the processes and files. There is, however, the additional need to describe the waiting relationships which can exist between shut nodes of this graph. We have again found it useful to use a graph model to describe these waiting relationships. Because of the fact that shut nodes are the sources of 'proposed' new edges in the MW-graph, we call this graph the P-graph.

A node in the P-graph represents a file which is owned by a process. That process has a file access request currently outstanding, and the access controller is determining whether a new edge or edges, representing the request, may be introduced safely into the MW-graph.

A labelled directed edge in the P-graph represents the suspended processing of an access request from a process X. The source node of the edge is a file owned by process X. The destination node of the edge is the file which is the end node in that connected component of the MW-graph which contains the requested file. This file is owned by some other process, Y, which also has an access request being processed. The edge in the P-graph shows that the access

controller cannot continue the processing of the access request from process X, until the processing of the request from Y has been completed (Figure 4.1). It should be noted that a node in the P-graph is also an end node in the MW-graph. This is because new edges in the MW-graph can only emanate from end nodes.

FIGURE 4.1. The P-graph



The edge (C,F) in the P-graph shows that the processing of the access request from process X for file E is suspended until the processing of Y's request for file G has been completed.

A node is introduced into the P-graph when the access controller wishes to create a new edge in the MW-graph and must determine that it is safe to do so. In this case, all the files owned by the requesting process are shut and therefore become nodes in the P-graph. A node is removed from the P-graph when the processing of the request has been completed, i.e., when the corresponding new edge has

actually been introduced into the MW-graph or the request has been rejected because deadlock would have resulted had the process been allowed to wait for the requested file.

As in the case of the W-graph, a directed loop in the P-graph is also a necessary and sufficient condition for the existence of a deadlock. In the P-graph a loop describes a deadlock between the suspended processing of requests. As in the W-graph, such a deadlock can be avoided by examining the P-graph to see whether the introduction of a new edge would cause a loop in the graph. Deadlock in the P-graph can be detected by checking the graph for the existence of a loop. If found, a deadlock can be broken by removing one edge from the loop. This corresponds to rejecting one of the suspended access requests which have formed the deadlock.

The behaviour of the P-graph is identical to that of the W-graph. A P-node can have only one emanating edge, but may be the destination of many edges. A connected component in the P-graph has at most one end node. If no end node exists then the component contains a loop. As a result of this, a loop in the P-graph can only form at the end of a connected component. As in the W-graph the edges representing two or more distinct access requests may be introduced into the P-graph simultaneously. These edges combine the two or more connected components in such a way as to form one connected component which contains a loop. The loop is at the end of this new component since it contains at least the last edge in each of the subcomponents

which together form the new connected component.

4.5 The P-graph and MW-graph Combined

We can now approach the problem of avoiding and detecting deadlock in the network in terms of the P-graph model. The P-graph reflects the fact that new edges are about to be introduced into the MW-graph. The P-graph can be superimposed on the MW-graph, with nodes in the P-graph corresponding to shut nodes in the MW-graph. Because new edges emanate only from the end nodes in the MW-graph, the P-graph may be obtained from the the MW-graph by selecting all connected components of the MW-graph which contain a shut node and by contracting these components to a single node. The edges in the P-graph show the waiting relationships between the suspended processing of the access requests from the owners of the shut node files.

A desired new edge in the MW-graph is not introduced until it has been detemined that the corresponding new edge in the P-graph is not an edge in a loop. However, if a loop does occur in the P-graph, the removal of one of its edges in order to break the deadlock, is reflected in the MW-graph by the rejection of the corresponding access request. That means the desired edge which would have eventually resulted in a deadlock is not introduced into the MW-graph.

For these reasons, if, in the P-graph, deadlock is avoided in the majority of cases and detected and broken

when it does occur, then we can guarantee that no deadlock will occur in the MW-graph. This means that deadlock among the processes and files of the network is completely avoided.

4.6 The Complete Deadlock Avoidance Algorithm

Because the behaviour of the P-graph is identical to that of the W-graph in the previous approach, we can use identical data structures and an algorithm identical to the avoidance and detection algorithm to avoid and detect deadlock in graph. For the purpose of the present algorithm, the MW-graph is described by the immediate successor pointer. No other data structure is maintained for this graph.

That part of the complete deadlock avoidance algorithm which manipulates the P-graph data structures, is identical to the algorithm described in section 3.9 for avoiding, detecting and recovering from deadlock in the W-graph. However, in this algorithm, a LAC maintains the immediate sucessor pointer of its file in the MW-graph as well as the P-graph data structures of its file. Nevertheless, the LAC only maintains P-graph information when its file is a node in the P-graph. Because a file may be a node in the P-graph only while its owner's request is being processed, a LAC does not manipulate its file's P-graph data structures except at this time.

We have described an algorithm for an access controller which completely avoids deadlock among the files of a distributed database. We will refer to this algorithm in subsequent chapters as the "Complete Deadlock Avoidance" algorithm. The algorithm is more complicated and uses more elaborate data structures than the deadlock detection and avoidance algorithm discussed in chapter 3. However, the additional complexity of the algorithm and its data structures is compensated for by the fact that it requires less message traffic overhead than the previous algorithm, particularly in the case of database with high locality of reference. The comparison of the two algorithms is discussed in chapter 7.

CHAPTER 5

The Implementation of the Distributed Database Access Control System

In this chapter we discuss the implementation of an access control system for distributed databases. We call this system the Distributed Database Access Control System (DDACS). The access controller which runs in each station of the network is designed using the algorithm, described in section 3.9, which avoids deadlock in the majority of cases and which detects and recovers from deadlock if it does occur.

5.1 Common Functions of LACs

In the discussion of our access control philosophy we have used the concept of the Local Access Controller (LAC). The local access controller is abstracted from both the structure of the network and the distribution of the files of the database. However, the implementation of the theoretical model of the access control mechanism should incorporate the characteristics of the real network which would be relevant to the efficient execution of the mechanism.

The common functions of the LACs which run in the same station, can be combined in one access controller for that station. The function of this access controller is to

receive all file access requests from user processes in the same station, and file access requests which have been directed to that station from some remote station. The access controller also maintains the local and global file directories in its station.

Because of the assumed high locality of reference of the database, we can expect that, in the majority of cases, complete connected components of the wait graph will be contained in one station, i.e., all the files represented in the connected component are resident in that one station. We can also expect that, where this is not the case, large continuous portions of the component will be contained in one station. In these cases, the access controller performs the predecessor list propagation routine on the predecessor lists of those files which form a continuous portion of a connected component of the W-graph in its station. This eliminates the need for predecessor list propagation messages between LACs resident in the same station.

In the case of deadlock recovery we can also expect large continuous parts of the deadlocked component to be contained in a particular station. In this case the action caused by "boss selection" and "deadlock recovery" can be performed by the access controller by repeating the action for every file which is a node on a continuous portion of the W-graph loop contained in its station. This speeds up the deadlock recovery routine by reducing the requirement for message passing between LACs.

In general, whenever some information causes a LAC to perform an action on its data structures, and must be propagated along a connected component, the access controller performs the necessary routines. It is more effecient to have the access controller perform the required action in the appropriate data structures for all files along that portion of the connected component which is in its station, than to have the message passed from LAC to LAC.

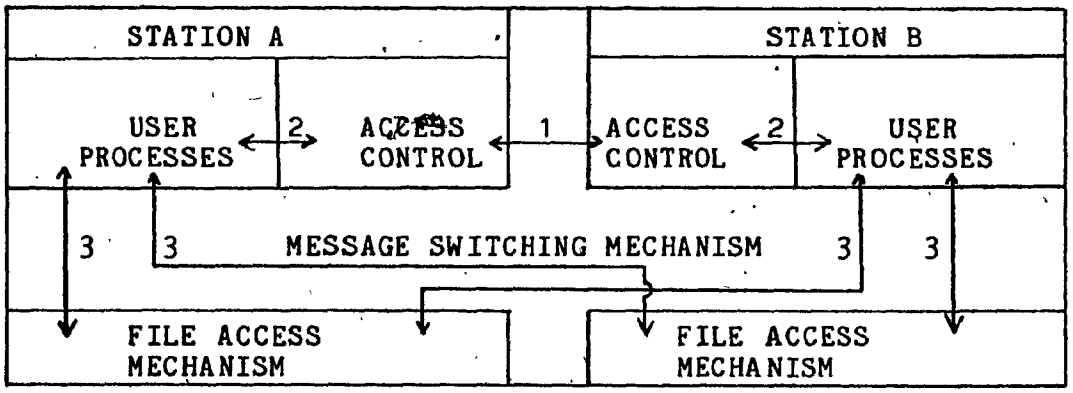
5.2 The Access Controller in the DDACS

In the Distributed Database Access Control System, there is one access controller in each station of the network. The design of the access contoller is based on the algorithm described in section 3.9. A user process addresses all file access requests to the access controller which runs in the same station as itself. If a process requests a local file, the access controller itself processes the request. However, if the request is for a remote file, the access controller passes the request to the access controller in the remote station where the requested file is stored. This scheme is also followed for the releasing of files.

In the model of the access controller which was implemented, there is no communication link between the access controller and the local access controllers of the files. Further, there are no communication links between the local access controllers themselves (Figure 5.1). This is

because the LAC is a conceptual tool for dealing with the maintenance of the data structures associated with a file. In the implementation itself, these data structures are maintained by the one access controller, so that no message passing is necessary. However, the concept of the LAC is still maintained within the access controller and may be regarded as the access controller itself when it is maintaining the data structures of a particular file. When it switches to those of another local file, it can be regarded as "assuming the identity" of that file's LAC. This essentially serializes the processing of the LACs in a station, so long as the access controller is implemented as a sequential process.

FIGURE 5.1. The Communication Links in the DDACS



The communication links in the DDACS are:
(1) Between access controllers in remote stations for passing control messages
(2) Between user processes and the access controller in the same station
(3) Between user processes and the File Access Mechanisms where their required files are stored

Control messages between the access controllers will consist of the following :

(1) Remote access request

(2) Request granted/rejected

(3) Predecessor list propagation

(4) Update immediate successor pointer

(5) Deadlock message (Boss selection)

(6) Deadlock recovery

Between a user process and its access controller, the messages will be:

(1) File access request

(2) File release

(3) Request granted/rejected

5.3 Requirements from System Software

In order to implement the access control mechanism as described above, it is necessary that a Message Switching Mechanism should be part of the underlying distributed system. The Message Switching Mechanism should provide a communication channel between any two processes in the network which wish to communicate with each other.

It is also necessary that there exist a File Access Mechanism which would perform the actual file handling on behalf of a process, once that process has been granted access to the desired file. In order to do this, the File Access Mechanism must be resident in each station of the network. Also, the File Access Mechanism in one station

should be able to communicate with a process in a remote station via the Message Switching Mechanism.

In order to make the distribution of the files of the database transparent to the user process, it is necessary that file request and release, and file access routines should be available to these processes. Further, these routines should not require the user process to know the location of the files in the distributed database. The routines will be of the type

GET FILE (file,return code),

RELEASE FILE (file, return code),

READ FILE (file, buffer, return code),

WRITE FILE (file, buffer, return code).

5.4 Available Hardware and System Software

The Distributed Database Access Control System was implemented for a network of Digital Equipment Corporation (DEC) PDP 11 minicomputers, consisting of one PDP 11/70 and one PDP 11/40 machine.

The operating system running on the network stations during the implementation was RSX/11M version 3.1 [66]. This operating system provides an interactive environment for software development. Also available on this system is a File Access Mechanism, FILES/11 and a Message Switching Mechanism, DECNET V1.2 [64].

The Message Switching Mechanism, DECNET V1.2, provides

a message service between processes in the same station, or in remote stations and between processes and the File Access Mechanism. The DECNET facilities which are necessary for the implementation of the access controller are as follows:

(1) To deliver a message from a sender process to a receiver process.

(2) To acknowledge the sender process when the message is received by the receiver process.

(3) To allow the sender to continue processing once the message is sent and acknowledged.

(4) To allow the sender to suspend itself, if desired, by waiting for a reply to a particular message.

(5) To buffer messages for the receiver if it is busy, so that the message may be extracted whenever the receiver desires.

(6) To resume the receiver, if suspended, upon the arrival of a message from a particular sender.

These services are invoked through the use of subroutines, one for each available service, which are called from the processes. These subroutines, which are stored as part of the system library, are designed to interface with programs written in FORTRAN or MACRO/11, the PDP/11 assembly language. The message switching protocol is completely contained within DECNET and is transparent to the user.

The File Access Mechanism, FILES/11, provides user access to sequential, index sequential and direct access files. The user process may invoke the services of FILES/11

directly through the use of system provided routines, or via the DECNET mechanism.

Because of the fact that FORTRAN was the highest level language with which the required DECNET user subroutines can interface, it was found necessary to write the DDACS in FORTRAN. However, a preprocessor was available on the system used for the implementation, which makes up for some of the deficiencies of FORTRAN. This preprocessor is called Structured Fortran (SF4) [67]. Hence, it was decided that the DDACS should be developed in Structured Fortran.

5.5 The Design of the DDACS

The access controller was developed as a sequential process. It can be broken into two distinct parts: the access control portion, which manipulates the data structures, as described in chapter 3, and the interface with the environment.

The interface performs all the communications with the user processes in the same station and with other access controllers in remote stations by means of DECNET V1.2. Message communication in DECNET V1.2 is accomplished by initalizing a communication link between the two processes which wish to communicate. This link is associated with an integer number, called a Logical Unit Number (LUN), where $1 \leq LUN \leq 255$. Then, any of the DECNET services can be invoked for that communication link by calling the

appropriate DECNET user subroutine and passing the logical unit number as parameter. It is necessary to state the maximum number of communication links which a process will use simultaneously, upon calling the DECNET initalization subroutine. This puts an upper limit on the total number of local processes and remote access controllers with which an access controller can communicate at any one time. A communication link between two processes may be broken by either process calling the appropriate DECNET subroutine, or by termination of either process.

The interface portion of the access controller consists of three routines which initalize the communication work areas, connect links to local processes and remote access controllers and receive and send messages on these links.

The access control portion of the DDACS consists of 30 routines. It performs the function of the access controller in a station when it manipulates the process descriptor data structures. It also performs the function of a LAC when it manipulates the data structures associated with one of its files: file descriptor, predecessor lists, immediate predecessor lists and immediate successor pointer. These data structures have been implemented as tables: a process descriptor table and a file descriptor table.

5.5.1 Data Structures in the DDACS

The process descriptor table includes the process

descriptors for those local processes which are currently in communicaton with the access controller. Figure 5.2 details an entry in the process descriptor table.

FIGURE 5.2. The Process Descriptor Table Entry

The fields of the process descriptor table entry are as follows:

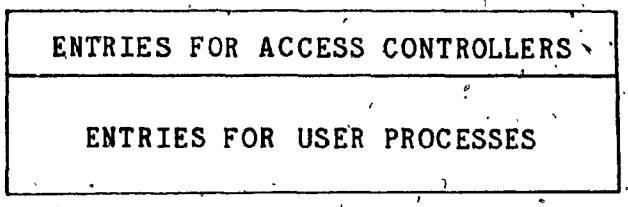
PNAME The process name
STTN The station in which the process runs
STATE The state of the processing of the file request or release from this process, (initial, suspended, resumed)
FILES The files currently used by the process
REQST The file requested by the process
REPLS The number of outstanding replies to predecessor list requests for the processing of this process's file request or release
PLIST Auxilary variable for forming the union of predecessor lists during the processing of file requests or releases from this process.
QUE The link for the file request queue

The table is organized so that the logical unit number of the communication link with a local process is the index of the associated process descriptor in the table. An entry in the table is initalized when the link with the process is established. The entry is deleted when the link is disconnected. An entry in the table is considered active if the process owns at least one file and/or, has a pending file access request. Otherwise, the entry is inactive although the link may not be disconnected. A section of the process descriptor table is set aside for information on the

communication links with other access controllers (Figure 5.3). Again, the entries are indexed by the logical unit number of the links.

To maintain a consistent approach to the processing of file requests from both local and remote process, the process descriptor table is also used to store process descriptors of remote processes which request or hold local files. Such an entry is created when a remote process requests a local file. The entry is updated for any further local file requests from that process, and is deleted when the process has released all the files local to the access controller. For remote processes the communication link number does not provide an index to the table, since all communication with a remote process is effected through its access controller. Entries for local and remote processes are stored in the same section of the table.

FIGURE 5.3. The Process Descriptor Table



The process descriptor table has an area reserved for holding information on communication links with remote access controllers. The remainder of the table contains process descriptors for user processes which own or request files in the station.

An entry in the File Descriptor Table (FDT) consists of a file descriptor, a predecessor list, an immediate predecessor list, an immediate successor pointer and a pointer to the head of the queue for the corresponding file. (Figure 5.3) There is an entry in the table for each of the local files of the access controller. The manipulation of an entry in this table may be regarded as the activity of the corresponding file's LAC. The list of file names in the FDT constitutes the access controller's local file directory.

FIGURE 5.4. The File Descriptor Table Entry

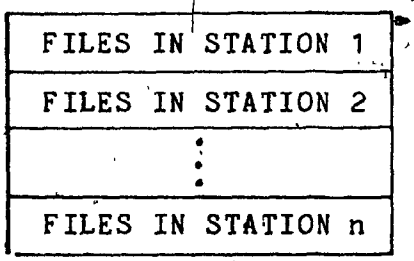
The fields of the file descriptor table entry are as follows:

FNAME The file name
MODE The current status of the file (free, allocated, deadlocked)
OWNER The process which currently owns the file
QUEUE The pointer to the head of the request queue for this file
PLIST The file's predecessor list in the wait graph
IPLIST The file's immediate predecessor list in the wait graph
ISUCC The file's immediate successor pointer in the wait graph

A global file directory is maintained by the access controller. There is an entry in this directory for each file in the network, which consists of the file name and the name of the station where the file is stored. Because the file population is static in both number and location, the global file directory can be ordered by station. All files in one station form a group of consecutive entries in the

global directory. The group of entries relating to one station is in the same order as the entries in the file descriptor table in the station's access controller. Then, the position of an entry for a particular file within its group is used as the index of that file's associated data structure in its access controller's file descriptor table.

FIGURE 5.5. The Global File Directory



The global file directory is comprised of sections. Each section contains the list of files stored at one of the network stations.

In communication between access controllers, files are referred to by their index in the global file directory. In communication between an access controller and a user process, files are refered to by their file names.

5.5.2 Message Handling in the DDACS

The access controller accepts messages from its interface one at a time. All processing which may be accomplished with the data immediately at hand, is performed before another message is accepted. This may involve the manipulation of more than one entry in either of the tables.

For example, a predecessor propagation travels to the end of a connected component of the W-graph and the predecessor list of each file on the portion of the component travelled by the propagation, is updated accordingly. When an access controller receives or initiates a propagation, it follows that portion of the connected component which is continuous and local to itself, updating all the appropriate predecessor lists, before performing any other action or accepting another message. Messages which are to be sent to a local process or remote access controller, are submitted to the Message Switching Mechanism.

5.5.3 Message Formats in the DDACS

For simplicity, the message formats in this implementation are all of the same length. The maximum length of the message format is decided by the length of the bit lists which form part of some messages. Since the bit lists indicate files, the length of the bit list depends on the number of entries in the global file directory. The message formats themselves are described in appendix A.

Whenever some processing of the access controller requires more than one piece of information of a particular type to be sent to a remote access controller, a bit list is formed of all the files in that remote station to which the information refers, and only one message is sent.

This occurs when an access controller queues a file

request from a process's which owns more than one remote file in the same remote station. Logically, the access controller would send a message to each of the LACs of the process's current files to update its file's immediate successor pointer. However in the DDACS the access controller sends one message to each remote station in which the process owns files. The message contains a bit list which indicates which files are to have their immediate successor pointers updated.

Similarily, when an access controller requires predecessor lists from a remote station, one message is sent to the remote access controller. The message contains a bit list which indicates which file's predecessor lists are required. The reply to a predecessor list request is one message which contains a predecessor list. This list is the union of all those predecessor lists which are requested.

5.5.4 Suspended Processing in the DDACS.

There are three actions of the access controller which cannot always be completed with the data immediately available. These actions all require predecessor lists. If the predecessor lists required are those of remote files, then the processing must be suspended until the predecessor lists can be obtained from the remote access controller. The three actions which require the predecessor lists are file release processing, the file request deadlock check and deadlock recovery processing. Because the access controller

is developed as a sequential process, the suspension of the processing of one of its logical LACs would mean the suspension of the access controller as a whole. To overcome this difficulty, the process descriptor table is extended to hold information on the processing of file requests or releases which must be suspended. This information, stored as part of the process descriptor for that process which made the file request or release, is the union of the predecessor lists received so far in response to requests, and the number of replies yet to be received (variable REPLS and PLIST, see Figure 5.2).

Similarily, a deadlock descriptor table is employed, in which all information on a suspended deadlock recovery can be stored. The number of suspended deadlock recoveries stored in this table, depends on the requirements of a particular system. Since deadlock occurs only rarely, this table requires little storage space. In the DDACS, up to three suspended deadlock recoveries are allowed by an access controller, at any one time.

5.5.5 Remote File Requests in the DDACS

In order to perform the deadlock check for a file request, the access controller requires the predecessor lists of all files owned by the requesting process. In the DDACS, when a process requests a remote file, the file request message which is sent to the remote access controller contains a list of the files owned by the

process. It also contains a predecessor list which is the union of the predecessor lists of all the files owned by the process which are local to that process's access controller. Thus, there is no need for messages concerning the files owned by the process. The number of predecessor list requests and replies is also reduced.

5.6 Testing the DDACS.

The DDACS was tested using two access controllers. Both access controllers were in fact run in the same station. However, since the DECNET Message Switching Mechanism allows communication links between processes in the same station, it was possible to simulate two remote stations with an access controller in each. Routines were developed which allowed the user processes to communicate with the access controller. The routines communicated with whichever of the two access controllers was in the same simulated "station" as the user process. Each access controller had sufficient space in their process descriptor table to allow up to five user processes to be in communication with it at the same time.

The distributed database consisted of ten files, five files under the juristiction of each of the access controllers. Actual file manipulation by the user processes was not implemented . The file manipulation was simulated by suspending a "user process" whenever it would have in reality been processing its files. The periods of suspension

were chosen according to which sequence of file requests was being tested.

The DDACS was implemented without error checking, error analysis or error recovery routines. The model which was implemented is that which performs only the access control algorithm using the wait graph structure to avoid deadlock in the majority of cases, and detect deadlocks whenever they occur. Consequently, testing the implemented model consisted of verifying the correct allocation of requested files to the user processes. A log file for each access controller was used to achieve this. All messages sent and received by the access controller, as well as significant events and the access controller's variables at the time of the events, were recorded on a log file. An example of this log file is shown in appendix B.

Appendix A contains the complete program listings of the DDACS. Appendix B contains the message formats. Appendix D contains the calling tree of the DDACS and appendix E contains examples of the log file.

CHAPTER 6

Error Handling in the Access Control Mechanism

We present below a discussion of possible error conditions in the distributed access control mechanism. With each type of error, we propose a method for reducing the occasions of the error, and for recovering from the adverse effects of such an error. We base our discussion on the model of the access control mechanism, described in chapter 5, which underlies the implementation of the Distributed Database Access Control System. However, this does not extend to a detailed discussion of specific errors in that particular implementation.

6.1 Errors due to Lost or Duplicated Messages

We define a lost message as a message which the sender places correctly on the message switching mechanism, and which is correctly addressed, but which is never received by the receiver to which it was addressed. The receiver is not aware of the fact that the message was sent to it.

A duplicated message is one which the sender process places only once on the message switching mechanism, but which is presented to the receiver process more than once, each time as a distinct message. The sender process is not aware of the fact that its message has been duplicated.

The most common effect of a lost message is the indefinite suspension of a user process. This is particularly true of messages which refer to file access requests: either the file access request itself or the reply to the request. A user process suspends itself when it makes a file access request, and remains suspended until a reply to the request is received from its acccess controller. If the request is lost, so that the access controller cannot send a reply, or if the reply is lost, then the process will remain suspended indefinitely. The loss of a reply which grants a file access request, not only leads to the indefinite suspension of the process, but also to the indefinite allocation of the granted file to that process.

The loss of a file release message leads to the indefinite allocation of the file to the process which issued the release message. However, when the process terminates, the access controller in the same station as the process, automatically releases any files which the process's descriptor indicates as still being allocated to that process. If a process releases a remote file and the release message from the access controller in the process's station to the remote access controller, is lost, then the file remains allocated to that process. The termination of the process has no effect on this, since the process's descriptor in the same station as the process, no longer shows that the file is allocated to that process. Then, the file is indefinitely allocated.

Lost or duplicated messages also affect the performance of the access controller if they result in wait graph variables which do not reflect the real relationships between processes and files in the network. The inaccurate wait graph variables may lead to faulty processing of file access requests, which may result in deadlock. The deadlock will be detected and broken in the usual way, provided that the inaccurate variables which caused it, are corrected by the normal processing of the access controller. This would be the case if a predecessor list propagation message was lost. The next propagation along the same path of the connected component would bring the predecessor lists up to date. However, the deadlock would not be detected and broken if some immediate sucessor pointer was incorrect, since this would cause the propagation to stop or be misdirected.

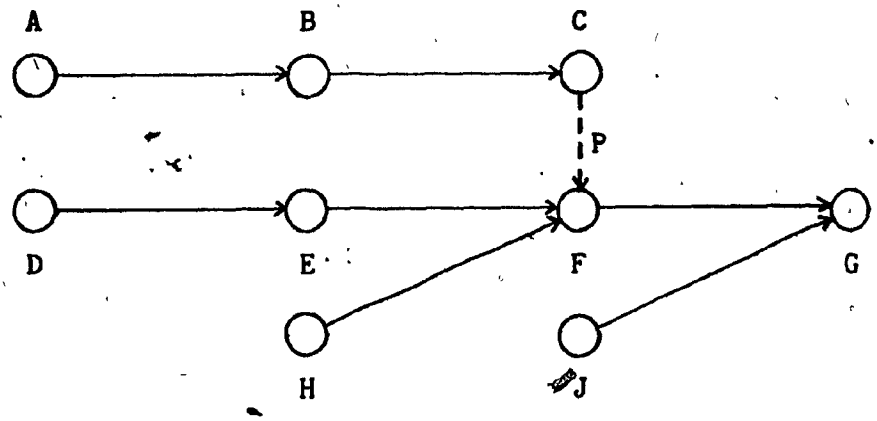
Indefinitely suspended processes, such as those in an undetectable deadlock or those which wait indefinitely because of lost access requests or replies, as well as indefinitely allocated files, lead to lock-up of parts of the distributed system. Files which are owned by indefinitely suspended processes are themselves indefinitely allocated. Any process which is waiting for an indefinitely allocated file, is then indefinitely suspended. Any file which such a process owns, becomes indefinitely allocated.

In terms of the W-graph, any connected component whose end node is an indefinitely allocated file, is a "dead" component. This means that it does not shrink since the end

node file cannot be released. All the files which are nodes in the dead component, are themsevles indefinitely allocated. The processes which own these files are indefinitely suspended.

If a process requests an indefinitely allocated file, and is allowed to wait for that file, the process becomes indefinitely suspended. Any file which that process owns, and all the files which are their predecessors in the W-graph, become indefinitely allocated. The processes which own these predecessor files are then indefinitely suspended (figure 6.1).

FIGURE 6.1. Indefinitely Suspended Processes and Allocated Files



The process which owns file G is indefinitely suspended. The connected component which has G as its end node will not shrink since the process which owns file G cannot release it. If process P is allowed to wait for file F, then it will wait indefinitely. The connected component (A,B,C) is joined to the "dead" component by the edge (C,F). All the files in this component are then indefinitely allocated.

The result, in terms of the W-graph, is a connected component which continues to grow with the addition of new sub-trees. A connected component will only shrink when the process which owns the file represented by the end node, releases that file and it is allocated to a waiting process. However, in this case the "end node" file is indefinitely allocated and, hence, the connected component does not shrink. The result is a connected component in the W-graph which will only grow.

6.2 Detection of Lost and Duplicated Messages

It has been noted in the literature that the design and implementation of resilient communication protocols is important in the prevention of faults in computer networks [48]. However, assuming that for a given Message Switching Mechanism there exists the possibility that a message will be lost or duplicated, we are interested in examining a mechanism which will detect this condition. We present below a scheme which is useful for detecting lost and duplicated messages, and which can be implemented as part of the communication routines of the access controllers and user processes. This scheme assumes that the message switching mechanism guarantees that all messages sent on a particular communication link, arrive in the order in which they are sent [64].

In each station of the network, the access controller maintains a logical communication link with every other

access controller and with all processes in its own station which own or have requested a file. The messages which are sent on a particular link can be uniquely identified by numbering them consecutively. Then, lost or duplicated messages can be detected, if the identification number of the last message received on a link is maintained for every logical communication link which the access controller has. This scheme can also be employed by a user process for detecting if messages sent to it by its access controller have been lost or duplicated.

Since the detection of a lost message on a logical communication link depends on the receipt of a subsequent message on the same link, the above scheme will not detect a file access request, from a user process to the access controller in the same station, which is lost. This is because a user process suspends itself once it makes a file access request, until it receives a reply to that request. Therefore, there are no subsequent messages on the logical communication link with the access controller, which would indicate that the request had been lost.

6.3 The Watchman Mechanism

Indefinitely suspended processes and indefinitely allocated files cannot be detected by any action of the access controller. It is desirable that a mechanism should be introduced into the system, which can detect such error conditions and recover from them. Described below is such a

mechanism, which we will call the "watchman".

The watchman mechanism contains two distinct parts. One part detects and resumes indefinitely suspended processes. This part we call the "process watchman". The other part, the "file watchman", detects and recovers indefinitely allocated files. Each station in the network has its own watchman mechanism which is concerned only with those processes and files which are local to its station.

6.3.1 The Process Watchman

The process watchman is a program which exists as part of the Message Switching Mechanism (MSM). It may be regarded as a routine which is invoked from within the MSM and which has access to the MSM service routines.

For the purpose of this study we postulate that the Message Switching Mechanism should have a facility which detects the absence of message traffic for a certain length of time on a particular communication link. This time-out facility is applied to the communication links between user processes and the access controller in the same station. In particular, it is applied when a user process suspends itself upon making an access request. Then, there is a "time-out" on this process's communication link, if the access controller does not send a reply to the access request within a specific length of time. This length of time is decided as part of the system tuning procedure and

depends on such variables as the requirements of the user processes and the system loading.

Whenever the MSM detects a time-out as described above, it invokes the process watchman, and supplies the name of the process whose link was timed-out. At this time one of three conditions exists with regard to the process's request: (1) it is currently being processed; (2) the request message was lost; (3) the reply message has not yet been received by the process and may be lost. The watchman must determine which of these conditions exist and correct it if the condition is erroneous.

The watchman learns from the process's descriptor, via the access controller, if the request is still pending. If so, then if the requested file is local, the request is currently being processed and no error exists. In this case the watchman resets the time-out for this communication link. However, in the case of a remote file, the watchman communicates with the file's access controller to determine the condition of the processing of the request. If the watchman learns that the request was not received, it sends a duplicate of the request message, via the MSM, to the remote access controller. This is facilitated by the fact that a message buffer is associated with each active local process. This buffer contains the last message related to that process, which was processed by the MSM. If, however, the remote access controller indicates that the request was granted, then the reply message has been lost. In this case,

the watchman repeats the reply message to the access controller in its station. If the request has been put on the file queue by the remote access controller, then the watchman takes no corrective action. An access controller keeps no record of requests which were rejected, and therefore cannot tell if it had already rejected some particular request. Then, if the remote access controller has already rejected the request and the reply was lost, the request will be repeated and processed as if it had not been received. When the watchman has finished its processing for a particular process it resets the time-out for that process's link.

In the case where the process descriptor in the same station as the process, shows that no request is pending for that process, then either the request message or the reply message was lost. However, the watchman cannot conclude which message was lost, since it does not know which file was requested. Then the watchman sends a message with an appropriate error code, via the MSM, to the process. On receipt of this message, the process reissues its file access request. If the request has already been granted, the access controller will reply that the file is already owned by the requesting process. Otherwise, the request is processed in the usual fashion.

6.3.2 The File Watchman

The file watchman is a program which requires close co-operation with both the Message Switching Mechanism and the File Access Mechanism (FAM). Thus, we postulate an interface for the File Access Mechanism, which exhibits the usual characteristics of the FAM interface, but with a communication link to the watchman. This interface also records the time and origin of all read/write requests which it receives. Further, this interface has a time-out facility which invokes the file watchman whenever an allocated local file has not been read or written to in longer than some specified time.

Whenever the file watchman is invoked, it determines whether the particular file, whose allocation it is examining, is indefinitely allocated. This is the case if the process has released the file, but some error has caused the file to remain allocated to the process. The file watchman learns from the FAM interface which process owns the file. The watchman then determines from the access controller in the station in which that process runs, the status of the process. If no process descriptor exists for that process or its descriptor does not show that file as owned by the process, then the file is indefinitely allocated. Then the file watchman sends a "file release" message to the file's access controller. The file is then free to be re-allocated. If, however, the process descriptor shows that the file is owned by that process, then no error

exists. In this case the watchman instructs the FAM interface to reset the time of last access to or last inspection of that file, to the current time.

The usefulness of the watchman mechanism can be measured with regard to a particular distributed system. The high cost in processing and network traffic overhead may not be justified by the service offered, especially if resilient protocols exist in the Message Switching Mechanism which minimise communication errors.

6.4 Corrupt Files

When the File Access Mechanism attemps to read or write portions of a file on behalf of a process, it may detect that the medium on which the file is stored is damaged so as to cause physical read or write errors. We describe a file in this condition as a "physically corrupt file". When a user process references a file, it may detect that the file contains invalid data which are inconsistent within the file itself or inconsistent with data from another source. We refer to a file in this condition as a "logically corrupt" file.

Described below is a framework for the recovery of files which have been corrupted. We assume the existence of a process, which we call the repairman, in each station of the network. A corrupt file is allocated to the repairman while it is being recovered. The mechanism of the repairman

is not of interest here. In fact, it may not always be possible to recover a file automatically, and therefore, the recovery may require some manual intervention by a human operator.

The repairman is allocated a file which needs to be recovered, and releases it when the recovery is complete. In this way, the data structures, W-graph and file queue which concern the corrupt file may be maintained in the usual fashion.

In the case of a physically corrupt file, the File Access Mechanism returns an error code to the user process when it detects the file to be corrupt. The process releases the file immediately by sending the access controller a release message which indicates that the file is corrupt. When the file's access controller receives the release message, it rejects the requests of all those processes which are currently waiting for the corrupt file. This effectively deletes the file's predecessor and immediate predecessor lists. The access controller then invokes the repairman and allocates the file to it. Any requests for the file which are received while it is allocated to the repairman, are also rejected. The rejection message indicates that the file is under repair. A process may wish, nevertheless, to wait for the file. If so, it then issues a special access request. The special access request is processed in exactly the same way as an ordinary access request for a file which is not currently free.

In the case where a file is logically corrupt, the process which detects this condition, reports it to the access controller. However, the condition may be detected in some manual processing of output data, rather than during the execution of a program. In this case a process may be run whose sole function is to report that the file is corrupt. In either case, the access controller rejects all pending requests for the file, invokes the repairman and allocates the file to it. New access requests for that file are handled as described above.

A process can detect that a file is corrupt only while it is accessing that file. This means that the file is either not in the W-graph, if no other process has requested it, or it is an end node in the W-graph. Thus, removing the file from the possession of that process and allocating it to the repairman does not involve W-graph manipulation, since the file has no successor in the graph.

However, as noted above, a corrupt file can be detected by processing other than that of the user process to which the file is allocated. In this case, the file may be a node in the W-graph which is not an end node, indicating that the process which owns the file is suspended. Then, the file is not removed from the possession of the process until that process has been resumed. Again, the allocation of the corrupt file to the "repairman" does not require W-graph manipulation, since it is not removed from the possession of the process until it becomes an end node.

The process which owns the file when its corrupt state is detected, may wish to have access to the file again as soon as it is recovered. When the process is informed or itself detects that the file is corrupt, it sends a special access request for that file along with the corrupt file release message, to the access controller. Then its request is placed at the head of the queue for that file, and the process is allocated the file as soon as it is recovered.

When the recovery of the corrupt file has been completed, the repairman releases the file. On receipt of the file release message, the access controller resumes normal functioning, with respect to that file, and allocates it according to the first entry on the queue. However, if the repairman cannot completely recover the file without some manual intervention, it informs the access controller of this fact. Because the time it takes a user or systems programmer to recover such a file is indeterminable, the access controller rejects all pending and future requests for that file with a suitable rejection message, including the special requests introduced above. Normal functioning for that file is resumed when the repairman informs the access controller that the file has been recovered.

6.5 Break-up of the Network

Faults may occur in the computer network which cause it to break into two or more distinct parts. In a geographically distributed network, faulty communications

equipment could isolate one or more subnetworks from the remainder of the network, effectively breaking the network into two or more subnetworks. Each subnetwork would then regard the rest of the network as being unavailable. A subnetwork could consist of only one station. Stations in the network could also become unavailable if the access controller in the station ceased to function correctly, or if there was a failure of some other vital component of the station.

Faults in communication equipment may occur which do not cause the computer network to break into two or more distinct parts. In such a case an underlying recovery mechanism can take advantage of the structure of the network to reroute messages which are normally routed through the equipment which is now faulty. Such cases are not of interest here. We consider only those cases which result in the break-up of the network.

When an access controller discovers that it can no longer communicate with another access controller in a remote station, it assumes that that access controller and all the files stored at that remote station are no longer available. It must then alter its data structures to reflect this fact. This involves informing any local process which owns or is waiting for one of the unavailable files, and reconstructing the local W-graph data structures which make reference to any of these files. Moreover, it must restructure the queues for its own local files so that it

does not allocate one of them to a process in the same station as an unavailable access controller.

When an access controller first detects a fault in the network, it checks the availability of the other access controllers to determine the extent of the failure. Then it broadcasts a "network recovery" message to the access controllers in the network stations which are still available. Every access controller which receives this broadcast message or detects the fault for itself, first broadcasts a "network recovery" message if it has not already done so, and performs the recovery routine described below. If the access controller discovers that some subnetwork of more than one station has become unavailable, the broadcast message and subsequent data structure recovery pertain to all the unavailable access controllers.

The access controller first flags, in its global file directory, all those files which have become unavailable to it. It then rejects any future requests for an unavailable file with an appropriate rejection message. Any process in the same station as the access controller which has a pending request for one of the unavailable files also has its request rejected. Any process which owns one of the unavailable remote files continues to use it and release it if informed by the File Access Mechanism that it can no longer gain access to that file.

The access controller reorders the queues for its local

files, so that all requests from processes in the remote stations where the access.controller is no longer available, are at the tail ends of the queues. These entries are flagged so that the file is not allocated to one of those remote entries as long as the corresponding access controller remains unavailable. Any new request which is put on a file queue is entered ahead of such flagged entries.

If a local file is allocated to a process in the same station as an unavailable access controller, it is removed from the possession of that process. This is done because the remote process cannot signal that it has released the file, as long as its own access controller remains unavailable.

To recover its W-graph variables, the access controller removes from the immediate predecessor lists and immediate successor pointers of its own local files, any reference to these unavailable files. It must also remove from each of its predecessor lists any sub-component of the W-graph which has one of the unavailable files as its root and is an immediate predecessor of one of its local files. This is because the removal of an immediate predecessor node breaks off that subcomponent of the W-graph of which that node was a root. Effectively, the edge emanating from this node, is removed from the W-graph by the error which caused the break-up of the network.

The removal of this edge from the W-graph is reflected

in the variable lists in a similar fashion to the deadlock recovery described in section 3.9. The access controller flags as invalid any predecessor list which contains one of the unavailable files. It then rebuilds its immediate predecessor lists by deleting from them any reference to the unavailable files. The access controller then rebuilds the predecessor list which is flagged as invalid, only when it requires it for file request or release processing or when it has been requested by some remote access controller.

The access controller rebuilds a predecessor list by first setting it to empty (null). It then enters all its immediate predecessors into that list. Subsequently, it forms the union of each of its immediate predecessors and auguments the list which is being rebuilt, with this union. However, if a predecessor list of an immediate predecessor is flagged as invalid, the access controller must rebuild this list before it can be used in the union. Hence, the rebuilding of the predecessor lists is a recursive process.

If the access controller requires a predecessor list for its recovery processing, which is in a remote station, it requests that list only if it has already received a broadcast "network recovery" message from the access controller in that remote station. Otherwise it continues the recovery without that list. Any deadlock which may occur as a result of this action will be detected and recovered in the usual fashion.

When a predecessor list has been rebuilt, the invalid flag is removed. A rebuilt predecessor list is propagated along the connected component of the W-graph, as in the deadlock recovery. If a predecessor list, which is flagged as invalid, is to be augmented by a propagated list, it is first set to null. It is then augmented with the propagated list. This effectively removes the unavailable files from the predecessor list, since any propagation will be either a result of the recovery described above or the result of introducing a new edge.

If the access controller suspends the processing of requests which would require propagation of an invalid predecessor list, then any new edge will have a valid propagated predecessor list associated with it. The flag is removed from a predecessor list which has been recovered or has received a propagation.

When this operation has been completed, the W-graph no longer contains any of the unavailable files. The access controller then rejects any pending requests from its local processes for any of these files. It also informs any local process which currently owns one of these files, that it is no longer available and updates the corresponding process descriptor to reflect the release of this file, or the rejection of the request. Any further requests for such unavailable files are rejected with an appropriate message.

If the network breaks into two or more subnetworks as a

result of a communications fault, each subnetwork detects that the remainder of the network is no longer available. When the above recovery routine has been completed, a subnetwork continues processing as usual, except that certain remote files are not available. When the fault has been corrected and the subnetworks reconnected, the access controller deletes the unavailable flags from the entries in the global file directory and file queues. Processing then continues as usual.

Any "file granted" message which is received by an access controller for a process which no longer waits for that remote file, is returned with a suitable error code to the remote access controller which issued it. The file is then free to be reallocated.

6.6 Corrupt Internal Variables in the Access Controller

The process descriptor, the file descriptor, the predecessor list, the immediate predecessor list and the immediate sucessor pointer are the variables maintained by the access controller. The redundancy of the information in the lists of these variables may be used to check them for validity. Described below is a scheme by which the access controller may check the validity of its variables and reconstruct them, should a valfdity check show them to be corrupt.

An access controller performs a validity check on its

variable lists if it is given some indication that these variables may be corrupt. If the access controller sends messages which contain corrupt data or sends messages to the wrong remote access controller, then the remote access controllers will return these messages with suitable error codes. When the access controller receives these erroneous messages back, and when it cannot find another obvious correction for them, it will perform the validity check on its variable lists.

This scheme is described in two parts: the first part checks the validity of both the process and file descriptors and recovers them if necessary; the second part is the validity check and recovery for those variables which describe the W-graph: the predecessor and immediate predecessor lists and immediate sucessor pointer.

Because of the assumed high locality of reference of the data base, the majority of the variables maintained by an access controller will refer to processes and files which are local to its own station. Then the access controller can check the validity of its own variables without requiring information from other access controllers.

The first level validity check by the access controller on its variables is a syntax check. An entry in one of the variable lists must conform to a particular syntax, otherwise the variable list is at least partially corrupt. For example, file names and process names, by convention,

should begin with a particular series of alphabetic characters. File names can be recovered from the global file directory if they are syntactically incorrect. There is, however, no such way of recovering the process names if they are incorrect. This problem can be overcome, if, among access controllers, processes are always referred to by the index of their process descriptor in the same station as the process.

The next level validity check compares the process and file descriptor tables. If a process descriptor indicates that the process owns a local file, then that file's descriptor should indicate that it is owned by that process. Otherwise, at least one of the variables is corrupt. The validity of the file descriptor can be checked further if the access controller determines from the File Access Mechanism, to which process the file is allocated. If the process descriptor shows that the process has requested a local file, then the corresponding file queue should contain an entry for that request.

If the file descriptor list proves to be corrupt, the access controller reconstructs it as follows: from the File Access Mechanism the access controller obtains information on which of its local files are free, which are allocated, and to what processes they are allocated. This information is sufficient to reconstruct the complete file descriptor list, except for the pointers to the head of the queues.

The file queue may be recovered from the entries in the process descriptor table which show that the process is waiting for a local file. If the process descriptor table is beleived to be corrupt, then the queues are reconstructed from information from the remote access controllers in which remote processes are waiting for local files. Local processes are resumed with a suitable message so that they re-issue their requests. The requests are then reprocessed.

The access controller reconstructs a process descriptor only when it requires that descriptor for some processing. This is done to avoid reconstructing a descriptor which is then deleted because of the termination of the process. To facilitate this method of reconstruction, the access controller flags its process descriptors as invalid, when it discovers the process descriptor table is corrupt.

When the access controller receives a file access request from a process whose descriptor is flagged as invalid, it checks the file descriptor list to determine which local files are owned by the process. It then determines which remote files are owned by the process by requesting the information from the remote access controllers. Once the access controller has determined which files the process owns, the request can be processed in the usual way. The flag is then removed from the process descriptor.

When the access controller receives a file release

message from a process whose descriptor is flagged as invalid, it processes the release in the usual way. As long as a process only releases a file which it owns, processing a release from such a process can have no adverse effect. In this case the flag is not removed, since the process may request some file in the future. The granting of a file request which had been pending is also processed in the usual way. Again, the flag is not removed. The flag is removed and the descriptor deleted when a process terminates.

There are relationships between the variables which describe the W-graph, which can be used by the access controller to check their validity. These relationships are as follows: for any file the immediate predecessor list must be a subset of the predecessor list. If the file's immediate sucessor pointer is not empty then the file should be a member of both the predecessor and immediate predecessor lists of its immediate sucessor. If these relationships do not exist between the variables which describe the W-graph, then at least one of the variables is corrupt.

If the access controller detects that its W-graph variables are corrupt, it reconstructs them as follows: it first flags all the W-graph variables as being invalid. New edges in the W-graph emanate from the end nodes of connected components, and the introduction of new edges may result in a loop in the graph. However, the access controller cannot use the invalid predecessor lists. AAAApAAAAAfor deadlock

checking and hence, allows new edges into the graph without the usual deadlock check. Any loop which may form as a result of this will be detected and broken in the usual way.

Predecessor list propagation follows the graph component described by the list of immediate sucessor pointers. If such a pointer is invalid, it must be reset before the propagation can continue. Whenever the access controller receives a predecessor list propagation or starts one itself it recreates any part of the W-graph variables which are flagged as invalid, as it processes the propagation.

To reconstruct an invalid predecessor list and immediate predecessor list, the access controller first sets them to null. The immediate predecessor is rebuilt by forming a list of files owned by the processes which have a request on the queue for the file whose W-grpah variables are being reconstructed. This information is obtained from the process descriptor table. The predecessor list is then rebuilt in the fashion described in section 6.5 above. However, in this case all remote access controllers are regarded as having correct predecessor lists and these predecessor lists are requested as required.

A file's immediate sucessor pointer is recreated by determining from the process descriptor table, which file was requested by the process which owns the file whose W-graph variables are being constructed. The reconstructed

predecessor list is then propagated to this immediate successor. The invalid flag is removed when a file's W-graph variables have been reconstructed.

The usual deadlock check is performed whenever a predecessor propagation is received. Hence, any deadlock which formed while deadlock checking was suspended, is detected and broken in the usual way.

CHAPTER 7

Comparison of Three Access Control Algorithms

In this chapter we compare three access control algorithms on the basis of the message traffic overhead which they incur, and their CPU time and storage requirements. The three access control algorithms which we compare are the deadlock detection and avoidance algorithm, described in chapter 3, the complete deadlock avoidance algorithm, described in chapter 4, and the distributed access control algorithm described by Mahmoud and Riordon in [39] and [40]. These algorithms are referred to as algorithm I, algorithm II and algorithm III respectively.

7.1 Outlines of the Algorithms

Below we present an outline of the combined deadlock detection and avoidance algorithm, described in chapter 3 above. We also present an outline of the Complete Deadlock Avoidance Algorithm described in chapter 4 above. These algorithms will be referred to as Algorithm I and Algorithm II respectively. It should be noted that the algorithms presented below are not descriptions of the processing of a program or programs. Rather, they describe the order of events in a distributed database system under particular circumstances. These outlines are presented in order to more conveniently derive a quantitative model of the algorithms.

In their discussion of access control, Mahmoud and Riordon [39,40] define two classes of processes:

Class 1: Processes which do not own files, whose request is for a single file and that file is not a multiple copy file.

Class 2: Processes which already own at least one file and/or whose access request is a multiple file request or a request for a multiple copy file.

Class 1 and 2 are mutually exclusive. In the case of algorithms I and II, multiple file requests and multiple copies of files do not occur.

The following notation is used in the outline of the algorithms:

AC - The access controller in the local station

RAC - The access controller of the remote station

SAC - If a class 2 process requests access to a file which cannot be granted immediately, then that file is a node in the wait graph. The node is then part of a connected component, which will have an end node. The access controller in the same station as the file which is that end node, is referred to as SAC.

PL - Predecessor List

IPL - Immediate Predecessor List

IS - Immediate Successor Pointer

P- - Prefix denoting P-graph structure e.g. P-PL is the P-graph predecessor list

7.1.1 Algorithm I (Deadlock Detection and Avoidance)

```
Local File Request:
Process sends file request to AC
if file is free then
    begin
    AC sends "file granted" message to process;
    AC updates process descriptor and file descriptor
    end
else begin
    AC obtains PL of files owned by process
    AC performs deadlock check;
    {this check is trivial if the process owns no files}
    if deadlock is false then
        begin
        AC updates IS of files owned by process;
        AC starts PL propagation;
        AC queues the request
        end
    else AC sends "request rejected" message to process
    end
```

```
Remote File Request:
Process sends file request to AC
AC obtains PL of local files owned by process
AC sends "file request" to RAC
{"file request" message includes PL of local files and list
of files owned by process}
if the file is free then
     begin
     RAC sends "file granted" message to AC;
     RAC updates process and file descriptors;
     AC sends "file granted" message to process;
     AC updates process descriptor
     end
else begin
     RAC obtains PL of files owned by process except files
     local to process;
     {PLs of files local to process have been sent in the
     "file request" message}
     RAC performs deadlock check;
     if deadlock is false then
          begin
          RAC updates IS of files owned by process;
          {files owned by process may be in any station of
          the network. Updating the IS of these files may
          require messages to remote stations.}
          RAC starts PL propagation;
          RAC queues the request
          end
     else begin
          RAC sends rejection message to AC;
          AC sends rejection message to process
          end
     end
```

```
Request granted after queuing:
{AC is in the same station as the file. Process may or may
not be in a remote station.}
{there may be edges incident on the requested file other
than those which are removed due to granting the request.}
if other class 2 processes are waiting for the requested
file then
     begin
     AC obtains PL of files owned by the process;
     AC deletes these Pls and files owned by process from PL
     of requested file;
     AC deletes files owned by process from IPL of requested
     file
     end
else AC sets PL and IPL of requested file to null;
AC sets IS of files owned by process to null;
AC updates file descriptor;
if process is in a remote station then
     begin
     AC sends "request granted" message to RAC;
     AC updates process and file descriptors;
     RAC sends "request granted" message to process;
     RAC updates process descriptor
     end
else begin
     AC sends "request granted" message to process;
     AC updates process and file descriptors
     end
```

7.1.2 Algorithm II (Complete Deadlock Avoidance)

```
Local File Request:
Process sends request to AC
if file is free then
    begin
    AC sends "file granted" message to process;
    AC updates process and file descriptors
    end
else begin
    AC shuts nodes of files owned by process;
    AC starts search for end node of connected component
    containing requested file;
    Search ends when an AC finds the end node of the
    connected component which contains the requested file
    {this AC is referred to as the SAC. AC may be the same
    access controller as SAC}
    while the end node is shut and request is not rejected
    do
        begin
        SAC obtains P-PL of files owned by process;
        SAC performs deadlock check
        if deadlock is true then
            begin
            SAC sends rejection message to AC;
            AC sends rejection message to process;
            AC reopens the shut nodes owned by process
            end
        else begin
            SAC updates P-IS of files owned by process;
            SAC starts P-PL propagation;
            SAC queues processing of request;
            SAC suspends the processing of the request;
            {SAC may process other requests while the
            processing of this request is suspended}
            SAC resumes the processing of the request
            when the node becomes open;
            {at this point the node which has become open
            will no longer be an end node, if the request
            which caused it to be shut has been placed
            and still remains on the file queue}
            if node is not an end node then
                begin
                SAC restarts search for end node;
                SAC finds end node of connected
                component;
                end
            end
        end
```

```
if end node was found to be open then
    begin
    SAC sends "end node open" message to AC;
    AC updates IS of files owned by process;
    AC reopens the shut nodes owned by process;
    AC queues the request;
    end
end
```

```
Remote File Request:
Process sends request to AC
AC sends "file request" message (including list of files
owned by process) to RAC
if file is free then
     begin
     RAC updates process and file descriptors;
     AC updates process descriptor;
     RAC sends "file granted " message to AC;
     AC sends "file granted" message to process
     end
else begin
     RAC shuts nodes of files owned by process;
     RAC starts search for end node of connected component
     containing requested file;
     Search ends when an AC finds the end node of the
     connected component which contains the requested file;
     {This SAC may be the same access controller as RAC}
     while the end node is shut and request is not rejected
     do
          begin
          SAC obtains P-PL of files owned by process;
          SAC performs deadlock check;
          if deadlock is true then
               begin
               SAC sends "request rejected" message to RAC;
               RAC sends rejection message to AC;
               AC sends rejection message to process;
               AC reopens the shut nodes owned by process
               end
          else begin
               SAC updates P-IS of files owned by process;
               SAC starts P-PL propagation;
               SAC queues processing of request;
               SAC suspends the processing of the request;
               {SAC may process other requests while the
               processing of this request is suspended}
               SAC resumes the processing of the request
               when the node becomes open;
               {at this point the node which has become open
               will no longer be an end node, if the request
               which caused it to be shut has been placed
               and still remains on the file queue}
               if node is not an end node then
                    begin
                    SAC restarts search for end node;
                    SAC finds end node of connected
                    component
                    end
               end
          end
```

```
if end node was found to be open then
    begin
    SAC sends "end node open" message to AC;
    AC updates IS of files owned by process;
    AC reopens the shut nodes owned by process;
    AC queues the request
    end
end
```

7.1.3 The Mahmoud and Riordon Distributed Access Control Algorithm

Below we present a description of the Distributed Access Control Algorithm described by Mahmoud and Riordon [39,40]. We refer to this algorithm as Algorithm III. The division of the requesting processes into class 1 and class 2 also applies in this algorithm. Mahmoud and Riordon include in their discussion, files with multiple copies in various stations of the network. Their algorithm also allows processes to request more than one file in a single request message.

The access controller, called the Distributed Database Management Facility (DDBMF), runs in each station of the network. A process sends its access request, which can be for one or more files, to the DDBMF in its own station. When a process makes a file access request, it enters a wait state until such time as all the files it has requested are available to it. A request for a remote file is sent by the local DDBMF, to the DDBMF in the remote station where the file is stored. If the file is free, it is granted immediately. When a DDBMF receives a request for a file which is not free, it takes into account the class of the requesting process. If the process is of class 1, the request is placed on the appropriate file queue. If the process belongs to class 2, the request is placed on a special queue called the pre-test queue.

The DDBMF acts on its pre-test queue only at specific times, which are separated by intervals of equal length. At the end of such an interval, a synchronized clock in each station of the network generates a signal for the DDBMF in that station. Then, every DDBMF broadcasts a status message concerning its own files and pre-test queue to each of the other DDBMFs in the network. Thus, if there are Nc DDBMFs, each one broadcasts and receives Nc-1 status messages. From these status messages, each DDBMF updates its global file queue information and constructs a global pre-test queue. This queue is then ordered according to some predefined static scheme, so that it is identical in every DDBMF.

The global pre-test queue contains all the requests issued in the previous time-interval which must be checked for deadlock. Each DDBMF in the network then performs the same deadlock test on all the entries in the global pre-test queues. The deadlock detection scheme uses the graph representation and deadlock detection algorithm of Murphy [50].

If the deadlock check shows that the request may be queued, it is placed on the appropriate file queue. If a request is rejected, however, the requesting process must release all the files which it currently owns and request them again in parallel with the new request. The DDBMF achieves this by pre-empting the process's current files and placing all its requests at the tail end of the appropriate queues. It then sends a message informing the process that

this has been done.

On receipt of such a rejection message the process leaves the wait state. If it does not wish to wait for all files to become free again, it informs the DDBMF of this. The DDBMF then removes all its requests from the queues. Otherwise, the process returns to the wait state and is resumed when all its requests can be satisfied.

7.2 Message Traffic Overhead in Algorithm I

Below we list the message traffic overhead incurred in the algorithm I for the various cases of the processing of file access requests. Each case is described and two formulae are given for the message traffic overhead which it incurs. The formulae express the message traffic overhead incurred for a local and remote file request respectively. The symbols used to express these formulae are as follows:

RR = RRt+RRy

RRt - The remote file request message between access controllers

RRy - reply to remote file request between access controllers

R = Rt+Ry

Rt - file request message to access controller from a process in the same station

Ry - The reply to file request, sent from the access controller to the process

Pr - The "predecessor list request" message and corresponding reply

Pl - The "predecessor list propagation" message

U - The "update immediate sucessor pointer" message

UPr - The "request to update immediate successor pointer to null and send predecessor list" message, and the corresponding reply.

B - The number of inter-station boundaries on the connected component of the wait graph between the requested file and the end node of that connected component

Kw = Kr+Kf+Kl

Kr - The number of remote stations in which files are owned by the requesting process.

Kl = 1 if the requesting process owns files in its own station, else Kl=0

Kf = -1 if the requesting process owns files in the same station as the requested file, else Kf=0.

7.2.1 Local and Remote File Requests

The message traffic overhead in algorithm I is as follows:

1. The file is found to be free and the request is granted, or the file is allocated to the "repairman" and the request is rejected:

   R

   R + RR

2. The request is rejected due to a potential deadlock:

   R + Kr*Pr

   R + RR + (Kr + Kf)*Pr

3. The request is queued after a deadlock check:

   Rt + Kr*Pr + Kr*U + B*Pl

   Rt + RRt + (Kr + Kf)*Pr + Kw*U + B*Pl

4. The request is granted after queuing; no other edges were incident on the requested file's node:

   R + Kr*Pr + Kr*U + B*Pl + Kr*U

   R + RR + (Kr +Kf)*Pr + Kw*U + B*Pl + Kw*U

5. The request is granted after queuing; or edges were incident on the the requested file's node:

   R + Kr*Pr + Kr*U + B*Pl + Kr*UPr (7.1)

   R + RR + (Kr + Kf)*Pr + Kw*U + B*Pl + Kw*UPr (7.2)

7.2.2 Deadlock Boss Selection and Recovery

Below we present the formulae for the message traffic incurred for the selection of a deadlock boss and the deadlock recovery in algorithm I. The symbols used to express the formulae are as follows:

Bs - The "boss selection" message

B - The number of inter-station boundaries on the loop in the wait graph.

N - The number of nodes in the wait graph loop

Ii - The number of remote stations containing the immediate predecessor files of node i in the loop

Ib - The number of remote stations which contain files owned by the process whose request is rejected in order to break the deadlock, but which do not contain any other file which is also an immediate predecessor of the boss file.

Rv - The "deadlock recovery" message

B - The number of inter-station boundaries on the loop in the wait graph

Pr - The "predecessor list request" message and the corresponding reply

The upper limit of message traffic for the selection of a deadlock boss, is incurred when each access controller which has a file in the deadlock, detects the deadlock and issues a boss selection message. Each message travels along the loop as far as the boss, where it is destroyed. In this case the overhead is

$$*Bs*B! \text{ messages}$$

When the boss has been selected, the recovery proceeds around the loop starting with the boss. The message traffic overhead for the deadlock recovery is then

$$Pr*(\sum_{i=1}^{N} Ii - Ib) + B*Rv$$

The total maximum message traffic overhead for deadlock boss selection and recovery is then given by

$$Bs*B! + Pr*(\sum_{i=1}^{N} Ii - Ib) + B*Rv \qquad (7.3)$$

7.3 The Message Traffic Overhead in Algorithm II

Below we list the message traffic overhead incurred in Algorithm II for the various cases of the processing of file access requests. Each case is described and two formulae are given for the message traffic overhead in that case. The formulae express the message traffic overhead incurred for a local and remote file request respectively. The symbols used to express these formulas are as follows:

R - as for Algorithm I

RR - as for Algorithm I

Pp - The request to shut nodes if they are not already shut, to set P-graph immediate successor pointers to null and to send P-graph predecessor lists, and the corresponding reply.

E - The "end node request" message

Ey - The "end node found" reply message

Pd - P-graph predecessor list propagation message

n - The number of times processing of a request is suspended

B - as for Algorithm I

Bp - The number of inter-station boundaries on the connected component of the P-graph between the immediate succesSon of the files owned by the requesting process and the end node of that component. Bp is the total number of boundaries for the whole of the n suspensions. If n= 0, then, Bp= 0.

U - as in Algorithm I

Up - The "update immediate successor pointer in the P-graph" message

Kw - as for Algorithm I

Kp = Kr + Ks + Kl

Kr - The number of remote stations in which files are owned by the requesting process.

Kl = 1 if the requesting process owns files in its own station, else Kl = 0.

Ks = -1 if the requesting process owns files in the same station as the end node which is shut, else Ks=0.

7.3.1 Local and Remote File Requests

The message traffic overhead in Algorithm II is as follows:

1. The file is found to be free and the request is granted:

   R

   R + RR

2. The request is rejected due to potential deadlock:

   R + Kp*Pp + B*E + Ey

   R + RR + Kp*Pp + B*E + Ey

3. The end node was found to be open and the request was queued:

   Rt + Kp*Pp + B*E + Ey + U*Kr

   Rt + RRt + Kp*Pp + B*E + Ey + U*Kw

4. The processing of the request is suspended after a deadlock check:

   Rt + Kp*Pp + Kp*Up + Bp*Pd + B*E

   Rt + RRt + Kp*Pp + Kp*Up + Bp*Pd + B*E

5. The processing of the request is suspended n times. In this case Bp describes the number of P-graph predecessor list propagation messages which must be sent between stations of the network, as a result of the processing of the request being suspended n times. Kp may not be the same for each of the n suspensions of the request. It will differ by at most 1, depending on whether the end node which is shut is not in a station which contains a file owned by the requesting process. B describes the number of "end node request" messages which are sent between stations in the network as a result of the search for an open end node:

   Rt + Kp*Pp + n[Kp*Pp + Kp*Up] + Bp*Pd + B*E

   Rt + RRt + Kp*Pp n[Kp*Pp + Kp*Up] + Bp*Pd + B*E

6. The request is rejected after n suspensions:

   R + Kp*Pp + n[Kp*Pp + Kp*Up] + Bp*Pd + B*E + Ey

   R + RR + Kp*Pp n[Kp*Pp + Kp*Up] + Bp*Pd + B*E + Ey

7. The request is queued after n suspensions of processing:

   Rt + Kp*Pp + n[Kp*Pp + Kp*Up] + Bp*Pd + B*E + Ey + U*Kr

   Rt + RRt + Kp*Pp + n[Kp*Pp + Kp*Up]
   + Bp*Pd + B*E + Ey + U*Kw

8. The request is granted after queuing:

$$R + Kp*Pp + n[Kp*Pp + Kp*Up] + Bp*Pd + B*E + Ey + U*Kr + U*Kr \quad (7.4)$$

$$R + RR + Kp*Pp + n[Kp*Pp + Kp*Up] + Bp*Pd + B*E + Ey + U*Kw + U*Kw \quad (7.5)$$

7.3.2 Deadlock Detection and Recovery

Since the selection of the deadlock boss and the deadlock recovery is identical for both the W-graph and the P-graph, the formulae for the message traffic overhead in Algorithm II for these cases are identical to those of Algorithm I. The equations are

Boss selection: $Bs*Bp!$

Deadlock recovery: $Pp*(\sum_{i=1}^{N} Ii - IB) + Bp*Rv$

It should be noted, however, that the values of N and Bp for the P-graph deadlock boss selection and recovery, are usually lower, but never greater than N and B in the equivalant W-graph deadlock boss selection recovery. This is discussed further in section 7.5 (see also figure 7.1).

7.4 The Message Traffic Overhead in Algorithm III

The message traffic overhead for local and remote file requests in Algorithm III is as follows:

1. The request is granted:

   R

   R + RR

2. The request is rejected:

   R + RFP

   R + RR + RFP

where

R - as in Algorithm I

RR - as in Algorithm I

RFP - The message sent to the requesting process to inform it that its request has been rejected and all its files pre-empted; requests for its required files have been placed on the tail-end of the appropriate file queues.

At the end of a time-interval each access controller broadcasts status messages concerning its file queue and pre-test queue. The message traffic overhead for this broadcast is

$$Nc*(Nc-1)*S \qquad (7.6)$$

where

Nc - The number of stations in the network

S - The file queue and pre-test queue status message.

The status message accounts for the majority of message traffic overhead in the network. The traffic overhead incurred by these status messages is not directly related to the request message traffic. Rather, it is a general

overhead which occurs in the network once every time interval.

7.5 The Comparison of Message Traffic Overhead

In all three algorithms the division of the processes into two classes is applicable. However, in algorithms I and II, class 1 contains only those processes which do not already own one or more files. Multiple copy files and multiple file requests are not permitted in these algorithms. In algorithms I and II, class 2 contains those processes which own at least one file when they make a file access request. As in Algorithm III, only requests from class 2 processes are checked for deadlock.

7.5.1 Algorithms I and II

The comparison of the message traffic overhead in the two algorithms is of interest only for class 2 processes. For class 1 processes, the message traffic overhead for local and remote access requests is R and R + RR respectively, in both algorithms.

The formulae for the message traffic overhead for a request are

$$R + Kr*Pr + Kr*U + B*Pl + Kr*Upr$$

and

$$R + Kp*Pp + n[Kp*Pp + Kp*Up] + Bp*Pd + B*E + Ey + U*Kr + U*Kr$$

for algorithms I and II respectively. It is assumed that the request is granted after it was queued; that deadlock does not occur. In both algorithms, this causes the highest possible message traffic, apart from the case in which deadlock occurs.

We compare the message traffic overheads in terms of number of messages. To this end, we need not concern ourselves with the particular nature of the different types of messages. We are only interested in the number of messages sent. We interpret the symbols introduced in sections 7.2 and 7.3 as representing the number of messages required for a particular purpose. The values for these symbols are derived from our implementation.

Formulae 7.1 and 7.3 describe the message traffic overhead which is incurred in algorithms I and II respectively, when a local file request is queued and then granted. We compare these formulae by using

E = Pl = 1, Prp = Upr = 2 and U = Up = 1.

Then, the difference between the formulae 7.1 and 7.3 is

$$Bp*Pd + (n-1)*Kp*Pp + (n+1)*U*Kp + Ey \qquad (7.5)$$

in the case where Kp = Kr. We use this simplification because Kp differs from Kr by at most 1, and only in the case where Kl = 1 and Ks = 0, or where Kl = 0 and Ks = -1.

Formulae 7.2 and 7.4 describe the message traffic overhead which is incurred in algorithms I and II respectively, when a remote file request is queued and then

granted. Again, we use the number of actual messages to compare the formulae. In doing so, we assume Kp = Kr+Kf. This is justified since Kp differs from Kr+Kf by at most 1, and only when Kf is not equal to Ks+Kl. Then, the difference between formulae 7.2 and 7.4 is

Bp*Pd+(n-1)*Kp*Pp+(n+1)*U*Kp+Ey

This is identical to formulae 7.5 for the difference between the maximum message traffic overhead for local requests. Formula 7.5, then, expresses the difference between the maximum message traffic overheads which is incurred by a file request in algorithms I and II respectively.

From our implementation we conclude,

Pd = 1, Pp = 2, U = 1, Ey = 1

Substituting these values into formula 7.5, we obtain

Bp+(3n-1)*Kp+1 (7.8)

This formula depends on n, the number of times the processing of the request is suspended in algorithm II, and Bp, the number of interstation boundaries on the P-graph component, for a given value of Kp.

for n = 0, Bp = 0, the difference is -Kp +1

for n = 1, the difference is 2Kp+1+Bp

for n = 2, the difference is 5Kp+1+Bp

The only case where a file request incurs less message traffic overhead in algorithm II than in algorithm I, is when n = 0, Bp = 0 and Kp > 1.

Where n = 0 and Bp = 0, the processing of the request was not suspended in algorithm II. Kp reflects the locality

of reference of the database. As Kp increases the locality of reference decreases.

In summary, the message traffic overhead for a file request in algorithm I, depends on the locality of reference of the database. In algorithm II, the message traffic overhead for a file request depends on the locality of reference of the database, and on the number of times the processing of a request is suspended. When the processing of a request in algorithm II is not suspended, the message traffic overhead incurred is less than that incurred in algorithm I, when $Kp > 1$. However, the message traffic overhead incurred in algorithm II increases with the number of times the processing of a request is suspended.

The number of times the processing of a request is suspended depends on the rate of file access requests and the average service time of the requests. As the average service time decreases, for a given rate of requests, the probability that the processing of a request will be suspended, also decreases. The average time decreases as the locality of reference increases. Thus, for a database with high locality of reference, we can expect that the processing of a request in algorithm II will not be suspended.

7.5.2 Deadlock Boss Selection and Recovery

In our comparison of message traffic overhead above, we

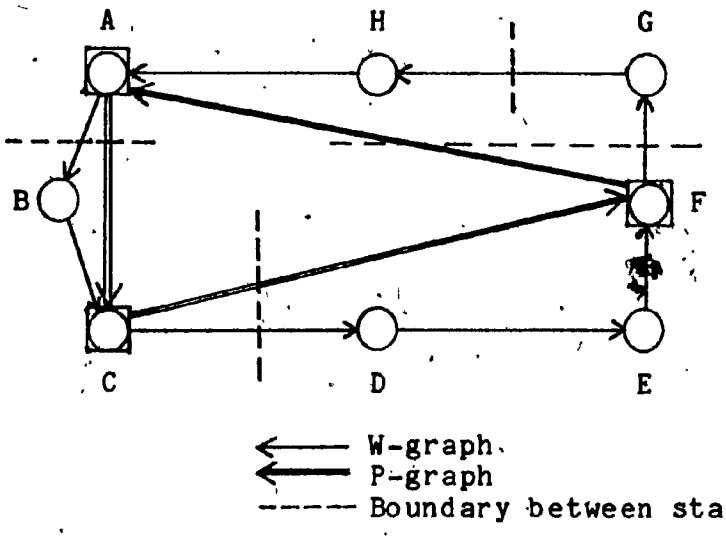
assumed that deadlock does not occur. However, for a particular set of network attributes, Mahmoud and Riordon [40] observed a number of deadlocks ranging from 12 to 105 in a simulated 6 hour session, depending on the precentage of class 2 processes which make additional file requests. Then, we must add the message traffic overhead for deadlock boss selection and recovery to the overhead for algorithm I and II.

The maximum message traffic overhead for boss selection and deadlock recovery is described by formulae 7.3

$$Bs*B! + Pr*(\sum_{I=1}^{N} Ii-Ib) + B*Rv$$

Although the expression for the deadlock boss selection and recovery message traffic overhead is the same for both algorithms I and II, the value of N may be different for the same case in the different algorithms. This is because in algorithm I, the deadlock is described by a loop in the W-graph; in algorithm II the loop is in the P-graph. A node in the P-graph corresponds to one or more nodes in the W-graph (see figure 7.1).

FIGURE 7.1. Deadlock in both the P-graph and the W-graph



The loop in the W-graph, (A,B,C,D,E,F,G,H,), was formed by the "simultaneous" introduction of the edges (A,B), (C,D) and (F,G). The P-graph for the same case is described by the loop (A,C,F). In this example in algorithm I, N = 8, B = 4; in algorithm II, N = 3, Bp = 3. If Ii = 1, Ib = 0 for algorithms I and II then the overhead is

24*8 + 4 = 196 messages for algorithm I

6*3 + 3 = 21 messages for algorithm II

When $\overline{Ii}$ and B are equal in both cases, although this need not be so, the message traffic overhead depends on the number of nodes in the loop. For the same case, the number of nodes in the P-graph loop is less than or equal to the number of nodes in the W-graph loop. This means that when B and $\overline{Ii}$ are equal for both algorithms, the message traffic overhead for deadlock boss selection and recovery in algorithm II is less than or equal to that of algorithm I in

the same case.

B may be less in algorithm II than in algorithm I, but may not be greater, since all nodes in the P-graph are also in the W-graph. Then the W-graph has at least the same number of interstation boundaries, B, as the P-graph.

It is not necessary that all Ii be equal in both algorithms for a given deadlock. However, for a given locality of reference, Ii should be equal for algorithms I and II. The different algorithms do not affect the distribution of file ownership in the network.

In summary, the message traffic overhead incurred for a deadlock boss selection and recovery in the P-graph of algorithm II is lower than in the W-graph of algorithm I. This can be seen in the example discussed in figure 7.1. Then, we conclude, that for a database with high locality of reference, the total message traffic overhead is lower in algorithm II than in algorithm I.

7.5.3 Algorithm III

In algorithm III the message traffic overhead depends not only on the file requests which are made, but also on the status broadcasts at the end of each time interval. The message traffic overhead due to status messages is described by

$$Nc*(Nc-1)*S$$

S depends on the number of files at a station, the length of

the queue for each file and the length of the pre-test queue. Let us assume that the status of each file can be included in one message Sf, regardless of the length of the queue. Similarly, we assume that the pre-test queue requires only one message Sp. Then the status broadcast message traffic overhead is

$$Nc*(Nc-1)*(Sf*F+Sp) \qquad (7.9)$$

where F is the average number of files per station.
Mahmoud and Riordon [40] use in their simulation, Nc = 16 with 128 files in the network, i.e., F = 8. Using these figures, and assuming that Sp = Sf = 1, we obtain for the message traffic overhead

Nc*(Nc-1)*(Sf*F+Sp)=

1920 messages per time interval

7.5.4 Comparison of Algorithms I and III

Using the simulation parameters of [40], we compare the message traffic overhead in algorithm III with several cases for algorithm I. In each case for algorithm I we use formula 7.1, the maximum traffic overhead for a file request, and formula 7.3, the maximum message traffic overhead for deadlock boss selection and recovery. We use formula 7.9 for the message traffic overhead in algorithm III.
In the comparison we use the following values:
Nc = 16, number files in the network = 128, thus F = 8.
R = 2, RR = 2.
Rate of requests = 20 requests/minute

Synchronized time interval = 1 minute

We vary the following parameters:

E = percentage of all requests which are for local files

D = the number of observed deadlocks in 6 hours

Below we describe the cases which we use in the comparison

Case 1: Kw = 0, B = 0, Ii = 0, Ib = 0, E = 90%, D = 12

Case 2: Kw = 2, B = 2, Ii = 1, Ib = 0, N = 4, E = 80%, D = 23

Case 3: Kw = 4, B = 3, Ii = 2, Ib = 1, N = 7, E = 70%, D = 39

Case 4: Kw = 6, B = 4, Ii = 4, Ib = 2, N = 10, E = 60%, D = 62

Case 1 reflects high locality of reference, since Kw = 0, Ii = 0, and E = 90%

FIGURE 7.2. Message Traffic Overhead in Algorithms I and III

Case	1	2	3	4
File request I	44	256	488	684
Deadlock I	0	0.78	3.79	18.36
Total I	44	256.784	491.794	702.36
Algorithm III	1964	1968	1972	1976

The message traffic overhead per time interval is computed for the four cases described above using formula 7.1 and 7.2 for the file request overhead and formula 7.6 for the deadlock overhead, in algorithm I and formula 7.7 for the status message overhead in algorithm III, assuming a request rate of 20 requests per minute

Using kl = 1, i.e., Kw-1 = Kr+Kf, and Bs = 1, Pr = 2, Rv = 1, U = 1, PL = 1, formula 7.1 becomes

$$5Kw+B-2$$

Formula 7.3 becomes

$$B!+2*(N*Ii-Ib)+B$$

The message traffic overhead was calculated for file requests and deadlock boss selection and recovery for algorithm I, in the four cases described above. The results are tabulated in figure 7.2. As can be seen from that figure, the message traffic overhead for algorithm III greatly exceeds that of algorithm I, for the cases described.

In the computation of the message traffic overhead in figure 7.2, we used that fact that in simulation, Mahmoud

and Riordon observed a number of deadlocks ranging from 12 to 62, depending on the ratio of local to remote file requests. However, it should be noted that in algorithm I, many of these deadlocks would be avoided. Hence, the message traffic overhead for deadlock detection is, in reality, lower than is tabulated in figure 7.2. To compute this number of deadlocks which would not indeed be avoided for a particular locality of reference, would require a detailed simulation model of the DDACS, which is beyond the scope of this work.

It should also be noted that figure 7.1 assumes that all requests in algorithm I are first queued and then granted. Again, this would not be the case in reality. Then, the message traffic overhead would be lower than is shown in the figure.

7.6 Comparison of CPU Requirements

7.6.1 Algorithms I and II

In algorithms I and II the processing of requests from class 1 processes is identical. Therefore, the CPU requirements of both algorithms for the processing of these requests are also identical. This is also true for requests from class 2 processes which can be granted immediately. It is in the processing of requests which must be checked for deadlock that the algorithms diverge. In algorithm II the amount of processing for a request depends on the number of

shut nodes which are encountered before the request is queued or rejected, and on the number of remote stations in which the requesting process owns files. In algorithm I the amount of processing also depends on the number of remote stations in which the requesting process owns files.

For the processing of the same request, algorithms I and II require identical CPU time up to the point where it has been decided that the request is to be checked for deadlock. The requirement in both algorithms for "walking" the connected component of the wait graph until the end node is reached, is comparable. In algorithm II, the "walking" is done before the request is queued. However, in algorithm I, the predecessor lists of the node, which would be visited in algorithm II's walk, are updated after the request has been queued. In algorithm I, the updating of the predecessor lists along the connected component amounts to an OR instruction at each node of the graph visited.

Establishing an edge in the wait graph demands identical routines in both algorithms. In the case where $n = 0$ for algorithm II, i.e., the processing of the request is not suspended, no more graph manipulation is required when the request is queued. However, in Algorithm I, the edge or edges must be correctly removed from the W-graph when the request is granted. This involves requesting remote file predecessor lists and updating the predecessor list of the requested file if there are other class 2 processes waiting for that file. The updating of immediate successor

pointers is identical in both algorithms.

This shows that in the case where the request is queued and then granted, algorithm I requires the processing of predecessor list requests and replies for every remote station in which the requesting process owns files, in excess of the processing required by algorithm II when n = 0 for the same case.

In the case where n = 1, algorithm II requests and processes P-graph predecessor lists from the remote stations. Since the P-graph manipulation routines in algorithm II are identical to the W-graph manipulation routines in algorithm I, the CPU requirements of both algorithms are equal when n=1 and the request is queued before it is granted in algorithm I. However, as n increases, algorithm II requires P-graph predecessor list manipulation in excess of the processing required by algorithm I for the same case.

In algorithm II, the probability that the processing of a request is suspended depends on the arrival rate of requests and the average service time of the requests. As noted in section 7.5.1, a database with high locality of reference, the probability that the processing of a request will not be suspended, is high. This means that in most cases where the request is queued before it is granted, we can expect Algorithm II to require less CPU time for request processing than algorithm I.

The deadlock boss selection and recovery routines in both algorithms are identical. As shown in the comparison of the message traffic overhead, the number of nodes in a P-graph loop is less than or equal to the number of nodes in the corresponding W-graph loop. The CPU requirements of algorithm II will not be greater than that of algorithm I for the same deadlock. The difference in their requirements will depend on the difference in the number of nodes in their respective loops. Algorithm I will require the execution of |N1|-|N2| boss selection routines and N1-N2 deadlock recovery routines more than algorithm II, where Ni is the number of nodes in the loop of algorithm i. This assumes that the maximum message traffic overhead, as described by formula 7.3, is required.

7.6.2 Algorithms I and III

A precise quantitive analysis of the CPU requirements of algorithm III is not possible, since no implementation details are available to us. However, using the outline of the algorithm presented in section 7.1.3, we may attempt a rough analysis and comparison with algorithm I.

In algorithms I and III, the CPU requirements are equal for requests from class 1 processes and requests from class 2 processes which can be granted immediately, since the algorithms are identical for the processing of these types of requests. Again it is in the deadlock check processing that the algorithms differ.

In algorithm I, the access controller where the requested file is stored is the only one which checks the request for deadlock. Because the wait graph information is stored as bit lists, once the necessary lists have been collected, the deadlock test consists of an AND operation. In algorithm III, all access controllers (DDBMFs) apply the same deadlock detection algorithm [49] to all pending requests.

A similar observation may be made regarding the maintenance of file status information. In algorithm III, every access controller maintains global file queue and pre-test queue information. In algorithm I, only information on those files which precede a given file in the wait graph is maintained by an access controller. Further, this information is only maintained for those files which are local to the access controller. In the case where a deadlock does occur in algorithm I, only those access controllers which have files in the deadlock are involved in the recovery. The recovery itself again consists of the manipulation of bit lists.

From this comparison we can conclude that the CPU requirements of Algorithm III exceed those of algorithm I. This is attributed to the simultaneous execution of similar algorithms by each of the access controllers in algorithm III.

In algorithm I, the CPU requirements decrease as the

locality of reference of the database increases. This is due to the fact that the deadlock check algorithm is faster, the fewer the access controllers that must contribute some predecessor list to it. High locality of reference also implies that in algorithm I, there is a low probability that deadlock will occur. However, in algorithm III, the CPU requirements for the maintenance of the global status information and the execution of the deadlock check algorithm are unaffected by the locality of reference of the database.

7.7 Comparison of Storage Requirements

7.7.1 Algorithms I and II

In both algorithms I and II, the storage requirements depend on the maximum number of files in the network. This determines the size of the local and global file directories, and the length of the bit list in the process and file descriptor. In algorithm I a bit list is used to describe predecessor lists, immediate predecessor lists and the files owned by a process. In a 16 bit per word architecture 128 files would require 8 words of storage for each of these variables.

Algorithms I and II have some storage requirements in common: the file directory and the deadlock descriptor table. The process descriptor table (PDT) in both algorithms differs only in that algorithm II requires one more variable

than algorithm I to describe the queue of requests whose processing has been suspended at a particular shut node. An extra variable in the file descriptor table of algorithm II is also required to point to the head of this queue. The amount of storage required for the W-graph in algorithm I is identical to that required for the P-graph in algorithm II. However, an extra variable in algorithm II is required for the immediate successor pointer of the wait graph.

In summary, algorithm II requires one more variable than algorithm I, for each file in the network, since there is a file descriptor table entry for each file. However, in both algorithms, there are more process descriptor table entries than there are active processes. This is because a process has an entry in the PDT of each station in which it owns or waits for a file. Then algorithm II requires two more variables per active PDT entry than algorithm I.

In our implementation, algorithm I requires 22 words of storage per process descriptor table entry and 22 words of storage per file descriptor table entry (local file), if we assume there are 128 files in the network. Then algorithm II requires 23 words of storage per process descriptor table entry and 24 words of storage per file descriptor table entry.

We conclude that algorithm II's storage requirement is greater than that of algorithm I. However, as noted above, the extra variables required are pointers which do not

depend on the number of files. Therefore, the precentage difference decreases as the number of files in the network increases.

7.7.2 Algorithms I and III

In algorithm III, each access controller stores a global file directory. This is comparable in storage requirements to the global file directory in algorithm I. It also requires a process descriptor, as in algorithm I, but does not require auxiliary variables for storing predecessor lists in connection with file request and release processing. Algorithm III does not use the W-graph structure. However, the deadlock detection algorithm which is employed, requires that each access controller have at its disposal the queues for all files in the network, and a global pre-test queue.

In comparing the storage requirements of algorithms I and III, omitting those elements which are common to both, we compare the storage allocated to the auxiliary variables in the process descriptor table and the wait graph variables in algorithm I, with the storage allocated to global file queues and pre-test queues in algorithm III.

Using Mahmoud and Riordon's simulation parameters of 16 stations, 4000 users, 128 files we obtain for algorithm I

Auxiliary variables = 10 words;

Wait graph variables = 8+8+1 = 17 words.

If we say that each station may have up to 50 concurrent users then the storage requirement for algorithm I is

50*16*10 + 17*128

= 128*(17+16.25)

= 128*78.25 words

For algorith III, the storage requirement is

(average queue length) *16*128

+ (average pre-test queue length) *16*16

If we assume

Q = average queue length = average pre-test queue length,

this becomes

storage = Q*16*128 + Q*16*16

= Q*16*16*9

Since the expected length of a queue in the simulation is not available to us, we cannot compute the amount of storage required for these queues. Therefore, we compute the queue length needed in algorithm III, in order for it to have the same storage requirement as algorithm I. Then, we compute what average queue length could reasonably be expected in Mahmoud and Riordon's simulation, from the simulation parameters in [40].

For equal storage in algorithms I and III

Q = 4.4 words

If a queue entry requires 1 word, this indicates an average queue length of 4.4 entries.

Na = 20 requests/min is the average file request rate from Mahmoud and Riordon's simulation parameters. Assuming that

the distribution of the requests is uniform over the 128 files in the network, then the average request rate per file is given by

$$Nf = 20/128 \text{ requests/minute}$$

From Kleinrock [35], the average queue length is given by

$$\bar{q} = \rho / (1 - \rho)$$

where ρ = (average arrival rate) * (average service time)

$$= \lambda * \bar{X}$$

In this case $\lambda = 20/128$ and $\bar{X} = 5$ minutes per request [40]

Then $\bar{q} = (100/128)/(1-100/128) = 3.57$ requests

This is of the same order of magnitude as the 4.4 requests computed above for equality of storage requirements for algorithms I and III. From this we can conclude that the storage requirements of the two algorithms are within the same order of magnitude for the simulation parameters given in [40].

The average queue length and as a result, the storage requirement of algorithm III, depends on the rate of file requests. If we increase Nf to 25/128 requests/min, then $\bar{q} = 42$ requests, more than 10 times the requirements for the parameters given in [40]. The storage requirement for algorithm I, however, does not depend on the request rate.

CHAPTER 8

Conclusions and Directions of Further Work

8.1 Conclusions

We have presented in this thesis, a new access control mechanism for distributed databases. We have shown in this mechanism, that access control for distributed databases can itself be distributed to the stations of the computer network, without the maintenance of global file information at each station.

The proposed mechanism has been fully implemented for a network of PDP/11 minicomputers. This implementation shows that the conceptual tool called the local access controller (LAC), which was used in the design of our access control method, may be incorporated in an implementation without increasing the message traffic overhead in the mechanism.

In the discussion of error detection and recovery mechanisms for our access control method, we have proposed solutions which would substantially improve the reliability of our system. On the other hand, these mechanisms would incur a considerable overhead, and would in part require modifications to the underlying Message Switching Mechanism.

We have developed a variation of our access control mechanism which avoids deadlock in the database in all

cases. While this variation requires more storage, it reduces the CPU time requirement and message traffic overhead for a database with high locality of reference.

We have compared our access control mechanism with the distributed access control mechanism of Mahmoud and Riordon [39,40]. While noting that their mechanism is more general, we found that for those cases where our restrictions are applicable, our mechanism is superior, particularly with regard to message traffic overhead.

8.2 Further Work

The primary aim of further work is to remove the restrictions of our access control mechanism. The ability to handle multiple file requests and to provide for shared access to files, as well as the introduction of multiple copy files, would greatly increase the applicability of our mechanism.

The introduction of multiple file requests and shared access to files means that a node in the wait graph may have more than one emanating edge. In the latter case, all the emanating edges need not belong to the same process. Avoidance and detection of deadlock using such a graph model would not be a simple variation of our mechanism.

It remains to be seen if our model can be generalised in the way described above, or if an altogether different approach might be appropriate. The basic idea, however,

remains valid, viz., that in a distributed system, a waiting relationship should only be represented in a particular station if it affects the allocation of objects residing in that station.

BIBLIOGRAPHY

1. Abrial J.R., Cahen J.P., Favre J.C., Portal D., Mazare G., Morin R. "Project SOCRATE - nouvelles specifications (version 3)", IMAG Universite de Grenoble, Sept. 1972
2. Asohim F. "Data base networks - an overview", Management Informatics, vol.3, no.1., 1974, pp.13-28
3. Bachman C.W. "Trends in database management 1975", Procs AFIPS NCC, 1975, pp.569-576
4. Bernstein P.A., Goodman N., Rothnie J.B., Papadimitriou C.A. "The concurrency control mechanism of SDD1: A system for distributed databases", IEEE Trans. on Software Engineering, vol.SE-4, no.3, May 1978, pp.154-168
5. Booth G.M. "Distributed information systems", Procs. AFIPS NCC, vol.45, 1976, pp.789-794
6. Booth G.M. "The use of distributed data bases in information networks", Procs. First Computer Communications Conference, Washington, D.C., Oct.1972, pp.371-376
7. Bucci G., Golinelli S. "A distributed strategy for resource allocation in information networks", Procs. International Computing Symposium, Belgium, April 1977, pp.345-356
8. Casey R.G. "Allocation of copies of a file in an information network", Procs. AFIPS SJCC, May 1972, pp.617-625

9. Chamberlin D.D., Boyce R.F., Traiger I.L. "A deadlock-free scheme for resource locking in a database environment", Procs. IFIP, Congress, 1974, pp.340-343

10. Chang E. "A distributed medical data base", Computer Networks, vol.1, no.1, June 1976, pp.32-38

11. Chang E., Linders J. "A distributed medical data base", Methods of Information in Medicine, Oct. 1974, pp.221-225

12. Chu W.W. "Optimal file allocation in a multiple computer system", IEEE Trans. on Computers, vol.c-18, no.10, Oct.1969, pp.885-889

13. Chu W.W. "Performance of file directory systems for data bases in star and distributed networks", Procs. AFIPS NCC, vol.45, 1976, pp.577-587

14. Chu W.W., Ohlmacher G. "Avoiding deadlock in distributed data bases", Procs. ACM National Symposium, vol.1, Nov.1974, pp.156-160

15. Chupin J.C. "Control concepts of a logical network machine for data banks", Procs. IFIP, 1974, pp.291-295

16. Chupin J.C., Seguin J. "A network direct access method", Procs. European Workshop on Distributed Computer Systems, Oct.1974

17. Coffman E.G. Jr., Elphick M.J., Shoshani A. "System dead-locks", Computing Surveys, vol.3, no.2, pp.67-68, June 1971

18. Collmeyer A.J. "Database management in a multi-access environment", Computer, vol.4, no.6, pp.36-46, Nov./Dec. 1971

19. Davenport R.A. "Distributed database technology - a survey", Computer Networks, vol.2, no.3, July 1978, pp.155-167

20. Davenport R.A. "Distributed or centralised database?", Computer Journal, vol.21, no.1, Feb. 1978, pp.7-14

21. Deppe M.E., Fry J.P. "Distributed data bases: a summary of research", Computer Networks, vol.1, no.2, 1976, pp.130-138

22. Frailey D.J. "A practical approach to managing resources and avoiding deadlocks", CACM, May 1973, pp.323-329

23. Fry J.P., Maurer J. "Operational and technological issues in distributed data bases", Auerbach Report, 1977

24. Ghosh S.P. "Distributing a database with logical associations on a computer network for parallel searching", IEEE Trans. on Software Engineering, vol.SE-2, no.2, June 1976, pp.106-113

25. Grapa E., Belford C.G. "Some theorems to aid in solving the file allocation problem", Comm. ACM, vol.20, no.11, Nov.1977, pp.878-882

26. Habermann A.N. "Prevention of system deadlocks", CACM, vol.12, no.7, July 1969, pp.373-377

27. Hermann J. "Flow control in the ARPA Network", Computer Networks, vol.1, no.1, 1976, pp.65-76

28. Hobgood W.S. "The role of the network control program in Systems Network Architecture", IBM Systems Journal, vol.15, no.1, 1976, pp.39-52

29. Holt R.C. "Some deadlock properties of computer

systems", Computing Surveys, vol.4, no.3, Sept. 1972, pp.179-196

30. Hutchinson D.A., Riordon J.S., Mahmoud S.A. "A recursive algorithm for deadlock preemption in computer networks", Procs. IFIP, 1977, pp.241-245
31. Johnson P.R., Beeler M. "Notes on distributed data bases", Draft Report, BBN Inc., 1974
32. Karl M. "The distributed database of the information system of the German police", Procs. European Workshop on Distributed Computer Systems, 1974
33. Kimbelton S.R., Schneider G.M. "Computer communications networks: approaches, objectives and performance considerations", Computing Surveys, vol.7, no.3, 1975, pp.129-173
34. King P.F., Collmeyer A.J. "Data base sharing - an efficient mechanism for supporting concurrent processes", Procs. AFIPS NCC, vol.42, pp.271-275
35. Kleinrock L. "Queueing systems", vol.1, John Wiley and Sons, Inc., 1975
36. Levin K.D., Morgan H.L. "Optimal program and data locations in computer networks", Comm. ACM, vol.20, no.5, May 1977, pp.315-322
37. Levin K.D., Morgan H.L. "Optimizing distributed data bases: a framework for research", Procs. AFIPS NCC, vol.44, 1975, pp.473-478
38. Mahmoud S.A. "Resource allocation and file access control in distributed information networks", Ph.D. Thesis, Carleton University, Jan.1975

39. Mahmoud S.A., Riordon J.S. "Software controlled access to distributed data bases", INFOR, vol.15, no.1, Feb.1977, pp.22-36

40. Mahmoud S.A., Riordon J.S. "Protocol considerations for software controlled access methods in distributed databases", Procs. International Symposium on Computer Performance, Modelling, Measurement and Evaluation, March 1976, pp.241-256

41. Mahmoud S.A., Riordon J.S. "Optimal allocation of resources in distributed information networks", ACM Trans. on Data Base Systems, vol.1, no.1, March 1976, pp.66-78

42. Manning E.G., Peebles R.W. "A homogeneous network for data-sharing communications", Computer Networks, vol.1, no.4, 1977, pp.211-224

43. Manning E.G., Peebles R.W., Labetoulle J. "A homogeneous computer network analysis and simulation", Computer Networks, vol.1, no.4, 1977, pp.225-240

44. Marill T., Stern D. "The Datacomputer - a network data utility", Procs. AFIPS NCC, vol.44, 1975, pp.389-395

45. Maryanski F.J. "A survey of developments in distributed data base Management systems", (IEEE) Computer, vol.11, no.2, 1978, pp.28-38

46. Maryanski F. J. et al "A minicomputer-based distributed data base management system", Proc. NBS-IEEE Trends and Applications Symposium: Micro and Mini Systems, May 1976, pp.113-117

47. McFadyen J.H. "Systems network architecture: an

overview", IBM Systems Journal, vol.15, no.1, 1976, pp.4-23

48. Morgan D.E., Taylor D.J., Custeau G. "A survey of methods for improving computer network reliability and availability", (IEEE) Computer, vol.10, no.11, Nov. 1977, pp.42-51
49. Murphy J.E. "Resource allocation with interlock detection in a multitask system", Procs. FJCC, vol.33, pp.1169-1176, 1968
50. Paolini P., Pelagatti G., Schreiber F.A. "An application oriented approach to distributed data bases", Procs. AFCET, Journees de Formation, Bases de Donnees Reparties, Paris, March 1977, pp.139-151
51. Parry D. "Distributed data base management systems", Procs ONLINE Conference on DBMS, April 1976
52. Peebles R.W. "Design considerations for a distributed data access system", Ph.D. Thesis, (AD-775569), Wharton School of Finance and Commerce, 1973
53. Rosenkrantz D.J., Stearns R.E., Lewis II P.M. "System level concurrency control for distibuted database systems", ACM Trans. on Database Systems, vol.3, no.2, June 1978, pp.178-198
54. Schreiber F.A. "A framework for distributed data bases", Procs. International Computing Symposium, Belgium, April 1977, pp.475-482
55. Schreiber F.A. "Distributed data bases: some problems still to be solved", Procs. Convention Informatique, Paris, Sept.1975

56. Schreiber F.A. "Problems and models in distributed data base systems", Internal Report n.75-14, Lab.di Calcolatori, Politecnico di Milano, 1975

57. Shoshani A., Bernstein A.J. "Synchronisation in a parallel-accessed data base", Comm. ACM, vol.2, no.11, Nov.1969, pp.604-607

58. Stefferud E. "Economics of network delivery of computer services", Computer Networks, vol.1, no.1, 1976, pp.53-64

59. Taylor F.E., [National Computing Centre Ltd.] "The relative merits of distributed computing", ICS Procs, April 1977, pp.357

60. Thomas R.H. "A solution to the update problem for multiple copy databases which use distributed control", Bolt, Beranek, and Newman Inc., Report 3340, July 1976

61. Walden D.C. "A system for interprocess communication in a resource sharing computer network", CACM, vol.15, no.4, April 1972, pp.221-230

62. Wallentine V.E., Maryanski F.J. "Implementation of a distributed database system", TR-CS 76-03, Feb, 1976, Kansas State University

63. Whitney V.K.M. "A study of optimal file assignement and communication network configuration", Ph.D. Thesis, U. Of Michigan, 1970

64. "DECNET-11 Programmers Guide and Reference Manual", Digital Equipment Corporation, Maynard, Mass., 1975

65. "DECNET-11 V1.2 Release Notes", Digital Equipment Corporation, Maynard, Mass., 1977

66. "RSX-11M V3.1, Executive Reference Manual", Digital Equipment Corporation, Maynard, Mass., 1977

67. "Structured FORTRAN Programmers Manual", Institutional Research Laboratory of Electronics, Cambridge, Mass., 1977

APPENDIX A

Program Listings of the DDACS

INDEX

Subroutine COM

```
      IMPLICIT INTEGER (A-Z)
/*      COMMON AREAS
        THIS IS THE COMMON AREA FOR ALL THE ROUTINES. IT IS INSERTED IN
        THE CODE OF A ROUTINE BY THE COMMAND
                                  $INSERT COM.SF
*/
      COMMON /DLOCK/ DFILE(3),DPRED(3),DBOSS(3),DREPS(3),DLIST(3),
     1               DSTAT(3),DNUM,DPROC
/*      DLOCK:  THE DEADLOCK DESCRIPTOR TABLE
                WHENEVER A DEADLOCK RECOVERY MUST BE SUSPENDED BECAUSE
                PREDECESSOR LISTS ARE REQUIRED FROM SOME REMOTE STATION,
                THE VARIABLES WHICH DESCRIBE THE PRESENT STATE OF THE
                RECOVERY PROCESSING ARE STORED IN THIS TABLE.
        DFILE:  THE FILE WHOSE PREDECESSOR LIST IS BEING RECOVERED
        DPRED:  THE FILE'S IMMEDIATE PREDECESSOR IN THE LOOP
        DBOSS:  THE BOSS OF THIS DEADLOCK
        DREPS:  THE NUMBER OF REPLIES STILL OUTSTANDING FOR PREDECESSOR
                LIST REQUESTS
        DLIST:  THE UNION OF THE PREDECESSOR LIST REPLIES RECEIVED SO FAR
        DSTAT:  THE STATUS OF THE RECOVERY PROCESSING   1 - INITIAL
                2 - SUSPENDED              3 - RESUMED
        DNUM:   THE NUMBER OF DEADLOCKS CURRENTLY BEING RECOVERED
                AT THIS STATION
        DPROC:  THE INDEX OF THE DEADLOCK DESCRIPTOR IN THE TABLE DLOCK
*/
      COMMON /DNET/ STAT(2,10),WORK(224),WORDS,NODE(3),TASK(3),LEN
/*      DNET:   DECNET VARIABLES
                THE VARIABLES REQUIRED BY THE DECNET USER ROUTINES ARE
                CONTAINED IN THIS AREA
        STAT:   THE STATUS OF AN ACTION ON A COMMUNICATION LINK. THE LINK
                NUMBER IS THE INDEX TO THE STATUS TABLE.
                E.G. STAT(1,N) IS THE STATUS OF THE LAST ACTION ON LINK N.
                STAT(1,N)=1 INDICATES THAT THE ACTION WAS COMPLETED
                SUCCESSFULLY
        WORK:   THE DECNET WORK AREA
        WORDS:  THE NUMBER OF WORDS IN THE DECNET WORK AREA
        NODE:   THE NAME OF THE STATION TO WHICH A COMMUNICATION LINK
                IS TO BE MADE
        TASK:   THE NAME OF THE TASK TO WHICH A COMMUNICATION LINK
                IS TO BE MADE
        LEN:    THE LENGTH OF A MESSAGE, IN BYTES, WHICH IS TO BE SENT
                ON A LINK
```

Subroutine COM (Continued)

```
*/
      COMMON /VARS/ MSG(5,5),FILE,PROC,RCODE,ERR,LOCAL,REMOTE,RTN,
     1              START,LENGTH,LACS,DISPL,LFILES,BOSS,BPRED,PRED,LIST,
     2              PRLIST,MASK,SUSPEN,LOCK,RECV
/*       VARS:    THE VARIABLES USED BY THE ACCESS CONTROLLER
         MSG:     THE MESSAGE BUFFER FOR THE SEND AND RECEIVE ROUTINES.
                  A 5 WORDUFFER IS RESERVED FOR EACH OF 5 POSSIBLE
                  COMMUNICATION LINKS.
                  THE NUMBER OF THE LINK IS THE INDEX OF ITS BUFFER.
         FILE:    THE FILE WHOSE DATA STRUCTURES ARE CURRENTLY BEING
                  MANIPULATED.
                  THIS VARIABLE IS AN INDEX TO THE FILE DESCRIPTOR TABLE
         PROC:    THE PROCESS WHOSE ACCESS REQUEST IS CURRENTLY BEING
                  PROCESSED.
                  THIS VARIABLE IS AN INDEX TO THE PROCESS DESCRIPTOR TABLE.
         RCODE:   THE RETURN CODE IN REPLIES TO FILE REQUESTS
         ERR:     LOG CODE FOR THE LOG SUBROUTINE
         LOCAL:   THE NAME OF THE LOCAL STATION
         REMOTE:  THE NAME OF THE REMOTE STATION
         RTN:     THE NUMBER OF THE SUBROUTINE WHICH WAS CONTROL.
         START:   STARTING POINT OF THE PROCESS DESCRIPTORS IN THE
                  PROCESS DESCRIPTOR TABLE
         LENGTH:  THE LENGTH OF THE PROCESS DESCRIPTOR TABLE
         LACS:    THE NUMBER OF REMOTE ACCESS CONTROLLERS IN THE SYSTEM
         DISPL:   THE DISPLACEMENT OF THE LOCAL FILE DIRECTORY IN THE
                  GLOBAL FILE DIRECTORY
         LFILES:  THE NUMBER OF LOCAL FILES
         BOSS:    THE BOSS OF THE DEADLOCK CURRENTLY UNDER CONSIDERATION.
                  THIS IS A POINTER TO THE FILE DESCRIPTOR TABLE
         BPRED:   THE IMMEDIATE PREDECESSOR OF THE BOSS IN THE LOOP.
         PRED:    AN IMMEDIATE PREDECESSOR IN THE W-GRAPH OF THE FILE
                  UNDER CONSIDERATION
         LIST:    A BIT LIST INDICATING FILES BY THE POSITION OF THEIR
                  ENTRY IN THE GLOBAL DIRECTORY. USED MAINLY FOR INDICATING
                  WHICH PREDECESSOR LISTS ARE REQUESTED.
                  A FILE IS REPRESENTED IN THE LIST AS FOLLOWS:
                  FILE N IN THE DIRECTORY IS IN THE LIST IF
                  BIT N IN THE LIST = 1
                  I.E.   LIST.AND.2**(N-1) = 0 INDICATES BIT N IS ZERO
                  ALL BIT LIST VARIABLES REPRESENT FILES IN THIS WAY:
                  LIST, PRLIST, PLIST, PL, IP.
         PRLIST:  STORAGE AREA FOR THE UNION OF PREDECESSOR LISTS
         MASK:    A BIT LIST INDICATING ALL THE LOCAL FILES
         SUSPEN:  A BOOLEAN VARIABLE WHICH SHOWS THAT THE PROCESSING
                  OF THE CURRENT REQUEST HAS BEEN SUSPENDED.
         LOCK:    A BOOLEAN VARIABLE WHICH SHOWS WHETHER A POTENTIAL
                  OR REAL DEADLOCK HAS BEEN DETECTED.
         RECV:    A BOOLEAN VARIABLE WHICH SHOWS WHETHER THE RECEIVE
                  MESSAGE ROUTINE SHOULD BE INVOKED FOR A PARTICULAR LINK
                  AS A RESULT OF THE LAST ACTION OF THE ACCESS CONTROLLER.
*/
```

Subroutine COM (Continued)

```
      COMMON /PDT/ PNAME(10),STTN(10),STATE(10),OWNED(10),REQST(10),
     1             REPLS(10),PLIST(10),QUE(10)
/*       PDT:    THE PROCESS DESCRIPTOR TABLE
                 THE VARIALBE 'PROC' IS THE INDEX TO THIS TABLE.
                 THIS IS ALSO THE NUMBER OF THE COMMUNICATION LINK
                 IF THE ENTRY IS FOR A LOCAL PROCESS.
                  THE ENTRIES FROM THE BEGINNING OF THE TABLE
                 TO POSITION START-1 ARE RESERVED FOR INFORMATION
                 ON THE REMOTE ACCESS CONTROLLERS.
                 THESE ENTRIES ARE ALSO INDEXED BY THE
                 NUMBER OF THE COMMUNICATION LINK.
           PNAME: THE PROCESS NAME
           STTN:  THE STATION IN WHICH THE PROCESS RUNS
           STATE: THE STATUS OF THE PROCESSING OF THE FILE REQUEST
                  OR FILE RELEASE FROM THIS PROCESS.
                  0 - NO PROCESSING              1 - INITIAL
                  2 - SUSPENDED                  3 - RESUMED
           OWNED: THE FILES OWNED BY THE PROCESS. A BIT LIST
           REQST: THE FILE REQUESTED BY THE PROCESS. THIS IS AN
                  INDEX TO THE FILE DESCRIPTOR TABLE
           REPLS: THE NUMBER OF OUTSTAINING REPLIES TO PREDECESSOR
                  LIST REQUESTS FOR THIS PROCESS.
           PLIST: THE UNION OF THE PREDECESSOR LIST REPLIES RECEIVED
                  SO FAR.
           QUE:   THE FILE REQUEST QUEUE.
                  A LINKED LIST OF QUEUED REQUESTS.
*/
      COMMON /DIRCTY/FNAME(10),HOST(10),MODE(10),OWNER(10),WAIT(10),
     1               PL(10),IP(10),IS(10)
/*       DIRCTY: THE FILE DESCRIPTOR TABLE AND THE FILE DIRECTORY
                 THE VARIABLE 'FILE' IS THE INDEX TO THIS TABLE.
                 THE VARIABLES 'FNAME' AND 'HOST' FORM THE GLOBAL
                 AND LOCAL FILE DIRECTORY. THE REMAINING VARIABLES
                 FORM THE FILE DESCRIPTOR FOR THE LOCAL FILES.
           FNAME: THE FILE NAME
           HOST:  THE STATION AT WHICH THE FILE IS STORED.
           MODE:  THE CURRENT STATUS OF THE FILE.
                  0 - FREE        1 - ALLOCATED          2 - DEADLOCKED
           OWNER: THE PROCESS WHICH OWNES THE FILE. THIS IS AN INDEX
                  TO THE PROCESS DESCRIPTOR TABLE
           WAIT:  A POINTER TO THE HEAD OF THE QUEUE FOR THIS FILE.
           PL:    THE PREDECESSOR LIST.   A BIT LIST
           IP:    THE IMMEDIATE PREDECESSOR LIST.   A BIT LIST
           IS:    THE IMMEDIATE SUCCESSOR POINTER.  AN INDEX TO
                  THE GLOBAL FILE DIRECTORY
*/
      LOGICAL*1 SUSPEN,LOCK,RECV
```

Subroutine EXEC

```
/*      EXEC.SF
        THE EXECUTIVE ROUTINE
        THIS IS A CYCLIC ROUTINE. IT ACCEPTS MESSAGES FROM THE RECEIVE
        MESSAGE ROUTINE AND SELECTS THE SUBROUTINE TO PROCESS THE
        MESSAGE ACCORDING TO THE MESSAGE CODE.
*/
      %INSERT COM.SF
      LOGICAL*1 STOP
      RTN=1
      CALL INIT                        ;INITIALIZATION
      .DO
          CALL MWAIT                     ;ACCEPT A MESSAGE
          RECV=.FALSE.
          .SWITCHON MSG(1,PROC)
C
C                       FILE REQUEST FROM LOCAL PROCESS
      .CASE 1
              STOP=.FALSE.
              I=0
              .UNTIL STOP .OR. I == 10
                  I=I+1
                  .IF FNAME(I) == MSG(2,PROC)        ;VALIDATE THE
                      STOP=.TRUE.                    ;FILE NAME
                  .FI
              .REPEAT
              .IF STOP
                  FILE=I
                  CALL FLREQ                 ;FILE REQUEST
              .ELSE
                  RCODE=-2                  ;REJECT REQUEST -
                  CALL SEND                 ;UNKNOWN FILE NAME
              .FI
              RECV=.TRUE.           ;REMEMBER TO ISSUE A RECEIVE
              .ENDCASE              ;MESSAGE ON THIS LINK
C
C                       FILE RELEASE FROM LOCAL PROCESS
      .CASE 2
              STOP=.FALSE.
              I=0
              .UNTIL STOP .OR. I == 10
                  I=I+1
                  .IF FNAME(I) == MSG(2,PROC)        ;VALIDATE THE
                      STOP=.TRUE.                    ;FILE NAME
                  .FI
              .REPEAT
              .IF STOP
                  FILE=I
                  CALL RELEAS             ;RELEASE THE FILE
              .ELSE
                  RCODE=-2               ;REJECT THE RELEASE
                  CALL SEND              ;- UNKNOWN FILE NAME
              .FI
```

Subroutine EXEC (Continued)

```
            RECV=.TRUE.          ;REMEMBER TO ISSUE A RECEIVE
            .ENDCASE             ;MESSAGE ON THIS LINK
C
C                   FILE REQUEST FROM REMOTE PROCESS
        .CASE 3
            CALL REMRQ
            .ENDCASE
C
C                   FILE RELEASE FROM REMOTE PROCESS
        .CASE 4
            FILE=MSG(2,PROC)
            .IF PNAME(OWNER(FILE)) == MSG(3,PROC)    ;VALIDATE
                PROC=OWNER(FILE)                     ;THE FILE'S OWNER
                CALL RELEAS
            .ELSE
                ERR=4
                CALL LOG             ;LOG ERROR
            .FI
            .ENDCASE
C
C                   REQUEST FOR REMOTE FILE GRANTED
        .CASE 5
            I=START-1
            STOP=.FALSE.
            .UNTIL STOP .OR. I == START+LENGTH
                I=I+1
                .IF PNAME(I) == MSG(3,PROC)       ;FIND THE
                    STOP=.TRUE.                   ;REQUESTING PROCESS
                .FI
            .REPEAT
            .IF STOP
                FILE=MSG(2,PROC)
                PROC=I
                CALL RMGRN           ;REMOTE REQUEST GRANTED
            .ELSE
                ERR=4
                CALL LOG        ;LOG THE ERROR
            .FI
            RECV=.TRUE.
            .ENDCASE
C
```

Subroutine EXEC (Continued)

```
C                       REQUEST FOR REMOTE FILE REJECTED
      .CASE 6
            I=START-1
            STOP=.FALSE.
            .UNTIL STOP .OR. I == START+LENGTH
                I=I+1
                .IF PNAME(I) == MSG(3,PROC)    ;FIND THE
                    STOP=.TRUE.                ;REQUESTING PROCESS
                .FI
            .REPEAT
            .IF STOP
                FILE=MSG(2,PROC)
                RCODE=MSG(4,PROC)
                PROC=I
                CALL REJECT
            .ELSE
                ERR=4
                CALL LOG
            .FI
            RECV=.TRUE.
            .ENDCASE
C
C                       PREDECESSOR LIST PROPAGATION
      .CASE 7
            FILE=MSG(2,PROC)
            PRED=MSG(3,PROC)
            PRLIST=MSG(4,PROC)
            CALL PROPGT       PREDECESSOR PROPAGATION
            .ENDCASE
C
C                       BOSS SELECTION (DEADLOCK BREAK)
      .CASE 8
            FILE=MSG(2,PROC)
            PRED=MSG(3,PROC)
            BOSS=MSG(4,PROC)
            BPRED=MSG(5,PROC)
            CALL DBREAK
            .ENDCASE
C
C                       DEADLOCK RECOVERY MESSAGE
      .CASE 9
            FILE=MSG(2,PROC)
            BPRED=MSG(3,PROC)
            BOSS=MSG(4,PROC)
            CALL RECOVR
            .ENDCASE
C
C                       UPDATE IMMEDIATE SUCCESSOR POINTER
      .CASE 10
            FILE=MSG(2,PROC)
            LIST=MSG(4,PROC)
            CALL UPDIS
            .ENDCASE
```

Subroutine EXEC (Continued)

```
C
C                       PREDECESSOR LIST REQUEST MESSAGE
      .CASE 11
              LIST=MSG(4,PROC)
              CALL PLRQST
              .ENDCASE
C
C                       PREDECESSOR LIST REPLY
      .CASE 12
              PRLIST=MSG(4,PROC)
              PROC=MSG(3,PROC)
              CALL PLRPLY
              .ENDCASE
C
C                               REMOTE ACCESS CONTROLLER STARTUP
      .CASE 13
              CALL BEGIN
              .ENDCASE
C
C                               REMOTE ACCESS CONTROLLER CLOSE DOWN
      .CASE 14
              ERR=MSG(4,PROC)
              CALL CLDOWN
              .ENDCASE
          .ENDSWITCH
      .REPEAT.
      END
```

Subroutine INIT

```
      SUBROUTINE INIT
/*      THE INITIALIZATION ROUTINE
        ALL THE ACCESS CONTROLLER VARIABLES ARE INITIALISED.
        THE DECNET INITIALIZATION ROUTINES ARE INVOKED AND
        THE COMMUNICATION LINKS WITH THE REMOTE ACCESS CONTROLLERS
        ARE ESTABLISHED. THE LOG AND LOGGING REQUIREMENTS ARE
        ALSO INITIALISED.
*/
      %INSERT COM.SF
      COMMON /LOGS/ LOGON,MSGS,EVNS,TABS
/*      LOGS:  THE BOOLEAN VARIABLES INDICATE THE LOGGING REQUIREMENTS.
        LOGON: LOGGING IS REQUIRED          T/F
        MSGS:  LOG ALL MESSAGES             T/F
        EVNS:  LOG ALL EVENTS               T/F
        TABS:  LOG ALL TABLES               T/F
*/
      LOGICAL*1 LOGON,MSGS,EVNS,TABS
      DATA LOGON,MSGS,EVNS,TABS/4*.FALSE./
      DATA YES/'Y '/
      DATA FNAME/'F1','F2','F3','F4','F5',      ;INITIALISE DIRECTORY
     1  'R1','R2','R3','R4','R5'/                ;FILE NAMES
      DATA LOCAL,REMOTE/'L1','L2'/            ;STATION NAMES
      DATA SUSPEN,LOCK,RECV/3*.FALSE./
      DATA MSG/25*0/
      DATA MASK/"037/        A BIT LIST OF ALL LOCAL FILES
      DATA NODE,TASK/'HO','ST',4*'  '/
      SRTN=RTN       ;SAVE CALLING ROUTINE
      RTN=2
      N=10             ;   MAXIMUM NUMBER OF DECNET LINKS
      WORDS=14+21*N
C
C                      ;    INITIALIZE PROCESS DESCRIPTOR TABLE
      .FOR I= 1 TO 10
        PNAME(I)=0
        STTN(I)=0
        OWNED(I)=0
        REQST(I)=0
        STATE(I)=0
        REPLS(I)=0
        PLIST(I)=0
        QUE(I)=0
      .REPEAT
```

Subroutine INIT (Continued)

```
C
C                   ;   INITIALIZE FILE DESCRIPTOR TABLE
      .FOR I= 1 TO 10
         .IF I > 5
            HOST(I)=REMOTE          ; FIRST 5 FILES IN TABLE
         .ELSE                      ;ARE LOCAL
            HOST(I)=LOCAL           ;NEXT 5 FILES IN TABLE
         .FI                        ;ARE REMOTE
         MODE(I)=0
         OWNER(I)=0
         WAIT(I)=0
         PL(I)=0
         IP(I)=0
         IS(I)=0
      .REPEAT
C
C                   ;   INITIALIZE DEADLOCK DESCRIPTOR TABLE
      .FOR I= 1 TO 3
         DFILE(I)=0
         DPRED(I)=0
         DBOSS(I)=0
         DREPS(I)=0
         DLIST(I)=0
         DSTAT(I)=0
      .REPEAT
C
C                   ;   INITIALIZE COMMON VARIABLES
      FILE=0
      PROC=0
      DPROC=0
      START=2
      LENGTH=8
      LEN=10
      LACS=1
      DISPL=0
      LFILES=5
      DNUM=0
```

Subroutine INIT (Continued)

```
C
C                ;    INITIALZE THE NETWORK
      CALL NTINIT(STAT,WORDS,WORK)
      .IF STAT(1,1) == 1
C
C                     INITIALISE THE LINK WITH L2
          TASK(1)=REMOTE
          PROC=1        USE LINK 1 FOR REMOTE ACCESS CONTROLLER
          PNAME(PROC)=REMOTE
          STTN(PROC)=REMOTE
         OWNED(PROC)=OWNED(PROC).OR.(.NOT.MASK)     ;
                ;FILES OWNED BY REMOTE ACCESS CONTROLLER
          CALL NTCONW(PROC,STAT(1,PROC),ICON,NODE,TASK)
                ;CONNECT LINK
          .IF STAT(1,PROC) == 1
             RCODE=13
             CALL SEND          ;SEND STARTUP MESSAGE
          .ELSE
             WRITE(12,.LF) STAT(1,PROC)
             (' L1 FAILED TO INITIALIZE THE LINK TO L2,  ERROR =',I5)
          .FI
```

Subroutine INIT (Continued)

```
C                ;   INITIALIZE THE LOG
      WRITE(12,.LF)
      ('$L1 - LOG ?  [Y/N]:')
      READ(12,.LF) ANS
      (A2)
      .IF ANS == YES
          LOGON=.TRUE.
          CALL ASSIGN(11,'L1.DAT')
          WRITE(12,.LF)
          ('$L1 - LOG MESSAGE ?  [Y/N]:')
          READ(12,.LF) ANS
          (A2)
          .IF ANS == YES
              MSGS=.TRUE.
          .FI
          WRITE(12,.LF)
          ('$L1 - LOG EVENTS ?  [Y/N]:')
          READ(12,.LF) ANS
          (A2)
          .IF ANS == YES
              EVNS = .TRUE.
          .FI
          WRITE(12,.LF)
          ('$L1 - LOG TABLES ? [Y/N]:')
          READ (12,.LF) ANS
          (A2)
          .IF ANS == YES
              TABS=.TRUE.
          .FI
      .FI
      CALL LOG
      WRITE(12,.LF)
      (' L1 INITIALIZATION COMPLETE')
      RTN=SRTN       ;RESTORE CALLING ROUTINE
      RETURN
.ELSE
      ERR=9          ;DECNET ERROR
      WRITE(12,.LF) STAT(1,1)
      (' L1 FAILED TO INITIALIZE THE NETWORK,  ERROR =',I5)
      STOP
.FI
END
```

Subroutine INIT2

```
      SUBROUTINE INIT
      %INSERT COM.SF
/*      THIS IS THE INITIALISATION ROUTINE FOR ACCESS CONTROLLER L2
        IT IS IDENTICAL TO INIT EXCEPT THAT IT DOES NOT ATTEMPT TO
        CONNECT A LINK WITH ACCESS CONTROLLER L1.
*/
      COMMON /LOGS/ LOGON,MSGS,EVNS,TABS
/*
        LOGS:    THE BOOLEAN VARIABLES FOR LOGGING REQUIREMENTS
        LOGON:   LOGGING IS REQUIRED                    T/F
        MSGS:    LOG ALL MESSAGES                       T/F
        EVNS:    LOG ALL EVENTS                         T/F
        TABS:    LOG ALL TABLES                         T/F
*/

      LOGICAL*1 LOGON,MSGS,EVNS,TABS
      DATA LOGON,MSGS,EVNS/3*.FALSE./
      DATA YES/'Y '/
      DATA FNAME/'F1','F2','F3','F4','F5',      ;REMOTE FILES
     1  'R1','R2','R3','R4','R5'/                ;LOCAL FILES
      DATA LOCAL,REMOTE/'L2','L1'/
      DATA SUSPEN,LOCK,RECV/3*.FALSE./
      DATA MSG/25*0/
      DATA MASK/"1740/          ;A BIT LIST OF ALL LOCAL FILES
      DATA NODE,TASK/'HO','ST',4*'  '/
      SRTN=RTN
      RTN=2
      N=10                ;   MAXIMUM NUMBER OF DECNET LINKS
      WORDS=14+21*N
C
C                         ;   INITIALIZE PROCESS DESCRIPTOR TABLE
      .FOR I= 1 TO 10
         PNAME(I)=0
         STTN(I)=0
         OWNED(I)=0
         REQST(I)=0
         STATE(I)=0
         REPLS(I)=0
         PLIST(I)=0
         QUE(I)=0
      .REPEAT
```

Subroutine INIT2 (Continued)

```
C
C                   ;   INITIALIZE FILE DESCRIPTOR TABLE
      .FOR I= 1 TO 10
          .IF I > 5
              HOST(I)=LOCAL          ;FIRST 5 FILES IN TABLE
          .ELSE                      ;ARE REMOTE
              HOST(I)=REMOTE         ;NEXT 5 FILES IN TABLE
          .FI                        ;ARE LOCAL
          MODE(I)=0
          OWNER(I)=0
          WAIT(I)=0
          PL(I)=0
          IP(I)=0
          LS(I)=0
      .REPEAT
C
C                   ;   INITIALIZE DEADLOCK DESCRIPTOR TABLE
      .FOR I= 1 TO 3
          DFILE(I)=0
          DPRED(I)=0
          DBOSS(I)=0
          DREPS(I)=0
          DLIST(I)=0
          DSTAT(I)=0
      .REPEAT
C
C                   ;   INITIALIZE COMMON VARIABLES
      FILE=0
      PROC=0
      DPROC=0
      START=2
      LENGTH=8
      LEN=10
      LACS=1
      DISPL=5
      LFILES=5
      DNUM=0
```

Subroutine INIT2 (Continued)

```
C
C                ;   INITIALZE THE NETWORK
      CALL NTINIT(STAT,WORDS,WORK)
      .IF STAT(1,1) == 1
C
C                   INITIALIZE ENTRY FOR L1 IN PROCESS TABLE
          PROC=1
          PNAME(PROC)=REMOTE
          STTN(PROC)=REMOTE
          OWNED(PROC)=OWNED(PROC).OR.(.NOT.MASK)
                ;FILES OWNED BY REMOTE ACCESS CONTROLLER L1
C
C                ;   INITIALIZE THE LOG
          WRITE(12,.LF)
          ('$L2 - LOG ?  [Y/N]:')
          READ(12,.LF) ANS
          (A2)
          .IF ANS == YES
              LOGON=.TRUE.
              CALL ASSIGN(11,'L2.DAT')
              WRITE(12,.LF)
              ('$L2 - LOG MESSAGE ?  [Y/N]:')
              READ(12,.LF) ANS
              (A2)
              .IF ANS == YES
                  MSGS=.TRUE.
              .FI
              WRITE(12,.LF)
              ('$L2 - LOG EVENTS ?  [Y/N]:')
              READ(12,.LF) ANS
              (A2)
              .IF ANS == YES
                  EVNS = .TRUE.
              .FI
              WRITE(12,.LF)
              ('$L2 - LOG TABLES ? [Y/N]:')
              READ (12,.LF) ANS
              (A2)
              .IF ANS == YES
                  TABS=.TRUE.
              .FI
          .FI
          CALL LOG
          WRITE(12,.LF)
          (' L2 INITIALIZATION COMPLETE')
          RTN=SRTN
          RETURN
      .ELSE
          ERR=9
          WRITE(12,.LF) STAT(1,1)
          (' L2 FAILED TO INITIALIZE THE NETWORK,  ERROR =',I5)
          STOP
      .FI
      END
```

Subroutine MWAIT

```
      SUBROUTINE MWAIT
/*      THE WAIT FOR NEXT MESSAGE ROUTINE
        THIS ROUTINE RECEIVES MESSAGES ON THE DECNET COMMUNICATION
        LINKS. IF NO MESSAGES HAVE BEEN SENT TO THE ACCESS CONTROLLER,
        IT WAITS FOR THE NEXT MESSAGE OR LINK CONMNECTION REQUEST.
*/
      %INSERT COM.SF
      SRTN=RTN        ;SAVE CALLING ROUTINE
      RTN=3
      .IF RECV
          .IF SITN(PROC) ==  LOCAL .AND. REQST(PROC) == 0
              .IF OWNED(PROC) == 0
C * COMMENT ****       CALL NTDISW(PROC,STAT(1,PROC))       DISCONNECT LINK
                  ERR=6
                  CALL LOG      ;LOG THE DISCONNECT
                  MSG(1,PROC)=0
              .ELSE
                  CALL NTRCV(PROC,STAT(1,PROC),LEN,MSG(1,PROC))
                      ;RECEIVE ON THIS LINK
                  ERR=7
                  CALL LOG        ;LOG RECEIVE ISSUED
              .FI
          .FI
      .ELSE
          PROC=1        ;RECEIVE FROM REMOTE ACCESS CONTROLLER
          CALL NTRCV(PROC,STAT(1,PROC),LEN,MSG(1,PROC))
      .FI
```

Subroutine MWAIT (Continued)

```
      RCODE=0
      PROC=0
               ;SETTING THE LINK NUMBER (PROC) TO ZERO ALLOWS THE
               ;ACCESS CONTROLLER TO WAIT FOR THE NEXT MESSAGE OR
      CALL NTWAIT(PROC)          ;LINK CONNECTION REQUEST
      .IF PROC == 0        ;  PROC = 0 IS A CONNECTION REQUEST
          CALL NTCGTW(STAT(1,10),ICON,NODE,TASK)    ; GET CONNECTION INFORMATI
          IF (STAT(1,10).NE.1) GO TO 800
         .IF TASK(1) == REMOTE
             PROC=TASK(1)
          .FI
          CALL INACT       ;FIND INACTIVE ENTRY IN PDT
          CALL NTCONW(PROC,STAT(1,PROC),ICON,NODE,TASK)      ;CONNECT LINK
          IF (STAT(1,PROC).NE.1) GO TO 800
          .IF RCODE ^= 0           ;   RCODE = 0 INACTIVE ENTRY FOUND
              CALL SEND        ;REJECT IF NO INACTIVE ENTRY IN PDT
              CALL NTDISW(PROC,STAT(1,PROC))     ;AND DISCONNECT LINK
              IF (STAT(1,PROC).NE.1) GO TO 800
          .ELSE
              .IF PNAME(PROC) ^= REMOTE
                  PNAME(PROC)=TASK(1)
                  STTN(PROC)=LOCAL    ;
              .FI
          .FI
          CALL NTRCVW(PROC,STAT(1,PROC),LEN,MSG(1,PROC))
                ;RECEIVE A MESSAGE ON LINK PROC
      .FI
/*
  A FAULT IN DECNET CAUSES ALL OUTSTANDING RECEIVES TO BE LOST
  WHEN A LINK IS DISCONNECTED. THIS IS RECOGNISED BY A MESSAGE
  RECEIVED NOTIFICATION ON THE LINK WHERE NO MESSAGE WAS ACTUALLY
  RECEIVED. ALL THE 'RECEIVES' WHICH WERE LOST ARE REISSUED.
*/
      .IF MSG(1,PROC) == 0
          SPROC=PROC
          .FOR PROC=1 TO 10
              .IF OWNED(PROC) ^= 0 .AND. REQST(PROC) == 0
                  CALL NTRCV(PROC,STAT(1,PROC),LEN,MSG(1,PROC))
              .FI
          .REPEAT
          PROC=SPROC
C
C
      .FI
      IF(STAT(1,PROC).EQ.1) GO TO 900
800   ERR=9          ;DECNET ERROR
900   CALL LOG
      RTN=SRTN       ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine INACT

```
      SUBROUTINE INACT
/*      THE FIND ACTIVE ENTRY IN PDT ROUTINE
        IF THE VARIBLE 'PROC' IS ZERO, AN INACTIVE ENTRY IS SEARCHED
        FOR IN THE PDT. OTHERWISE, THE ENTRY FOR THE REMOTE ACCESS
        CONTROLLER, WHOSE NAME IS IN 'PROC', IS SOUGHT.
        INACTIVE ENTRY:          FILES OWNED=0    FILE REQUESTED=0
*/
      %INSERT COM.SF
      LOGICAL*1 STOP
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=4
      STOP=.FALSE.
      RCODE=0
      .IF PROC == 0
          PROC=START-1           ;FIND INACTIVE ENTRY IN PDT
          .UNTIL STOP .OR. PROC == LENGTH
              PROC=PROC+1
              .IF OWNED(PROC) == 0 .AND. REQST(PROC) == 0
                  STOP=.TRUE.
              .FI
          .REPEAT
          .IF .NOT.STOP
              RCODE=-4
              PROC=LENGTH+1
          .FI
      .ELSE
          I=0
          .UNTIL STOP .OR. I == START-1    ;FIND THE ENTRY FOR THE
              I=I+1                        ;REMOTE ACCESS CONTROLLER
              .IF PNAME(I) == TASK(1)      ;NAMED IN 'PROC'
                  STOP=.TRUE.
              .FI
          .REPEAT
          .IF STOP
              PROC=I
          .ELSE
              RCODE=-6                     ;LAST LINK IS USED FOR
              PROC=LENGTH+1                ;REJECTION OF CONNECTION
          .FI                              ;REQUESTS
      .FI
      CALL LOG
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine REJECT

```
      SUBROUTINE REJECT
/*      THE REJECT FILE REQUEST ROUTINE
        THE FILE ACCESS REQUEST IS REJECTED AND THE PDT ENTRY
        FOR THE PROCESS IS UPDATED. IF THE PROCESS IS REMOTE,
        ALL NON-LOCAL FILES ARE DELETED FROM THE 'FILES OWNED'
        FIELD IN ITS PDT ENTRY.
        THEN THE ENTRY BECOMES INACTIVE ONCE THE PROCESS RELEASES
        ALL ITS LOCAL FILES.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=5
      REQST(PROC)=0
      .IF STTN(PROC) ^= LOCAL                    ;DELETE ALL NON-LOCAL
          OWNED(PROC)=OWNED(PROC).AND.MASK       ;FILES OWNED BY
      .FI                                        ;REMOTE PROCESS
      CALL SEND                  ;SEND REJECTION MESSAGE
      CALL LOG
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine FLREQ

```
      SUBROUTINE FLREQ
./*     FILE REQUEST FROM A LOCAL PROCESS
       THE REQUEST IS GRANTED IF THE FILE IS LOCAL AND FREE.
       IF THE PROCESS OWNS NO OTHER FILES OR THE DEADLOCK TEST
       PROVES FALSE, THE REQUEST IS QUEUED. A REQUEST FOR A REMOTE
       FILE IS SENT TO THE STATION WHERE THE FILE IS STORED.
 /*
      %INSERT COM.SF
      DATA FREE/0/                ;FILE MODE=0 INDICATES FILE IS FREE
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=6
      CALL LOG
      .IF HOST(FILE) == LOCAL
          .IF MODE(FILE) == FREE
              CALL ALLOC          ;ALLOCATE A FREE FILE
          .ELSE
              .IF OWNED(PROC) == 0
                  CALL QUEUE              ;QUEUE THE REQUEST
              .ELSE
                  CALL DEADLK             ;DEADLOCK CHECK
                  .IF .NOT.SUSPEN
                      .IF LOCK
                          RCODE=-3        ;REJECT DUE TO
                          CALL REJECT     ;POTENTIAL DEADLOCK
                      .ELSE
                          CALL QUEUE      ;QUEUE THE REQUEST
                      .FI
                  .FI
              .FI
          .FI
      .ELSE
          CALL RQSEND             ;REQUEST FOR A REMOTE FILE
      .FI
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine RELEAS

```
      SUBROUTINE RELEAS
/*    THE FILE RELEASE ROUTINE
      THE FILE IS DELETED FROM THE PROCESS'S PDT ENTRY. IF THE FILE
      IS REMOTE, A RELEASE MESSAGE IS SENT TO THE REMOTE ACCESS
      CONTROLLER. OTHERWISE, THE FILE IS ALLOCATED TO THE REQUEST AT
      THE HEAD OF ITS QUEUE, IF ANY.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=7
      AMASK=2**(FILE-1)
      CALL LOG
      .IF HOST(FILE) == LOCAL
          CALL ALLOCF              ;UPDATE FDT ENTRY
          .IF .NOT. SUSPEN
              OWNED(PROC)=OWNED(PROC).AND.(.NOT.AMASK)  ;DELETE FROM
              .IF STTN(PROC) == LOCAL                   ;PDT ENTRY
                  RCODE=1                  ;ACKNOWLEDGE RELEASE
                  CALL SEND                ;FROM LOCAL PROCESS
              .ELSE
                  OWNED(PROC)=OWNED(PROC).AND.MASK
        ;DELETE REMOT FILES FROM PDT ENTRY IF PROCESS IS REMOTE
              .FI
              .IF WAIT(FILE) ^= 0          ;ALLOCATE THE FILE
                  SPROC=PROC               ;SAVE PROCESS INDEX
                  PROC=WAIT(FILE)
                  WAIT(FILE)=QUE(PROC)     ;RESET HEAD OF QUEUE
                  RECV=.TRUE.              ;ISSUE RECEIVE ON THIS LINK
                  CALL ALLOC               ;ALLOCATE FILE TO HEAD OF QUEUE
                  PROC=SPROC               ;RESTORE PROCESS INDEX
              .FI
          .FI
      .ELSE
          RCODE=4                  ;SEND RELEASE MESSAGE TO REMOTE A.C.
          CALL SEND                ;IF FILE IS REMOTE
          OWNED(PROC)=OWNED(PROC).AND.(.NOT.AMASK)      ;DELETE FROM
                                                        ;PDT ENTRY
          RCODE=1                  ;ACKNOWLEDGE RELEASE
          CALL SEND                ;FROM LOCAL PROCESS
      .FI
      .IF OWNED(PROC)==0                    ;INITIALIZE PDT ENTRY
          PNAME(PROC)=0                     ;IF IT IS INACTIVE
          STTN(PROC)=0
          PLIST(PROC)=0
          STATE(PROC)=0
          MSG(1,PROC)=0
      .FI
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine ALLOC

```
      SUBROUTINE ALLOC
/*      THE ALLOCATE THE FILE ROUTINE
        THE PROCESS'S PDT ENTRY IS UPDATED TO SHOW THAT THIS FILE IS
        OWNED. THE FILE'S FDT ENTRY IS UPDATED TO SHOW IT IS OWNED BY
        THIS PROCESS. THE FILE GRANTED MESSAGE IS SENT TO THE PROCESS.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=8
      AMASK=2**(FILE-1)                    ;INSERT FILE AS OWNED
      OWNED(PROC)=OWNED(PROC).OR.AMASK  ;IN PDT ENTRY
      REQST(PROC)=0
      STATE(PROC)=0
      OWNER(FILE)=PROC          ;UPDATE FDT ENTRY
      MODE(FILE)=1
      RCODE=1
      CALL SEND          ;SEND FILE GRANTED MESSAGE
      CALL LOG
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine SEND

```
      SUBROUTINE SEND
/*      THE SEND MESSAGE ROUTINE
        THE MESSAGE IS COMPOSED ACCORDING TO THE MESSAGE CODE SUPPLIED
        BY THE CALLING ROUTINE - 'RCODE'. THE LINK ON WHICH THE MESSAGE
        IS SENT - 'LUN' - IS EITHER THE LINK TO THE REMOTE ACCESS
        CONTROLLER OR TO THE LOCAL PROCESS CONCERNED.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=9
      LUN=1               ;LOGICAL UNIT NUMBER OF THE COMMUNICATION LINK
      .SWITCHON RCODE
C
C                 REQUEST GRANTED
   .CASE 1
            .IF STTN(PROC) ==  LOCAL
                MSG(1,PROC)=RCODE
                MSG(2,PROC)=FNAME(FILE)
                LUN=PROC
            .ELSE
                MSG(1,PROC)=5
                MSG(2,PROC)=FILE
            .FI
            MSG(3,PROC)=PNAME(PROC)
            .ENDCASE
C
C              REQUEST REJECTED
   .CASE -3,-5
            .IF STTN(PROC) == LOCAL
                MSG(1,PROC)=RCODE
                MSG(2,PROC)=FNAME(FILE)
                LUN=PROC
            .ELSE
                MSG(1,PROC)=6
                MSG(2,PROC)=FILE
                MSG(4,PROC)=RCODE
            .FI
            MSG(3,PROC)=PNAME(PROC)
            .ENDCASE
C
C          FILE NOT FOUND OR INSUFFICIENT TABLE SPACE
   .CASE -2,-4,-6
            .IF STTN(PROC) == LOCAL
                LUN=PROC
                MSG(1,PROC)=RCODE
            .ELSE
                MSG(1,PROC)=6
                MSG(4,PROC)=RCODE
            .FI
            .ENDCASE
```

Subroutine SEND (Continued)

```
C
C           REMOTE FILE REQUEST
   .CASE 3
            MSG(1,PROC)=RCODE
            MSG(2,PROC)=FILE
            MSG(3,PROC)=PNAME(PROC)
            MSG(4,PROC)=OWNED(PROC)
            MSG(5,PROC)=PRLIST
            .ENDCASE
C
C           REMOTE FILE RELEASE OR REMOTE REQUEST GRANTED
   .CASE 4,5
            MSG(1,PROC)=RCODE
            MSG(2,PROC)=FILE
            MSG(3,PROC)=PNAME(PROC)
            .ENDCASE
C
C             PROPGATE PREDECESSOR LIST
   .CASE 7
            MSG(1,PROC)=RCODE
            MSG(2,PROC)=FILE
            MSG(3,PROC)=PRED
            MSG(4,PROC)=PRLIST
            .ENDCASE
C
C           BOSS SELECTION (DEADLOCK BREAK)
   .CASE 8
            MSG(1,PROC)=RCODE
            MSG(2,PROC)=FILE
            MSG(3,PROC)=PRED
            MSG(4,PROC)=BOSS
            MSG(5,PROC)=BPRED
            .ENDCASE
C
C           DEADLOCK RECOVERY
   .CASE 9
            MSG(1,PROC)=RCODE
            MSG(2,PROC)=FILE
            MSG(3,PROC)=BPRED
            MSG(4,PROC)=BOSS
            .ENDCASE
C
C           UPDATE IMMEDIATE SUCCESSOR POINTER
   .CASE 10
            MSG(1,PROC)=RCODE
            MSG(2,PROC)=FILE
            MSG(4,PROC)=LIST
            .ENDCASE
```

Subroutine SEND (Continued)

```
C
C         REQUEST PREDECESSOR LIST
C
  .CASE 11
          MSG(1,PROC)=RCODE
          MSG(3,PROC)=PROC
          MSG(4,PROC)=LIST
          .ENDCASE
C
C         PREDECESSOR LIST REPLY
C
  .CASE 12
          MSG(1,PROC)=RCODE
          MSG(3,PROC)=PROC
          MSG(4,PROC)=PRLIST
          .ENDCASE
C
C         START UP MESSAGE
C
  .CASE 13
          MSG(1,PROC)=RCODE
          MSG(3,PROC)=LOCAL
          .ENDCASE
C
C         CLOSE DOWN MESSAGE
C
  .CASE 14
          MSG(1,PROC)=RCODE
          MSG(3,PROC)=LOCAL
          MSG(4,PROC)=ERR
C
          .ENDCASE
      .ENDSWITCH
      CALL LOG
      CALL NTSNDW(LUN,STAT(1,PROC),LEN,MSG(1,PROC))
      .IF STAT(1,PROC).NE.1
          ERR=9          ;DECNET ERROR
          CALL LOG
      .FI
      .IF RECV .AND. STTN(PROC) == LOCAL        ;RECEIVE ON THIS LINK IF
          CALL NTRCV(PROC,STAT(1,PROC),LEN,MSG(1,PROC)) ; PROCESS LOCAL
      .FI
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine PROPGT

```
      SUBROUTINE PROPGT
/*      THE PROPAGATE PREDECESSOR LIST ROUTINE
        THE VARIABLE 'PRLIST' CONTAINS THE PROPAGATED LIST. A PATH IN
        THE GRAPH, GIVEN BY THE IMMEDIATE SUCCESSOR POINTERS, IS THE
        ROUTE OF THE PROPAGATION. THE DEADLOCK TEST IS PERFORMED AT
        EACH NODE OF THE GRAPH BEFORE THE PREDECESSOR LIST IS UPDATED.
        THE PROPAGATION ENDS IF THE I.S. IS NULL, A DEADLOCK IS
        DETECTED, OR THE FILE IS IN DEADLOCK MODE. THE PROPAGATION IS
        SENT TO THE REMOTE ACCESS CONTROLLER IF THE I.S. IS A REMOTE
        FILE.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=10
      LOCK=.FALSE.
      .UNTIL FILE ==0 .OR. HOST(FILE) ^=LOCAL .OR. LOCK
                .OR. MODE(FILE) == 2
          AMASK=2**(FILE-1)              ;DEADLOCK IS DETECTED IF THE
          AMASK=AMASK.AND.PRLIST         ;FILE IS ITS OWN PREDECESSOR
          .IF AMASK ^=0
              LOCK=.TRUE.
          .ELSE
              CALL LOG
              PL(FILE)=PL(FILE).OR.PRLIST       ;UPDATE PREDECESSOR LIST
              PRLIST=PL(FILE)
              PRED=FILE                  ;PROPAGATE P.L. TO THE
              FILE=IS(FILE)              ;NEXT NODE IN THE GRAPH
          .FI
      .REPEAT
      .IF LOCK                           ;DEADLOCK DETECTED
          PROC=0
          BOSS=0
          BPRED=0
          CALL DBREAK                    ;START BOSS SELECTION
      .ELSF FILE ^=0 .AND. HOST(FILE) ^= LOCAL
          RCODE=7                        ;SEND PROPAGATION TO THE
          CALL SEND                      ;REMOTE ACCESS CONTROLLER
      .FI
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine DBREAK

```
      SUBROUTINE DBREAK
/*      THE DEADLOCK BOSS SELECTION ROUTINE
        THE BOSS SELECTION FOLLOWS THE LOOP IN THE GRAPH. THE BOSS
        IS THE HIGHEST NUMBERED FILE IN THE LOOP ACCORDING TO THE
        GLOBAL DIRECTORY. THE SELECTION PROCESS ENDS WHEN THE NEXT
        FILE IS ALSO THE BOSS OR THE NEXT FILE IS REMOTE. IN THIS
        CASE THE BOSS SELECTION MESSAGE IS SENT TO THE REMOTE STATION.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=11
      .UNTIL BOSS == FILE .OR. HOST(FILE) != LOCAL
          MODE(FILE)=2
          .IF BOSS < FILE                ;SELECT BOSS BY HIGHEST
              BPRED=PRED                 ;POSITION IN DIRECTORY
              BOSS=FILE
          .FI
          CALL LOG
          PRED=FILE
          FILE=IS(FILE)                  ;NEXT FILE IN LOOP
      .REPEAT
      .IF BOSS == FILE
          DPROC=0
          CALL RECOVR                    ;START RECOVERY
      .ELSE
          RCODE=8                        ;SEND BOSS SELECTION MESSAGE TO
          CALL SEND                      ;THE REMOTE ACCESS CONTROLLER
      .FI
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine RECOVR

```
SUBROUTINE RECOVR
/*    THE DEADLOCK RECOVERY ROUTINE
      THE RECOVERY STARTS AT THE BOSS. THE FILES OWNED BY THE OWNER
      OF THE BOSS'S IMMEDIATE PREDECESSOR IN THE LOOP, ARE DELETED
      FROM THE BOSS'S IMMEDIATE PREDECESSOR LIST. THE RECOVERY, WHICH
      FOLLOWS THE LOOP, RECREATES THE FILE'S PREDECESSOR LIST FROM
      THOSE OF ITS IMMEDIATE PREDECESSORS. IF A REMOTE PREDECESSOR LIST
      IS REQUIRED, THE RECOVERY IS SUSPENDED UNTIL IT IS RECEIVED.
      WHEN ALL PREDECESSOR LISTS IN THE LOOP HAVE BEEN RECOVERED, THE
      REQUEST FOR THE BOSS'S FILE, WHICH FORMS PART OF THE LOOP, IS
      REJECTED IN ORDER TO BREAK THE DEADLOCK.
*/
%INSERT COM.SF
SRTN=RTN  ;SAVE CALLING ROUTINE
RTN=12
.IF DPROC == 0              ;CLOSE DOWN IF THERE ARE
    .IF DNUM > 2            ;>2 DEADLOCK RECOVERIES
        ERR=8               ;IN PROGRESS
        CALL CLDOWN
    .ELSE
        DNUM=DNUM+1
        DPROC=1
        .UNTIL DFILE(DPROC) == 0          ;SELECT 'DLOCK' ENTRY
            DPROC=DPROC+1                 ;FOR THIS RECOVERY
        .REPEAT
    .FI
.FI
.IF BOSS == FILE .AND. MODE(FILE) == 2    ;IF START OF RECOVERY
    IP(BOSS)=IP(BOSS).AND.(.NOT.OWNED(OWNER(BPRED)))
          ;DELETE FILES OWNED BY REQUESTER OF
.FI       ;THE BOSS FILE FROM BOSS'S I.P.L.
.UNTIL DSTAT(DPROC) == 2 .OR. HOST(FILE) ^= LOCAL
          .OR. MODE(FILE) ^= 2
    LIST=IP(FILE)
    CALL UNION              ;FORM UNION OF P.L.'S OF I.P. LIST
    .IF DSTAT(DPROC) == 2           ;RECOVERY SUSPENDED
        DFILE(DPROC)=FILE
        DPRED(DPROC)=BPRED
        DBOSS(DPROC)=BOSS
    .ELSE
        PL(FILE)=IP(FILE)           ;RECOVER PREDECESSOR LIST
        PL(FILE)=PL(FILE).OR.DLIST(DPROC)
        MODE(FILE)=1                ;REMOVE FILE FROM DEADLOCK MODE
        CALL LOG
        FILE=IS(FILE)               ; NEXT FILE IN LOOP
        DSTAT(DPROC)=1
    .FI
.REPEAT
```

Subroutine RECOVR (Continued)

```
.IF DSTAT(DPROC) ^= 2
    .IF HOST(FILE) ^= LOCAL
        RCODE=9             ;SEND RECOVERY MESSAGE TO THE
        CALL SEND           ;REMOTE ACCESS CONTROLLER
    .ELSE
        DPROC=OWNER(BPRED)
        RCODE=-3            ;REJECT REQUEST TO BREAK
        CALL REJECT         ;THE DEADLOCK
        CALL DELETE
    .FI
    DSTAT(DPROC)=0          ;RESET DLOCK ENTRY
    DFILE(DPROC)=0
    DNUM=DNUM-1
    BOSS=0
    BPRED=0
.FI
RTN=SRTN  ;RESTORE CALLING ROUTINE
RETURN
END
```

Subroutine UPDIS

```
      SUBROUTINE UPDIS
/*      THE UPDATE IMMEDIATE SUCCESSOR POINTER ROUTINE
        THE VARIABLE 'LIST' CONTAINS A BIT LIST OF ALL FILES WHOSE
        IMMEDIATE SUCCESSOR POINTERS ARE TO BE UPDATED TO 'FILE'.
*/
      %INSERT COM.SF
      SRTN=RTN   ;SAVE CALLING ROUTINE
      RTN=13
      .FOR I = 1 TO LFILES
          AMASK=2**(I-1+DISPL)          ;IF A LOCAL FILE IS IN THE
          AMASK=AMASK.AND.LIST          ;LIST, UPDATE ITS IMMEDIATE
          .IF AMASK .= 0                ;SUCCESSOR POINTER
              PRED=I+DISPL
              IS(PRED)=FILE
              CALL LOG
          .FI
      .REPEAT
      RTN=SRTN   ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine PLRQST

```
      SUBROUTINE PLRQST
/*      THE SATISFY THE PREDECESSOR LIST REQUEST ROUTINE
        THE VARIABLE 'LIST' CONTAINS A BIT LIST OF THOSE FILES WHOSE
        PREDECESSOR LISTS ARE REQUIRED. THE UNION OF THESE PREDECESSOR
        LISTS IS RETURNED IN 'PRLIST'. THE REQUEST CAN BE FROM THIS
        ACCESSOR CONTROLLER, ROUTINE 'UNION', OR FROM A REMOTE ACCESS
        CONTROLLER.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTTINE
      RTN=14
      PRLIST=0
      .FOR I = 1 TO LFILES
          AMASK=2**(I-1+DISPL)          ;IF A LOCAL FILE IS IN THE LIST
          AMASK=AMASK.AND.LIST          ;ITS P.L. IS REQUIRED
          .IF AMASK ^= 0
              PRLIST = PRLIST .OR. PL(I+DISPL)
          .FI
      .REPEAT
      CALL LOG
      .IF SRTN ^= 19      ;  UNION = RTN 19    ROUTINE UNION REQUIRES
          RCODE=12            ;THE PRLIST, OTHERWISE REQUIRED BY THE
          CALL SEND           ;REMOTE ACCESS CONTROLLER
      .FI
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine PLRPLY

```
      SUBROUTINE PLRPLY
/*      THE PREDECESSOR LIST REPLY RECEIVED ROUTINE
        A REPLY TO A PREDECESSOR LIST REQUEST IS RECEIVED. WHATEVER
        PROCESSING MADE THE REQUEST MAY BE RESUMED IF ALL OUTSTANDING
        REPLIES HAVE BEEN RECEIVED.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=15
      .IF PROC > 10              ;INDICATES PL REQUEST BY DEADLOCK
          DPROC=PROC-10          ;RECOVERY DLOCK INDEX
          DLIST(DPROC)=DLIST(DPROC).OR.PRLIST
          DREPS(DPROC)=DREPS(DPROC)-1
          .IF DREPS(DPROC) == 0          ;RESUME RECOVERY IF ALL
              DSTAT(DPROC)=3             REPLIES HAVE BEEN RECEIVED
              BOSS=DBOSS(DPROC)
              PRED=DPRED(DPROC)
              FILE=DFILE(DPROC)
              CALL LOG
              CALL RECOVR
          .FI
      .ELSE              ;P.L. REQUEST BY FILE RELEASE PROCESSING
          PLIST(PROC)=PLIST(PROC).OR.PRLIST
          REPLS(PROC)=REPLS(PROC)-1
          .IF REPLS(PROC) == 0
              FILE=REQST(PROC)
              STATE(PROC)=3
              CALL LOG
              CALL RELEAS                ;RESUME RELEASE PROCESSING
          .FI
      .FI
      RTN=SRTN
      RETURN
      END
```

Subroutine DEADLK

```
      SUBROUTINE DEADLK
/*      THE DEADLOCK CHECK ROUTINE
        THE UNION IS FORMED OF THE PREDECESSOR LISTS OF THE FILES OWNED
        BY THE REQUESTING PROCESS. IF THE REQUESTED FILE IS A MEMBER OF
        THIS UNION, THEN A POTENTIAL DEADLOCK EXISTS.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=16
      LOCK=.FALSE.
      SUSPEN=.FALSE.
      LIST=OWNED(PROC)          ;FORM UNION OF PL'S OF FILES OWNED
      CALL LOG
      CALL UNION                ; BY REQUESTING PROCESS
      .IF STATE(PROC) == 2
          SUSPEN=.TRUE.         ;PROCESSING SUSPENDED
      .ELSE
          AMASK=2**(FILE-1)
          AMASK=AMASK.AND.(PLIST(PROC).OR.OWNED(PROC))
          .IF AMASK ^= 0                ;IF REQUESTED FILE IN UNION
              LOCK=.TRUE.               ;POTENTIAL DEADLOCK
          .FI
      .FI
      RTN=SRTN  ;RESTORE CALLIN ROUTINE
      RETURN
      END
```

Subroutine QUEUE

```
      SUBROUTINE QUEUE
/*      THE PUT REQUEST ON THE FILE QUEUE ROUTINE
        THE REQUEST IS ENTERED ON THE FILE QUEUE. THE IMMEDIATE
        SUCCESSOR POINTERS OF THE REQUESTING PROCESS'S FILES ARE
        UPDATED TO POINT TO THE REQUESTED FILE. THIS FILE'S PREDECESSOR
        LIST IS AUGMENTED WITH THOSE OF THE REQUESTING PROCESS'S FILES.
        A PREDECESSOR LIST PROPAGATION IS STARTED.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=17
      CALL LOG
      CALL ENTER                ;ENTER REQUEST ON THE QUEUE
      REQST(PROC)=FILE
      .IF OWNED(PROC) ^= 0                ;UPDATE I.S. POINTERS OF FILES
          LIST=OWNED(PROC).AND.MASK       ;OWNED BY REQUESTING PROCESS
          CALL UPDIS
          LIST=OWNED(PROC).AND.(.NOT.MASK)
          .IF LIST ^= 0           ;IF REMOTE FILES OWNED
              RCODE=10            ;SEND I.S. UPDATE MESSAGE TO THE
              CALL SEND           ;REMOTE ACCESS CONTROLLER
          .FI
          PL(FILE)=PL(FILE).OR.PLIST(PROC).OR.OWNED(PROC)
                ;UPDATE FILE'S P.L. AND I.P.L.
          IP(FILE)=IP(FILE).OR.OWNED(PROC)
          PRLIST=PL(FILE)
          PRED=FILE
          FILE=IS(FILE)
          CALL PROPGT           ;START PREDECESSOR PROPAGATION
      .FI
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine RMREQ

```
      SUBROUTINE RMREQ
/*      THE FILE REQUEST FROM A REMOTE PROCESS ROUTINE
        THE FILE IS ALLOCATED IF IT IS FREE. THE REQUEST IS QUEUED IF
        THE PROCESS OWNES NO OTHER FILES OR THE DEADLOCK TEST PROVES
        FALSE. THE REQUEST IS REJECTED IF A POTENTIAL DEADLOCK EXISTS.
*/
      %INSERT COM.SF
      DATA FREE/0/
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=18
      CALL LOG
      .IF MODE(FILE) == FREE              ;ALLOCATE A FREE FILE
          CALL ALLOC
      .ELSE
          .IF OWNED(PROC) == 0
              CALL QUEUE                  ;QUEUE THE REQUEST
          .ELSE
              CALL DEADLK                 ;DEADLOCK CHECK
              .IF .NOT.SUSPEN
                  .IF LOCK
                      RCODE=-3            ;REJECT REQUEST DUE TO
                      CALL SEND           ;POTENTIAL DEADLOCK
                  .ELSE
                      CALL QUEUE                  ;QUEUE THE REQUEST
                  .FI
              .FI
          .FI
      .FI
      RTN=SRTN  ;RESTORE CALLING PROCESS
      RETURN
      END
```

Subroutine UNION

```
    SUBROUTINE UNION
/*    THE FORM UNION OF PREDECESSOR LISTS ROUTINE
      THE VARIABLE 'LIST' CONTAINS A BIT LIST OF THE FILES WHOSE
      PREDECESSOR LISTS ARE REQUIRED. THE UNION IS REQUESTED EITHER
      BY DEADLOCK RECOVERY PROCESSING OR BY FILE REQUEST OR RELEASE
      PROCESSING. IF THERE ARE ANY REMOTE FILES IN 'LIST', PREDECESSOR
      LIST REQUESTS ARE SENT TO THE REMOTE STATION. IF ALL REPLIES
      HAVE BEEN RECEIVED, - STATUS = 3 - THE FORMATION OF THE UNION
      MAY CONTINUE.
*/
    %INSERT COM.SF
    SRTN=RTN  ;SAVE CALLING ROUTINE
    RTN=19
    CALL PLRQST                ;GET P.L.'S OF LOCAL FILES
    .IF SRTN == 12             ;ROUTINE 12 = RECOVERY ROUTINE 'RECOVR'
        .IF DSTAT(DPROC) == 3          ;RESUME RECOVERY PROCESSING
           DLIST(DPROC)=DLIST(DPROC).OR.PRLIST
        .ELSE
            ALIST=LIST.AND.(.NOT.MASK)
            .IF ALIST == 0
                DLIST(DPROC)=DLIST(DPROC).OR.PRLIST
            .ELSE
                LIST=ALIST             ;REMOTE FILES IN LIST
                DREPS(DPROC)=DREPS(DPROC)+1
                DSTAT(DPROC)=2                  ;SUSPEND PROCESSING
                DPROC=DPROC+10
                RCODE=11                        ;SEND P.L. REQUESTS
                CALL SEND                       ; TO REMOTE STATION
            .FI
        .FI
```

Subroutine UNION (Continued)

```
.ELSE SRTN == 22            ;ROUTINE 'ALLOCF' = SRTN 22
    .IF STATE(PROC)==3      ;RESUME FILE RELEASE PROCESSING
        PLIST(PROC)=PLIST(PROC).OR.PRLIST
    .ELSE
        ALIST=LIST.AND.(.NOT.MASK)
        .IF ALIST==0
            PLIST(PROC)=PLIST(PROC).OR.PRLIST
            STATE(PROC)=3
        .ELSE
            LIST=ALIST
            REPLS(PROC)=REPLS(PROC)+1
            STATE(PROC)=2
            RCODE=11                ;SEND P.L. REQUEST
            CALL SEND               ;TO REMOTE STATION
        .FI
    .FI
.ELSE                       ;FILE REQUEST PROCESSING
    PLIST(PROC)=PLIST(PROC).OR.PRLIST
    STATE(PROC)=3
.FI
CALL LOG
RTN=SRTN
RETURN
END
```

Subroutine ENTER

```
      SUBROUTINE ENTER
/*      THE ENTER REQUEST ON THE FILE QUEUE ROUTINE
        THE VARIABLE 'WAIT' POINTS TO THE HEAD OF THE QUEUE FOR A FILE.
        THE TAIL OF THE QUEUE IS FOUND AND THE REQUEST IS ENTERED THERE.
        THE QUEUE IS A LINKED LIST OF PROCESS DESCRIPTORS OF REQUESTING
        PROCESSES.
*/
      %INSERT COM.SF
      SRTN=RTN   ;SAVE CALLING ROUTINE
      RTN=20
      .IF WAIT(FILE) == 0                    ;IS QUEUE EMPTY?
          WAIT(FILE)=PROC                    ;YES - ENTER REQUEST AS HEAD
      .ELSE
          POINT=WAIT(FILE)                   ;NO - FIND TAIL
          .UNTIL QUE(POINT) == 0
              POINT=QUE(POINT)
          .REPEAT
          QUE(POINT)=PROC                    ;ENTER REQUEST
      .FI
      QUE(PROC)=0              ;RESET TAIL
      CALL LOG
      RTN=SRTN   ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine DELETE

```
      SUBROUTINE DELETE
/*      THE DELETE A REQUEST FROM A FILE QUEUE ROUTINE
        IF THE REQUEST IS AT THE HEAD OF THE QUEUE, THE POINTER TO THE
        HEAD IS RESET TO POINT TO THE NEXT IN THE QUEUE. OTHERWISE,
        WHEN THE REQUEST IS FOUND , THE POINTER TO IT IS RESET TO POINT
        TO THE NEXT ENTRY IN THE QUEUE.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=21
      .IF WAIT(FILE) == PROC                ; IS REQUEST AT HEAD  OF QUEUE?
         WAIT(FILE)=QUE(PROC)               ;YES - RESET HEAD POINTER
      .ELSE
         POINT=WAIT(FILE)                   ;NO - FIND REQUEST
         .UNTIL QUE(POINT) == PROC
             POINT=QUE(PROC)
         .REPEAT
         QUE(POINT)=QUE(PROC)               ;RESET POINTER TO NEXT IN QUEUE
      .FI
      QUE(PROC)=0
      CALL LOG
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine ALLOCF

```
      SUBROUTINE ALLOCF
/*    THE UPDATE FDT ENTRY DUE TO FILE ALLOCATION ROUTINE
      THE FILE IS ALLOCATED TO THE REQUEST AT THE HEAD OF THE QUEUE.
      THE FILE'S PREDECESSOR LIST AND IMMEDIATE PREDESESSOR LIST ARE
      UPDATED ACCORDINGLY, - NULL, IF THERE ARE NO MORE REQUESTS ON
      THE QUEUE. IF THERE ARE, THE PREDECESSOR LISTS OF THE FILES
      OWNED BY THE REQUESTING PROCESS AND THE OWNED FILES THEMSELVES
      ARE REMOVED FROM THE FILE'S PREDECESSOR LIST. THE FILES OWNED
      BY THE REQUESTING PROCESS ARE REMOVED FROM THE FILE'S I.P.L.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=22
      SPROC=0
      SUSPEN=.FALSE.
      CALL LOG
      .IF WAIT(FILE) == 0                ;SET FDT ENTRY TO NULL
          PL(FILE)=0                     ;IF QUEUE IS EMPTY
          IP(FILE)=0
          OWNER(FILE)=0
          MODE(FILE)=0
      .ELSE
          SPROC=WAIT(FILE)
          .IF QUE(SPROC) == 0            ;IF ONLY ONE WAITING SPROCESS
              IP(FILE)=0                 ;RESET P.L. AND I.P.L. TO NULL
              PL(FILE)=0
              STATE(SPROC)=3
```

Subroutine ALLOCF (Continued)

```
      .ELSE
          LIST=OWNED(SPROC)
          CALL UNION                    ; FORM UNION OF FILES OWNED
          .IF STATE(SPROC)==3           ;PROCESSING NOT SUSPENDED
              SFILE=FILE                ;SAVE FILE
              FILE=0                    ; NULL FOR UPDIS ROUTINE
              CALL UPDIS                ;UDATE I.S. POINTERS TO NULL
              LIST=LIST.AND.(.NOT.MASK)     ;IF REMOTE FILES
              .IF LIST ^= 0             ;ARE OWNED
                  RCODE=10              ;SEND UPDATE I.S. MESSSAGE
                  CALL SEND             ;TO REMOTE STATION
              .FI
              FILE=SFILE                ;RESTORE FILE
              IP(FILE)=IP(FILE).AND.(.NOT.OWNED(SPROC))
              PL(FILE)=PL(FILE).AND.(.NOT.(PLIST(SPROC)
                    .OR.OWNED(SPROC)))
               DELETE FILES OWNED FROM FILE'S P.L. AND I.P.L.
            DELETE P.L.S OF FILES OWNED FROM FILE'S P.L.
          .ELSE
              SUSPEN=.TRUE.             ;SUSPEND RELEASE PROCESSING
          .FI
      .FI
  .FI
  RTN=SRTN  ;RESTORE CALLING ROUTINE
  RETURN
  END
```

Subroutine REMRQ

```
      SUBROUTINE REMRQ
/*      THE INITIAL REMOTE REQUEST PROCESSING ROUTINE
        THE PDT IS SEARCHED FOR AN EXISTING ENTRY FOR THE REMOTE
        PROCESS. IF NONE IS FOUND, AN INACTIVE ENTRY IS SOUGHT.
        THE PDT ENTRY IS UPDATED FROM THE INFORMATION IN THE MESSAGE
        AND THE ACCESS CONTROLLER'S VARIABLES ARE SET.
*/
      %INSERT COM.SF
      LOGICAL*1 STOP
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=23
      REM=PROC          ;SAVE THE MESSAGE BUFFER INDEX
      PROC=START-1
      STOP=.FALSE.
      .UNTIL STOP.OR. PROC == LENGTH          ;LOCATE ENTRY IN PDT
          PROC=PROC+1                         ; FOR THIS PROCESS
          .IF STTN(PROC) == REMOTE.AND.PNAME(PROC) == MSG(3,REM)
              STOP=.TRUE.
          .FI
      .REPEAT
      .IF .NOT.STOP
          PROC=0                ;FIND AN INACTIVE ENTRY
          CALL INACT
      .FI
      PNAME(PROC)=MSG(3,REM)              ;SET PDT ENTRY AND VARIABLES
      STTN(PROC)=REMOTE                   ;ACCORDING TO INFORMATION
      OWNED(PROC)=MSG(4,REM)              ;IN THE MESSAGE
      REQST(PROC)=MSG(2,REM)
      STATE(PROC)=1
      PLIST(PROC)=MSG(5,REM)
      FILE=REQST(PROC)
      CALL RMREQ                  ;START FILE REQUEST PROCESSING
      CALL LOG
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine RMGRN

```
      SUBROUTINE RMGRN
/*      THE REMOTE REQUEST GRANTED ROUTINE
        THE PDT ENTRY IS UPDATED TO SHOW THAT A REQUEST FOR A
        REMOTE FILE HAS BEEN GRANTED. THE PROCESS IS NOTIFIED WITH
        A REQUESTED GRANTED MESSAGE.
*/
      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=25
      AMASK=2**(FILE-1)
      OWNED(PROC)=OWNED(PROC).OR.AMASK          ;UPDATE PDT ENTRY
      REQST(PROC)=0
      CALL LOG
      RCODE=1             ;SEND REQUEST GRANTED MESSAGE
      CALL SEND           ;TO THE PROCESS
      RECV=.FALSE.
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine RQSEND

```
      SUBROUTINE RQSEND
     /*
       THE SEND A REMOTE FILE REQUEST ROUTINE
       THE UNION OF THE PREDECESSOR LISTS OF THE LOCAL FILES
       OWNED BY THE REQUESTING PROCESS IS SENT AS PART OF THE
       REMOTE REQUEST MESSAGE.
     */

      %INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=26
      PRLIST=0
      .FOR I=1 TO LFILES
          AMASK=2**(I+DISPL-1)                    ;FORM UNION OF P.L.S OF
          .IF (OWNED(PROC).AND.AMASK) ^= 0        ;LOCAL FILES OWNED BY
              PRLIST=PRLIST.OR.PL(I+DISPL)        ;THE REQUESTING PROCESS
          .FI
      .REPEAT
      REQST(PROC)=FILE          ;UPDATE THE PDT ENTRY
      RCODE=3
      CALL LOG
      CALL SEND          ;SEND REMOTE REQUEST TO REMOTE STATION
      RTN=SRTN  ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine CLDOWN

```
      SUBROUTINE CLDOWN
/*    THE CLOSE DOWN ROUTINE
        A CLOSE DOWN MESSAGE IS SENT TO THE REMOTE ACCESS CONTROLLER
        IF THE CLOSE DOWN WAS LOCALLY INITIATED.
*/
      $INSERT COM.SF
      SRTN=RTN  ;SAVE CALLING ROUTINE
      RTN=27
      .IF (SRTN == 1 .AND. STTN(PROC) == LOCAL) .OR. SRTN == 12
COMMENT       SRTN 12 IS ROUTINE RECOVR
COMMENT       SRTN  1 IS ROUTINE EXEC
          RCODE=14.
          CALL SEND              ;SEND CLOSE DOWN MESSAGE
      .FI
      .IF ERR ^= 0
          WRITE(12,.LF) LOCAL,ERR
          (X,A2,' CLOSING DOWN DUE TO ERROR ',I2)
      .FI
      CALL LOG
      WRITE(12,.LF) LOCAL
      (' ',A2,' CLOSING DOWN')
      .IF LOGON              ;CLOSE THE LOG IF IT WAS OPENED
          CALL CLOSE(11)
      .FI
      STOP                   ;STOP PROCESSING
      END
```

Subroutine BEGIN

```
      SUBROUTINE BEGIN
/*      THE START UP ROUTINE
        FOR LOG USE ONLY
*/
      %INSERT COM.SF
      SRTN=RTN   ;SAVE CALLING ROUTINE
      RTN=28
      CALL LOG
      RTN=SRTN   ;RESTORE CALLING ROUTINE
      RETURN
      END
```

Subroutine LOG

```
      SUBROUTINE LOG
C       THIS ROUTINE LOGS THE MESSAGES, EVENTS AND TABLES AS REQUIRED
C       BY THE ACCESS CONTROLLER. THE LOG FILE NAME IS INITIALISED
C       IN THE INIT ROUTINE.
C
      %INSERT COM.SF
      COMMON /DIS/ DSLIST(10,2)
      COMMON /LOGS/LOGON,MSGS,EVNS,TABS
      LOGICAL*1 LOGON,MSGS,EVNS,TABS
      INTEGER NAME(3,30)
      DATA NAME/'EX','EC','  ','IN','IT','  ','MW','AI','T ',
     1  'IN','AC','T ','RE','JE','CT','FL','RE','Q ',
     2  'RE','LE','AS','AL','LO','C ','SE','ND','  ',
     3  'PR','OP','GT','DB','RE','AK','RE','CO','VR',
     4  'UP','DI','S ','PL','RQ','ST','PL','RP','LY',
     5  'DE','AD','LK','QU','EU','E ','RM','RE','Q ',
     6  'UN','IO','N ','EN','TE','R ','DE','LE','TE',
     7  'AL','LO','CF','RE','MR','Q ','RM','RE','L ',
     8  'RM','GR','N ','RQ','SE','ND','CL','DO','WN',
     9  'BE','GI','N ','  ','  ','  ','  ','  ','  '/
      .IF LOGON
         WRITE(11,.LF) RTN,(NAME(I,RTN),I=1,3)
         (' ',I2,X,3A2)
         .IF EVNS
            .SWITCHON RTN
         .CASE 1
                  .IF ERR == 4
                     WRITE(11,.LF) MSG(3,PROC)
                     (9X,'UNRECOGNIZED PROCESS NAME - ',A2)
                  .FI
                  .ENDCASE
         .CASE 2
                  WRITE(11,.LF) LOCAL,LFILES
                  (9X,' INITIALIZATION OF ',A2,' , LFILES=',I2)
                  .FOR J=1 TO 10
                     WRITE(11,.LF) FNAME(J),HOST(J)
                     (9X,2(A2,X))
                  .REPEAT
                  .ENDCASE
         .CASE 3
                  WRITE(11,.LF) ERR,RCODE,PROC
                  (9X,3(X,I2),2(X,A2))
                  .ENDCASE
         .CASE 4
                  WRITE(11,.LF) PROC
                  (9X,X,I2)
                  .ENDCASE
         .CASE 5
                  WRITE(11,.LF) FILE,PROC,RCODE
                  (9X,3(X,I2))
                  .ENDCASE
```

Subroutine LOG (Continued)

```
      .CASE 6
            WRITE(11,.LF) FILE,PROC,MODE(FILE),HOST(FILE),STTN(PROC)
            (9X,3(X,I2),2(X,A2))
            .ENDCASE
      .CASE 7
            CALL DSPLST(OWNED(PROC),0)
            WRITE(11,.LF) FILE,PROC,STTN(PROC),(DSLIST(I,1),I=1,10)
            (9X,2(X,I2),X,A2,X,10I1)
            .ENDCASE
      .CASE 8
            CALL DSPLST(OWNED(PROC),PL(FILE))
            WRITE(11,.LF) FILE,MODE(FILE),OWNER(FILE),PROC,
     1        ((DSLIST(I,J),I=1,10),J=1,2)
            (9X,4(X,I2),2(X,10I1))
            .ENDCASE
      .CASE 10
            CALL DSPLST(PRLIST,0)
            WRITE(11,.LF) FILE,PRED,(DSLIST(I,1),I=1,10)
            (9X,2(X,I2),X,10I1)
            .ENDCASE
      .CASE 11
            WRITE(11,.LF) FILE,PRED,BOSS,BPRED
            (9X,4(X,I2))
            .ENDCASE
      .CASE 12
            CALL DSPLST(PL(FILE),0)
            WRITE(11,.LF) FILE,IS(FILE),MODE(IS(FILE)),
     1        (DSLIST(I,1),I=1,10)
            (9X,3(X,I2),X,10I1)
            .ENDCASE
      .CASE 13
            WRITE(11,.LF) FILE,PRED
            (9X,2(X,I2))
            .ENDCASE
      .CASE 14
            CALL DSPLST(LIST,PRLIST)
            WRITE(11,.LF) ((DSLIST(I,J),I=1,10),J=1,2)
            (9X,2(X,10I1))
            .ENDCASE
      .CASE 15,20,21
            WRITE(11,.LF) FILE,PROC
            (9X,2(X,I2))
            .ENDCASE
      .CASE 16
            CALL DSPLST(OWNED(PROC),0)
            WRITE(11,.LF) FILE,PROC,(DSLIST(I,1),I=1,10)
            (9X,2(X,I2),X,10I1)
            .ENDCASE
```

Subroutine LOG (Continued)

```
.CASE 17
        WRITE(11,.LF) FILE,PROC,WAIT(FILE)
        (9X,3(X,I2))
        .ENDCASE
.CASE 18
        WRITE(11,.LF) FILE,PROC,MODE(FILE),STTN(PROC)
        (9X,3(X,I2),X,A2)
        .ENDCASE
.CASE 19
        .IF SRTN  == 12         ; RECOVR = 12
            CALL DSPLST(DLIST(DPROC-10),LIST)
            WRITE(11,.LF) DPROC-10,((DSLIST(I,J),I=1,10),J=1,2)
            (9X,I2,2(X,10I1))
        .ELSE
            CALL DSPLST(PLIST(PROC),LIST)
            WRITE(11,.LF) PROC,((DSLIST(I,J),I=1,10),J=1,2)
            (9X,I2,2(X,10I1))
        .FI
        .ENDCASE
.CASE 22
        CALL DSPLST(PL(FILE),0)
        WRITE(11,.LF) FILE,WAIT(FILE),(DSLIST(I,1),I=1,10)
        (9X,2(X,I2),X,10I1)
        .ENDCASE
.CASE 23,24,25
        CALL DSPLST(OWNED(PROC),0)
        WRITE(11,.LF) FILE,PROC,(DSLIST(I,1),I=1,10),STTN(PROC)
        (9X,2(X,I2),X,10I1,X,A2)
        .ENDCASE
.CASE 26
        CALL DSPLST(OWNED(PROC),PRLIST)
        WRITE(11,.LF) FILE,PROC,((DSLIST(I,J),I=1,10),J=1,2)
        (9X,2(X,I2),2(X,10I1))
        .ENDCASE
.CASE 27
        .IF ERR == 9
            WRITE(11,.LF) PROC,STAT(1,PROC)
            (9X,I2,' DISCONNECT ERROR',I5)
        .ELSE
            WRITE(11,.LF) LOCAL,ERR
            (9X,' CLOSE DOWN OF ',A2,' ERROR =',I2)
        .FI
        .ENDCASE
.CASE 28
        WRITE(11,.LF) TASK(1)
        (9X,A2)
        .ENDCASE
    .ENDSWITCH
.FI
```

Subroutine LOG (Continued)

```
    .IF MSGS .AND. (RTN == 3 .OR. RTN == 9)
        .SWITCHON ERR
    .CASE 6
            CALL DSPLST(OWNED(PROC),0)
            WRITE(11,.LF) PNAME(PROC),(DSLIST(I,1),I=1,10)
            (9X,A2,' DISCONNECTED ',10I1)
            .ENDCASE
    .CASE 9
            WRITE(11,.LF) PNAME(PROC),STAT(1,PROC)
            (9X,A2,' DECNET ERROR ',I5)
            .ENDCASE
    .CASE 7
            WRITE(11,.LF) PNAME(PROC)
            (9X,' RECEIVE FROM ',A2)
            .ENDCASE
    .DEFAULT
            WRITE(11,.LF) (MSG(I,PROC),I=1,5),MSG(2,PROC),MSG(3,PROC)
            (9X,5(X,I5),2(X,A2))
            .ENDCASE
        .ENDSWITCH
    .FI
.FI
.IF TABS .AND. ERR == 10
    .FOR I=1 TO 10
        CALL DSPLST(OWNED(I),PL(I))
        WRITE(11,.LF) I,PNAME(I),STTN(I),REQST(I),(DSLIST(J,1),J=1,10),
1         FNAME(I),HOST(I),OWNER(I),MODE(I),WAIT(I),IS(I),(DSLIST(J,2),J=1,10)
        (X,I2,2(X,A2),X,I2,X,10I1,10X,2(A2,X),4(I2,X),10I1)
    .REPEAT
.FI
ERR=0
RETURN
END
```

Subroutine DSPLST

```
      SUBROUTINE DSPLST (LIST1,LIST2)
/*      THE DSIPLAY BIT LIST ROUTINE
        THIS ROUTINE IS CALLED BY THE LOG ROUTINE TO COVERT A BIT
        LIST INTO A CHARACTER LIST OF CORRESPONDING ONES AND ZEROS.
        E.G.    LIST=23.        DISPLAY=1110100000
*/
      IMPLICIT INTEGER (A-Z)
      COMMON/DIS/ DSLIST(10,2)
      .FOR I=1 TO 10
          AMASK=2**(I-1)
          A1=AMASK.AND.LIST1
          .IF A1 == 0
              DSLIST(I,1)=0
          .ELSE
              DSLIST(I,1)=1
          .FI
          A2=AMASK.AND.LIST2
          .IF A2 == 0
              DSLIST(I,2)=0
          .ELSE
              DSLIST(I,2)=1
          .FI
      .REPEAT
      RETURN
      END
```

Subroutine PROC

```
      IMPLICIT INTEGER (A-Z)
/*      PROC.SF
        THIS IS THE USER PROCESS.
        THIS PROGRAM REQUESTS A PROCESS NAME FROM THE USER.
        IT THEN REQUESTS FILE NAMES FROM THE USER.
         FOR EACH FILE NAME A FILE ACCESS REQUEST
        IS SENT TO THE ACCESS CONTROLLER. IF A NULL FILE NAME
        IS ENTERED BY THE USER, THE PROGRAM RELEASES ITS FILES
        AND TERMINATES.
*/
      INTEGER FILE(4)
      DATA BLANK/'  '/
      RCODE=0
      NUM=0
      I=0
      WRITE(5,.LF)
      ('$PROCESS NAME ? :')
      READ(5;.LF) PNAME
      (A2)
      .DO
          WRITE(5,.LF) PNAME
          ('$',A2,' - FILE NAME ? :')
          READ (5,.LF) FILEX
          (A2)
          .IF FILEX == BLANK                ; NULL ENTERED BY USER
              .BREAK
          .FI
          CALL GETF (FILEX,PNAME,RCODE)          ;FILE ACCESS REQUEST
          .IF RCODE == 1                    ;FILE REQUEST GRANTED
              I=I+1
              FILE(I)=FILEX
              WRITE(5,.LF) PNAME,FILEX
              (X,A2,' GRANTED ',A2)
          .ELSE
              WRITE(5,.LF) PNAME,FILEX,RCODE
              (X,A2,' HAS ',A2,' REJECTED,  ERROR=',I2)
          .FI
          PAUSE
      .REPEAT.
```

Subroutine PROC (Continued)

```
NUM=I
I=0
.UNTIL I == NUM                          ; RELEASE FILES
    I=I+1
    CALL RELF(FILE(I),PNAME,RCODE)              ; FILE RELEASE
    .IF RCODE == 1                          ;RELEASE SUCCESSFUL
        WRITE(5,.LF) PNAME,FILE(I)
        (X,A2,' RELEASED ',A2)
    .ELSE
        WRITE(5,.LF) PNAME,FILE(I),RCODE
        (X,A2,' - FILE ',A2,' RELEASE ERROR ',I2)
    .FI
.REPEAT
PAUSE
STOP
END
```

Subroutine GETF

```
      SUBROUTINE GETF (FNAME,PNAME,RCODE)
/*      THE REQUEST FILE ACCESS ROUTINE
        THIS ROUTINE IS CALLED BY A USER PROCESS WHICH WISHES TO SEND
        A FILE ACCESS REQUEST TO THE ACCESS CONTROLLER. THE FIRST TIME
        THE ROUITNE IS CALLED, IT INITIALIZES THE COMMUNICATIONS WORK
        AREA. IF THE NUMBER OF FILES CURRENTLY OWNED BY THE PROCESS IS
        ZERO, A COMMUNICATION LINK MUST BE ESTABLISHED WITH THE ACCESS
        CONTROLLER BEFORE THE REQUEST IS SENT. ONCE THE REQUEST IS SENT
        THE ROUTINE SUSPENDS ITSELF BY WAITING FOR A REPLY FROM THE
        ACCESS CONTROLLER. THE REPLY CODE IS RETURNED TO THE PROCESS
        WHEN THE REPLY IS RECEIVED OR A COMMUNICATIONS ERROR OCCURS.
        THE PARAMETERS REQUIRED BY THE ROUTINE ARE:
        FNAME: THE NAME OF THE REQUESTED FILE
        PNAME: THE NAME OF THE CALLING PROCESS
        RCODE: THE REPLY CODE VARIABLE   RCODE=1 - REQUEST GRANTED
*/
      IMPLICIT INTEGER (A-Z)
      COMMON /VARS/ WORK(35),MSG(5),STATUS(2),NODE(3),FILES,TASK(3)
/*      VARS:   THE VARIABLES USED BY THIS ROUTINE
        WORK:   THE DECNET WORK AREA
        MSG:    DECNET MESSAGE BUFFER
        STATUS: STATUS OF THE ACTION ON THE COMMUNICATION LINK
        NODE:   THE NAME OF THE ACCESS CONTROLLER'S STATION
        TASK:   THE NAME OF THE ACCESS CONTROLLER
        FILES:  THE NUMBER OF FILES CURRENTLY OWNED BY THE
                CALLING PROCESS
*/
```

Subroutine GETF (Continued)

```
      LOGICAL*1 FIRST
      DATA NODE,TASK/'HO','ST','  ','L1','  ','  '/
      DATA FIRST/.TRUE./
      .IF FIRST          ;THE FIRST 'GETF' CALL BY THIS PROCESS
      FILES=0
      CALL NTINIT(STATUS,35,WORK)        ;INITIALIZE DECNET WORK AREA
      IF(STATUS(1).NE.1) GO TO 800
      FIRST=.FALSE.
      .FI
      .IF FILES == 0
      CALL NTCONW (1,STATUS,0,NODE,TASK)          ;CONNECT LINK WITH
      IF(STATUS(1).NE.1) GO TO 800                ;ACCESS CONTROLLER
      .FI
      MSG(1)=1
      MSG(2)=FNAME
      MSG(3)=PNAME
      MSG(4)=0
      MSG(5)=0
      CALL NTSNDW(1,STATUS,10,MSG)       ;SEND REQUEST MESSAGE
      IF(STATUS(1).NE.1) GO TO 800
      CALL NTRCVW(1,STATUS,10,MSG)       ;WAIT FOR AND RECEIVE REPLY
      IF (STATUS(1).NE.1) GO TO 800
      RCODE=MSG(1)                       ;REPLY CODE
      IF(MSG(1).EQ.1) FILES=FILES+1
      RETURN
800   WRITE(5,,LF) PNAME,STATUS(1)
      (X,A2,' DECNET ERROR ',I5)
      RCODE=-9                           ;DECNET ERROR
      RETURN
      END
```

Subroutine RELF

```
      SUBROUTINE RELF (FNAME,PNAME,RCODE)
/*      THE RELEASE FILE ROUTINE
        THIS ROUTINE IS CALLED BY A PROCESS WHICH WISHES TO SEND A FILE
        RELEASE MESSAGE TO THE ACCESS CONTROLLER. THE RETURN CODE
        INDICATES WHETHER THE RELEASE MESSAGE WAS SENT SUCCESSFULLY.
*/
      IMPLICIT INTEGER (A-Z)
      COMMON /VARS/ WORK(35),MSG(5),STATUS(2),NODE(3),FILES,TASK(3)
/*      VARS:   THE VARIABLES USED BY THIS ROUTINE
        WORK:   DECNET WORK AREA
        MSG:    DECNET MESSAGE BUFFER
        STATUS: THE STATUS OF THE ACTION ON THE COMMUNICATION LINK
        NODE:   THE NAME OF THE ACCESS CONTROLLER'S STATION
        TASK:   THE NAME OF THE ACCESS CONTROLLER
        FILES:  THE NUMBER OF FILES CURRENTLY OWNED BY THE
                CALLING PROCESS
*/
      MSG(1)=2
      MSG(2)=FNAME
      MSG(3)=PNAME
      MSG(4)=0
      MSG(5)=0
      CALL NTSNDW(1,STATUS,10,MSG)      ;SEND THE RELEASE MESSAGE
      IF(STATUS(1).NE.1) GO TO 800
      CALL NTRCVW(1,STATUS,10,MSG)      ;RECEIVE ACKNOWLEDGEMENT
      IF(STATUS(1).NE.1) GO TO 800
      RCODE=MSG(1)                      ;REPLY CODE
      FILES=FILES-1
      RETURN
800   WRITE(5,.LF) PNAME,STATUS(1)
      (X,A2,' DECNET ERROR ',I5)
      RCODE=-9                    ; DECNET ERROR
      RETURN
      END
```

Subroutine NTWAIT

THIS IS A DECNET ROUTINE

NTWAIT (lun,[istat])

Any task can call NTWAIT to suspend its execution. The calling task resumes execution when (1) a message is transmitted or received over a specified LUN, (2) the next message is transmitted or received by the calling task, regardless of the LUN, or (3) a request for a logical link connection is made by another task.

Arguments:

lun is either the logical unit number of a link, assigned by the calling task in a call to NTCON[W], or a variable set to 0 (zero) by the programmer before calling NTWAIT. istat is the name of the status block (a 2 -word integer array) to be examined by NTWAIT to check for completion on the associated LUN. If the argument lun is a variable set to 0, this argument must be omitted.

Subroutine NTDIS[W]

THIS IS A DECNET ROUTINE

NTDIS[W] (lun,istat,[iword],[iarray])

Once all messages have been transmitted, either task can call NTDIS[W] to disconnect the logical link.

Arguments:

lun is the logical unit number of the link, assigned by the calling task in a call to NTCON[W].

istat is the name of the status block (a 2 -word integer array) to contain the completion status on return from NTDIS[W].

iword is an integer specifying the number of words in the argument iarray. This argument must be less than or equal to four. If no user information is to be passed to the remote task, this argument can be omitted.

iarray is the name of the integer array containing user information to be passed to the destination task. If no user information is to be passed to the target task, this argument can be omitted.

W indicates that control is not returned to the calling task until the required action has been completed or an error occurs.

Subroutine NTRCV[W]

THIS IS A DECNET ROUTINE

NTRCV[W] (lun,istat,ibytes,iarray)

Arguments:

lun is the logical unit number of the link, assigned by the target task in a call to NTCON[W].
istat is the name of the status block (a 2-word integer array) to contain the completion status on return from NTRCV[W].
ibytes is an integer specifying the number of bytes to be received in the argument iarray. This argument must be greater than 0 (zero).
iarray is the name of the integer array to contain the message.
W indicates that control is not returned to the calling task until the required action has been completed or an error occurs.

Subroutine NTSND[W]

THIS IS A DECNET ROUTINE

NTSND[W] (lun,istat,ibytes,iarray)

The source task must call NTSND[W] to send a message to a target task.

Arguments:

lun is the logical unit number of the link, assigned by the source task in a call to NTCON[W].
is the name of the status block (a 2-word integer array) to contain the completion status on return from NTND[W].
ibytes is an integer specifying the number of bytes in the argument iarray to be transmitted. This argument must be greater than 0 (zero).
iarray is the name of the integer array containing the message to be transmitted.
W indicates that control is not returned to the calling task until the required action has been completed or an error occurs.

Subroutine NTINIT

THIS IS A DECNET ROUTINE

NTINIT (istat,iword,iarray)

Both the source and the target tasks must call NTINIT (once and only once before any logical links can be requested).

Arguments:

istat is the name of the status block (a 2-word integer array) to contain the completion status on return from NTINIT.

iwords is an integer indicating the number of words in the array iarray. This argument must be at least (14+21n+m). where n is the maximum number of logical links to be used by the calling task at any one time for intertask communication, and m is the largest record size to be accessed by the calling task for DECNET-11 file access. If DECNET-11 file access is used by the calling task, iword must be at least 50 (decimal).

iarray is the name of an integer array containing at least (14+21n+m) words.

Subroutine NTCON[W]

THIS IS A DECNET ROUTINE

NTCON[W] (lun,istat,icon,node,
[task],[iobj],[iuic],[iwords])

Arguments:

lun is the logical unit number (LUN) to be assigned to the logical link. This number is used after this procedure call by both the source and the target tasks to refer to this link. The LUN assigned by the source task need not be the same as that assigned by the target task.

istat is the name of the status block (a 2-word integer array) to contain the completion status on return from NTCON[W].

icon is the logical link connection number. For the initial request by the source task to request a link, this argument must be 0 (zero). To accept the link, the target task must specify the value returned in the argument icon by the NTCGT[W] subroutine.

node is a 6-character (or less) ASCII string. The source task must specify the node name of the target task. The target task must specify the node name of the specified task, received in a call to NTCGT[W].

task is a 6-character (or less) ASCII string. The source task must specify the name of the target task. The target task must specify the name of the source task, received in a call to NTCGT[W]. If the object type is a task, this argument should be omitted.

iobj is the name of a 2-word integer array containing the object type code.

iuic is the name of a 2-word integer array containing the octal UIC. The source task must specify the UIC of the target task. The target task must specify the UIC of the source task. If this argument is omitted, NTCON[W] uses the UIC of the calling task.

iwords is an integer specifying the number of words in the argument iarray. For the source task, this argument must be less than or equal to four. For the target task, this argument, if specified, must be 0 or 1. If no user information is to be passed, this argument can be omitted.

W indicates that control is not returned to the calling task until the required action has been completed or an error occurs.

APPENDIX B

Message Formats

B.1 Message Formats for Messages between Access Controllers

MESSAGE	CODE		FIELDS		
REMOTE FILE REQUEST	3	FILE	PROCESS NAME	FILES OWNED	PREDECESSOR LIST
REMOTE FILE RELEASE	4	FILE	PROCESS NAME	NOT USED	NOT USED
REMOTE REQUEST GRANTED	5	FILE	PROCESS NAME	NOT USED	NOT USED
REMOTE REQUEST REJECTED	6	FILE	PROCESS NAME	RELPY CODE	NOT USED
PREDECESSOR PROPAGATION	7	FILE	PREDECESSOR	PREDECESSOR LIST	NOT USED
BOSS SELECTION	8	FILE	PREDECESSOR	BOSS	BOSS'S PREDECESSOR
DEADLOCK RECOVERY	9	FILE	BOSS'S PREDECESSOR	BOSS	NOT USED
UPDATE I.S.	10	FILE (I.S.)	NOT USED	FILE LIST	NOT USED
PREDECESSOR LIST REQUEST	11	NOT USED	PROCESS	FILE LIST	NOT USED
PREDECESSOR LIST REPLY	12	NOT USED	PROCESS	PREDECESSOR LIST	NOT USED
START UP	13	NOT USED	ACCESS CONTROLLER	NOT USED	NOT USED
CLOSE DOWN	14	NOT USED	ACCESS CONTROLLER	ERROR CODE	NOT USED

B.2 Message Formats for Messages between user processes and the Access Controller in the same station.

MESSAGE	CODE	FIELDS			
FILE REQUEST	1	FILE NAME	PROCESS NAME	NOT USED	NOT USED
FILE RELEASE	2	FILE NAME	PROCESS NAME	NOT USED	NOT USED
FILE GRANTED	1	FILE NAME	PROCESS NAME	NOT USED	NOT USED
REQUEST REJECTED	2	FILE NAME	PROCESS NAME	REPLY CODE	NOT USED

B.3 The Fields Used in the Message Formats

FILE: An index to the global file directory. It indicates the file to which the message refers.
PROCESS: An index to the process descriptor table in the station where the predecessor list is required
FILES OWNED: A bit list describing the files owned by the requesting process
PREDECESSOR LIST: A bit list
REPLY CODE A numeric code indicating why the file request was rejected
PREDECESSOR: An index to the global file directory. This is the file whose LAC sent the message. The file is an immediate predecessor in the wait graph, of the file to which the message refers.
BOSS: The deadlock boss. A index to the global file directory. In the boss selection message, this refers to the file which, so far, has been selected boss.
BOSS's PREDECESSOR: the file which immediately precedes the boss file in the loop. An index to the global file directory.
I.S.: Immediate successor. An index to the global file directory. In the UPDATE I.S. message it refers to the file which is the immediate successor.
FILE LIST: A bit list indicating which files are to have their immediate successor pointers updated (in message type 10), or the files whose predecessor lists are requested (in message type 12).
ERROR: A numeric code indicating the reason for close down.

APPENDIX C

A Note on the Variable RECV

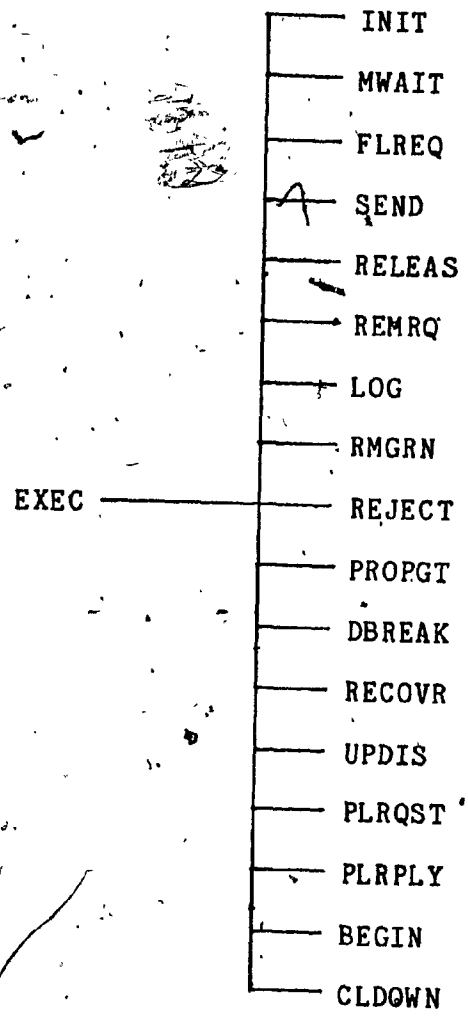
In the implementation of the Distributed Database Access Control System on the PDP/11 minicomputer network, the access controller used the variable RECV. This boolean variable indicates whether the "receive message" routine of the DECNET message switching mechanism should be invoked for a particular communication link, as a result of the last action of the access controller. The need for such a variable arises because of a flaw in DECNET.

When a message is sent on a link, any outstanding "receive" which the sender process may have had on that link, is destroyed. This means that the access controller cannot issue a "receive message" on a link with a user process, until it is sure that all the processing and sending of messages in connection with that link, have been completed. RECV is used by the access controller to remember whether a "receive message" will have to be issued on a particular link.

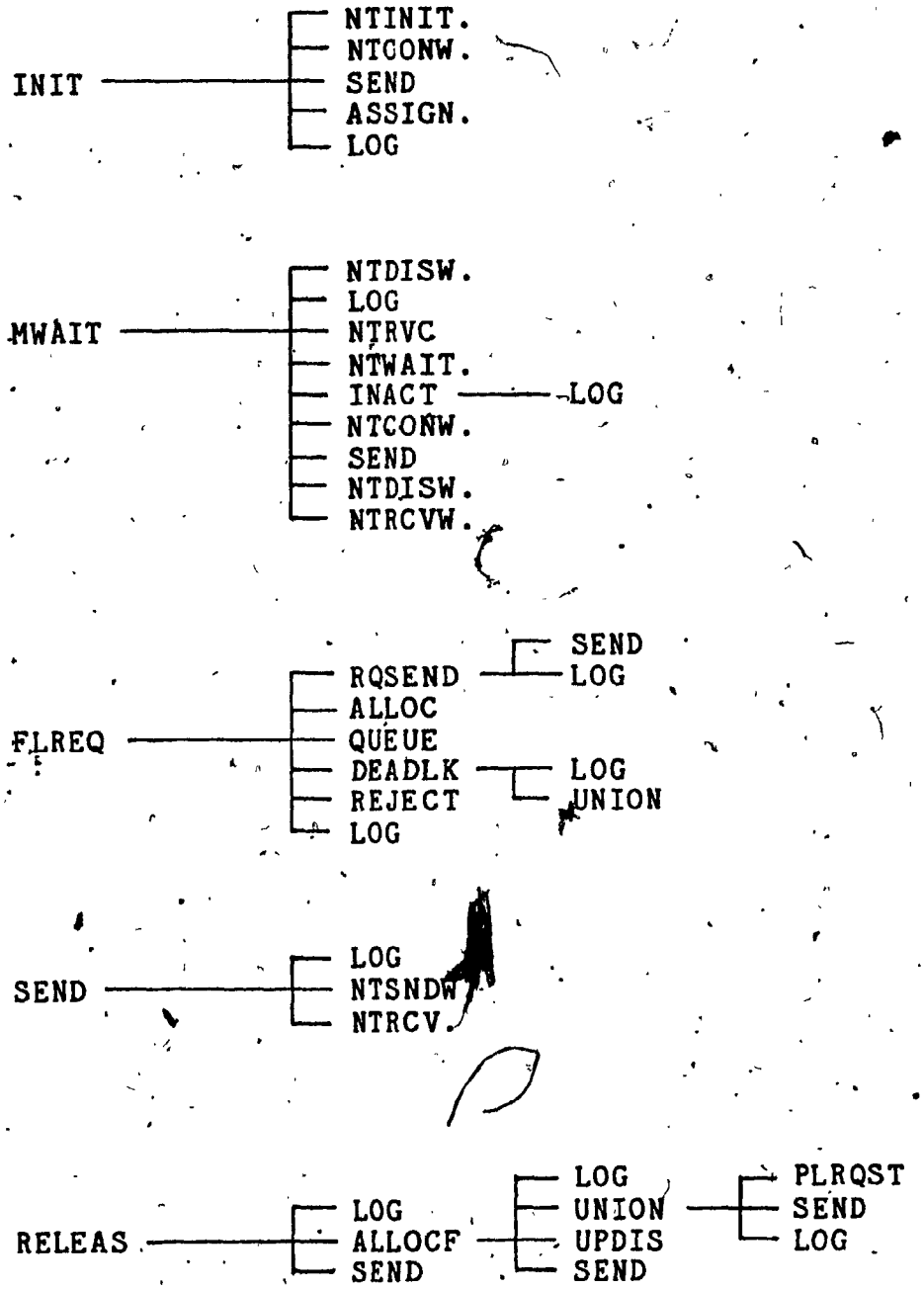
APPENDIX D

The Calling Tree in the DDACS

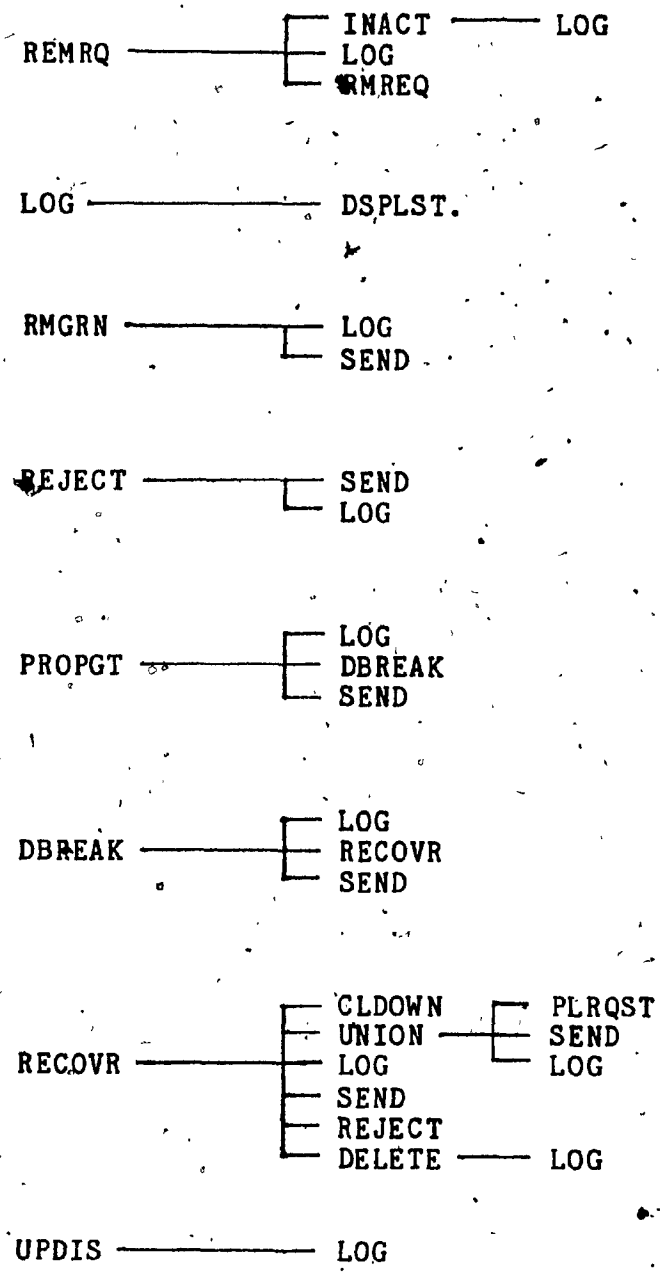
Terminal routines are denoted by a period (.).



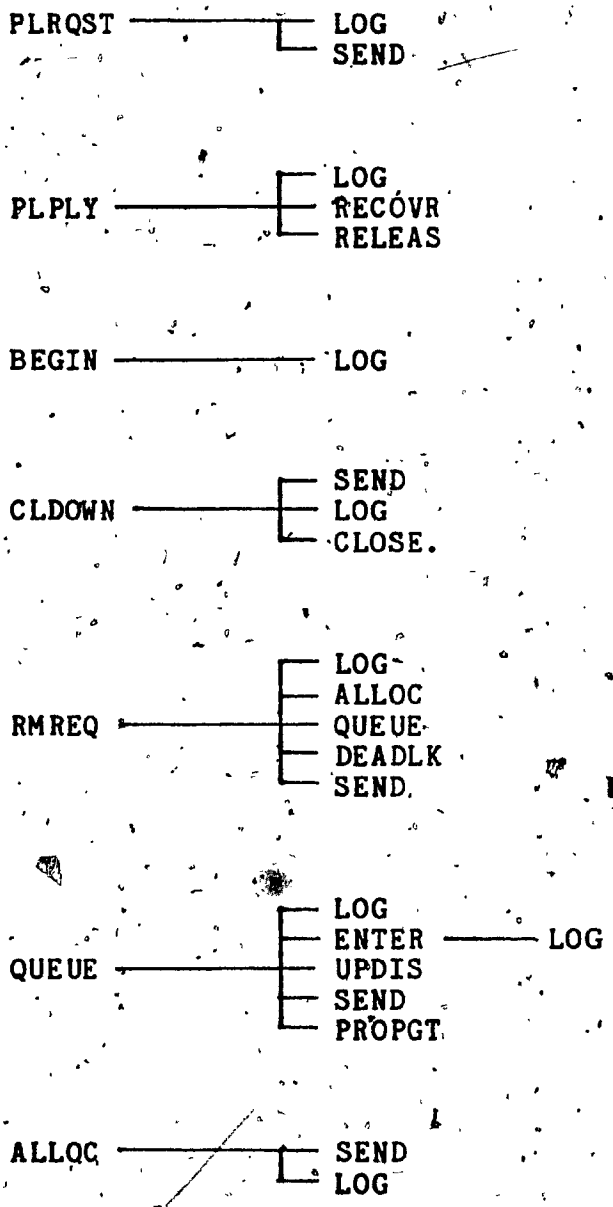
Calling Tree (Continued)



Calling Tree (Continued)



Calling Tree (Continued)



APPENDIX E

Example of the DDACS Log File

E.1 Log File for Access Controller L1

```
 2  INIT   INITIALIZATION OF L1  LOCAL FILES = 5  DISPLACEMENT = 0
    DIRECTORY OF FILES
    FILE NUMBER      1   2   3   4   5   6   7   8   9  10
    FILE NAME       F1  F2  F3  F4  F5  R1  R2  R3  R4  R5
    A.C. NAME       L1  L1  L1  L1  L1  L2  L2  L2  L2  L2

    PROCESS DESCRIPTOR TABLE          FILE DESCRIPTOR TABLE
    PROC STTN RQST    OWNED           FILE OWNER MODE I.S. WAIT   P.LIST
     1   L2   L2   0  0000011111        F1     0    0    0    0  0000000000
     2   0    0    0  0000000000        F2     0    0    0    0  0000000000
     3   0    0    0  0000000000        F3     0    0    0    0  0000000000
     4   0    0    0  0000000000        F4     0    0    0    0  0000000000
     5   0    0    0  0000000000        F5     0    0    0    0  0000000000

 4   INACT      PROC=2
 3   MWAIT      1   F1   X1   0   0              RECEIVED FROM X1-L1
 6   FLREQ      PROC=2  FILE=1
 9   SEND       1   F1   X1   0   0              SENT TO X1-L1
 8   ALLOC      PROC=2  FILE=1
 3   MWAIT                PDT                    FDT
     1   L2   L2   0  0000011111        F1     2    1    0    0  0000000000
     2   X1   L1   0  1000000000        F2     0    0    0    0  0000000000
     3   0    0    0  0000000000        F3     0    0    0    0  0000000000
     4   0    0    0  0000000000        F4     0    0    0    0  0000000000
     5   0    0    0  0000000000        F5     0    0    0    0  0000000000

 3   MWAIT      1   R1   X1   0   0              RECEIVED FROM X1-L1
 6   FLREQ      PROC=2  FILE=6
26   RQSEND     PROC=2  FILE=6  OWNED=1000000000  PL=0000000000
 9   SEND       3   6    X1   1   0              SENT TO L2
 3   MWAIT                PDT                    FDT
     1   L2   L2   0  0000011111        F1     2    1    0    0  0000000000
     2   X1   L1   6  1000000000        F2     0    0    0    0  0000000000
     3   0    0    0  0000000000        F3     0    0    0    0  0000000000
     4   0    0    0  0000000000        F4     0    0    0    0  0000000000
     5   0    0    0  0000000000        F5     0    0    0    0  0000000000

 3   MWAIT     10   6    0   1   0              RECEIVED FROM L2
13   UPDIS      FILE=1  I.S.=6
 3   MWAIT                PDT                    FDT
     1   L2   L2   0  0000011111        F1     2    1    6    0  0000000000
     2   X1   L1   6  1000000000        F2     0    0    0    0  0000000000
     3   0    0    0  0000000000        F3     0    0    0    0  0000000000
     4   0    0    0  0000000000        F4     0    0    0    0  0000000000
     5   0    0    0  0000000000        F5     0    0    0    0  0000000000
```

Log File L1 (Continued)

```
 4   INACT      PROC=3
 3   MWAIT      1   F2    X2    0    0              RECEIVED FROM X2-L1
 6   FLREQ      PROC=3   FILE=2
 9   SEND       1   F2    X2    0    0              SENT TO X2-L1
 8   ALLOC      PROC=3   FILE=2
 3   MWAIT                PDT                      FDT
  1  L2   L2    0   0000011111          F1    2     1    6    0    0000000000
  2  X1   L1    6   1000000000          F2    3     1    0    0    0000000000
  3  X2   L1    0   0100000000          F3    0     0    0    0    0000000000
  4   0    0    0   0000000000          F4    0     0    0    0    0000000000
  5   0    0    0   0000000000          F5    0     0    0    0    0000000000

 3   MWAIT      1   F1    X2    0    0              RECEIVED FROM X2-L1
 6   FLREQ      PROC=3   FILE=1
16   DEADLK     PROC=3   FILE=1   OWNED=0100000000
14   PLRQST     LIST=0100000000     PL=0000000000
19   UNION      LIST=0100000000     PL=0000000000
17   QUEUE      PROC=3   FILE=1   WAIT=0
20   ENTER      PROC=3   FILE=1
13   UPDIS      FILE=2   I.S.=1
 9   SEND       7   6     1    2    0              SENT TO L2
 3   MWAIT                PDT                      FDT
  1  L2   L2    0   0000011111          F1    2     1    6    3    0100000000
  2  X1   L1    6   1000000000          F2    3     1    1    0    0000000000
  3  X2   L1    1   0100000000          F3    0     0    0    0    0000000000
  4   0    0    0   0000000000          F4    0     0    0    0    0000000000
  5   0    0    0   0000000000          F5    0     0    0    0    0000000000

 3   MWAIT      5   6    X1    0    0              RECEIVED FROM L2
25   RMGRN      PROC=2   FILE=6
 9   SEND       1   R1    X1    0    0             SENT TO X1-L1
 3   MWAIT                PDT                      FDT
  1  L2   L2    0   0000011111          F1    2     1    0    3    0100000000
  2  X1   L1    0   1000010000          F2    3     1    1    0    0000000000
  3  X2   L1    1   0100000000          F3    0     0    0    0    0000000000
  4   0    0    0   0000000000          F4    0     0    0    0    0000000000
  5   0    0    0   0000000000          F5    0     0    0    0    0000000000

 3   MWAIT      2   R1    X1    0    0              RECEIVED FROM X1-L1
 7   RELEAS     PROC=2   FILE=6
 9   SEND       4   6     X1    0    0              SENT TO L2
 9   SEND       1   R1    X1    0    0              SENT TO X1-L1
 3   MWAIT                PDT                      FDT
  1  L2   L2    0   0000011111          F1    2     1    0    3    0100000000
  2  X1   L1    0   1000000000          F2    3     1    1    0    0000000000
  3  X2   L1    1   0100000000          F3    0     0    0    0    0000000000
  4   0    0    0   0000000000          F4    0     0    0    0    0000000000
  5   0    0    0   0000000000          F5    0     0    0    0    0000000000
```

Log File L1 (Continued)

```
 3   MWAIT      2   F1   X1     0     0          RECEIVED FROM X1-L1
 7   RELEAS     PROC=2  FILE=1
22   ALLOCF     FILE=1  WAIT=3
 9   SEND       1   F1   X1     0     0          SENT TO X1-L1
 9   SEND       1   F1   X2     0     0          SENT TO X2-L1
 8   ALLOC      PROC=3  FILE=1
 3   MWAIT                PDT                   FDT
  1  L2   L2    0   0000011111        F1    3    1     0     0    0000000000
  2   0    0    0   0000000000        F2    3    1     0     0    0000000000
  3  X2   L1    0   1100000000        F3    0    0     0     0    0000000000
  4   0    0    0   0000000000        F4    0    0     0     0    0000000000
  5   0    0    0   0000000000        F5    0    0     0     0    0000000000

 3   MWAIT      2   F2   X2     0     0          RECEIVED FROM X2-L1
 7   RELEAS     PROC=3  FILE=2
22   ALLOCF     FILE=2  WAIT=0
 9   SEND       1   F2   X2     0     0          SENT TO X2-L1
 3   MWAIT                PDT                   FDT
  1  L2   L2    0   0000011111        F1    3    1     0     0    0000000000
  2   0    0    0   0000000000        F2    0    0     0     0    0000000000
  3  X2   L1    0   1000000000        F3    0    0     0     0    0000000000
  4   0    0    0   0000000000        F4    0    0     0     0    0000000000
  5   0    0    0   0000000000        F5    0    0     0     0    0000000000

 3   MWAIT      2   F1   X2     0     0          RECEIVED FROM X2-L1
 7   RELEAS     PROC=3  FILE=1
22   ALLOCF     FILE=1  WAIT=0
 9   SEND       1   F1   X2     0     0          SENT TO X2-L1
 3   MWAIT                PDT                   FDT
  1  L2   L2    0   0000011111        F1    0    0     0     0    0000000000
  2   0    0    0   0000000000        F2    0    0     0     0    0000000000
  3   0    0    0   0000000000        F3    0    0     0     0    0000000000
  4   0    0    0   0000000000        F4    0    0     0     0    0000000000
  5   0    0    0   0000000000        F5    0    0     0     0    0000000000

 4   INACT      PROC=2
 3   MWAIT     14    0    0     0     0          RECEIVED FROM S1-L1
 9   SEND      14    0   L1     0     0          SENT TO L2
27   CLDOWN               CLOSE DOWN OF L1      ERROR=0
```

E.2 Log File for Access Controller L2

```
2 INIT   INITIALIZATION OF L2  LOCAL FILES = 5  DISPLACEMENT = 5
   DIRECTORY OF FILES
  FILE NUMBER      1   2   3   4   5   6   7   8   9  10
  FILE NAME        F1  F2  F3  F4  F5  R1  R2  R3  R4  R5
  A.C. NAME        L1  L1  L1  L1  L1  L2  L2  L2  L2  L2

   PROCESS DESCRIPTOR TABLE          FILE DESCRIPTOR TABLE
   PROC STTN RQST    OWNED           FILE OWNER MODE I.S. WAIT   P.LIST
 1  L1   L1   0   1111100000          R1    0     0    0    0   0000000000
 2  0    0    0   0000000000          R2    0     0    0    0   0000000000
 3  0    0    0   0000000000          R3    0     0    0    0   0000000000
 4  0    0    0   0000000000          R4    0     0    0    0   0000000000
 5  0    0    0   0000000000          R5    0     0    0    0   0000000000

 4   INACT     PROC=2
 3   MWAIT     1   R1   Y1   0    0          RECEIVED FROM Y1-L2
 6   FLREQ     PROC=2  FILE=6
 9   SEND      1   R1   Y1   0    0          SENT TO Y1-L2
 8   ALLOC     PROC=2  FILE=6
 3   MWAIT               PDT                FDT
 1  L1   L1   0   1111100000          R1    2     1    0    0   0000000000
 2  Y1   L2   0   0000010000          R2    0     0    0    0   0000000000
 3  0    0    0   0000000000          R3    0     0    0    0   0000000000
 4  0    0    0   0000000000          R4    0     0    0    0   0000000000
 5  0    0    0   0000000000          R5    0     0    0    0   0000000000

 3   MWAIT     3   6   X1   1    0          RECEIVED FROM X2-L1
 4   INACT     PROC=3
18   RMREQ     PROC=3  FILE=6
16   DEADLK    PROC=3  FILE=6  OWNED=1000000000
14   PLRQST    LIST=1000000000   PL=0000000000
19   UNION     LIST=1000000000   PL=0000000000
17   QUEUE     PROC=3  FILE=6  WAIT=0
20   ENTER     PROC=3  FILE=6
 9   SEND     10   6   0    1    0          SENT TO L1
23   REMRQ     PROC=3  FILE=6
 3   MWAIT               PDT                FDT
 1  L1   L1   0   1111100000          R1    2     1    0    3   1000000000
 2  Y1   L2   0   0000010000          R2    0     0    0    0   0000000000
 3  X1   L1   6   1000000000          R3    0     0    0    0   0000000000
 4  0    0    0   0000000000          R4    0     0    0    0   0000000000
 5  0    0    0   0000000000          R5    0     0    0    0   0000000000
```

Log File L2 (Continued)

```
 3   MWAIT    7    6     1     2    0              RECEIVED FROM L1
10   PROPGT   FILE=6  PRED=1  PLIST=0100000000
 3   MWAIT              PDT                       FDT
  1  L1   L1   0   0000011111        R1    2     1    0    3   1100000000
  2  Y1   L2   0   0000010000        R2    0     0    0    0   0000000000
  3  X1   L1   6   1000000000        R3    0     0    0    0   0000000000
  4   0    0   0   0000000000        R4    0     0    0    0   0000000000
  5   0    0   0   0000000000        R5    0     0    0    0   0000000000

 3   MWAIT    2    R1    Y1    0    0              RECEIVED FROM Y1-L2
 7   RELEAS   PROC=2  FILE=6
22   ALLOCF   FILE=6  WAIT=3
 9   SEND     1    R1    Y1    0    0              SENT TO Y1-L2
 9   SEND     5    6     X1    0    0              SENT TO L1
 8   ALLOC    PROC=3  FILE=6
 3   MWAIT              PDT                       FDT
  1  L1   L1   0   1111100000        R1    3     1    0    0   0000000000
  2   0    0   0   0000000000        R2    0     0    0    0   0000000000
  3  X1   L2   0   1000010000        R3    0     0    0    0   0000000000
  4   0    0   0   0000000000        R4    0     0    0    0   0000000000
  5   0    0   0   0000000000        R5    0     0    0    0   0000000000

 3   MWAIT    4    6     X1    0    0              RECEIVED FROM L1
 7   RELEAS   PROC=3  FILE=6
22   ALLOCF   FILE=6  WAIT=0
 3   MWAIT              PDT                       FDT
  1  L1   L1   0   1111100000        R1    0     0    0    0   0000000000
  2   0    0   0   0000000000        R2    0     0    0    0   0000000000
  3   0    0   0   0000000000        R3    0     0    0    0   0000000000
  4   0    0   0   0000000000        R4    0     0    0    0   0000000000
  5   0    0   0   0000000000        R5    0     0    0    0   0000000000

 3   MWAIT   14    0    L1    0    0              RECEIVED FROM L1
27   CLDOWN             CLOSE DOWN OF L2          ERROR=0
```

APPENDIX F

History of the Implementation of the DDACS

The Distributed Data Access Control System was implemented for a network of PDP/11 minicomputers, using DECNET V1.2 [64] as the Message Switching Mechanism. Initial testing of DECNET showed that it had all the functions necessary for our implementation. However, as noted in [66], many of the problems with DECNET are not apparent until DECNET itself or the whole computer system becomes heavily loaded.

Several problems with DECNET were uncovered during the implementation of the system. These problems were not apparent during the initial study of DECNET. The two main problems are;

(1) When a user process disconnects a communication link with the access controller, any outstanding "receive message" which the access controller may have had on other links, is destroyed.

(2) When a message is sent on a communication link, any "receive message" which the sender process had outstanding on that link, is lost.

The investigation of these problems took a considerable effort.

In July 1978, the operating system, RSX/11M, was

upgraded. The new version of the operating system did not include the DECNET mechanism. At that time, the DDACS had been fully implemented and substantially tested. However, the upgrade of the operating system precluded any further testing and experimentation.