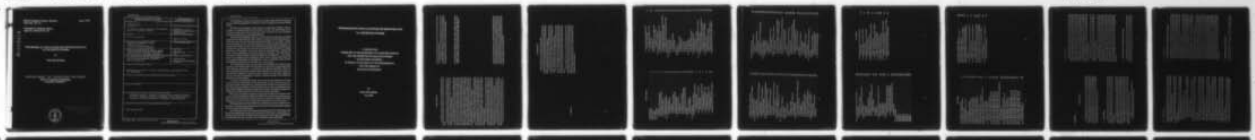
AD-A075 268 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 12/1
PERFORMANCE OF UPDATE ALGORITHMS FOR REPLICATED DATA--ETC(U)
JUN 79 H GARCIA-MOLINA MDA903-77-C-0322
UNCLASSIFIED STAN-CS-79-744 N/L

1 OF 2
AD A075268



Stanford Computer Science Laboratory
Memo CSL TR-172

June 1979

Department of Computer Science
Report No. STAN-CS-79-744

PERFORMANCE OF UPDATE ALGORITHMS FOR REPLICATED DATA IN A DISTRIBUTED DATABASE

by

Hector Garcia-Molina

DEPARTMENT OF COMPUTER SCIENCE
School of Humanities and Sciences
STANFORD UNIVERSITY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-79-744	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Performance of Update Algorithms for Replicated Data in a Distributed Database		5. TYPE OF REPORT & PERIOD COVERED technical, June 1979 6. PERFORMING ORG. REPORT NUMBER STAN-CS-79-744
7. AUTHOR(s) Hector Garcia-Molina		8. CONTRACT OR GRANT NUMBER(s) MDA903-77-C-0322
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, California 94305 USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Ave., Arlington, VA 22209		12. REPORT DATE June 1979 13. NUMBER OF PAGES 322
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Mr. Philip Surra, Resident Representative Office of Naval Research, Durand 165 Stanford University		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

centralized locking, controller, crash recovery, database consistency, distributed database, partition, performance, query, queueing theory, simulation, read-only transaction, transaction, update algorithm

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see reverse side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

In this thesis we study the performance of update algorithms for replicated data in a distributed database. In doing so, we also investigate several other related issues.

We start by presenting a simple model of a distributed database which is suitable for studying updates and concurrency control. We also develop a performance model and a set of parameters which represent the most important performance features of a distributed database.

The distributed database models are used to study the performance of update algorithms for replicated data. This is done in two steps. First the algorithms are analyzed in the case of completely replicated databases in a no failure, update only environment. Then, the restrictions that we made are eliminated one at a time, and the impact on the system performance of doing this is evaluated. For the first step, we develop a new technique for analyzing the performance of update algorithms. This iterative technique is based on queueing theory. Several well known update algorithms are analyzed using this technique. The performance results are verified through detailed simulations of the algorithms. The results show that centralized control algorithms nearly always perform better than the more popular distributed control algorithms. This is a surprising result because the distributed algorithms were thought to be more efficient.

The performance results are also useful for identifying the critical system resources. This insight leads to the development of several new update algorithms with improved performance. In particular, the MCLA-h algorithm which we present performs better than all other update algorithms in most cases of interest. The MCLA-h algorithm is based on the new concept of "hole lists" which are used to increase the parallelism of updates. Several variations of the MCLA-h algorithms are also analyzed.

In order to investigate the validity of our results in a general system, we relax the assumptions made initially in the performance studies. We show that it is possible to make a centralized control algorithm (like the MCLA-h) resilient in the face of many types of failures. We show that the cost in terms of performance for doing this is roughly the same for all algorithms and thus, the original performance comparisons are still valid in the case of crash resistant algorithms.

We analyze distributed databases with partitioned data and multiple independent control mechanisms (called controllers). We describe transaction processing in this environment and we discuss how the performance results of the fully duplicated case can be used to design update algorithms in this environment.

We demonstrate how updates that do not specify their base set initially can be processed. We present three fundamental alternatives for processing these updates and we analyze their performance.

The operation of read only transactions (queries) and their interaction with the update algorithms has not been covered in the literature. In this thesis, we classify queries into free, consistent and current queries, and we present algorithms for processing each type of query under the different update algorithms.

PERFORMANCE OF UPDATE ALGORITHMS FOR REPLICATED DATA IN A DISTRIBUTED DATABASE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Hector Garcia-Molina
June 1979

THESIS ABSTRACT

In this thesis we study the performance of update algorithms for replicated data in a distributed database. In doing so, we also investigate several other related issues.

We start by presenting a simple model of a distributed database which is suitable for studying updates and concurrency control. We also develop a performance model and a set of parameters which represent the most important performance features of a distributed database.

The distributed database models are used to study the performance of update algorithms for replicated data. This is done in two steps. First the algorithms are analyzed in the case of completely replicated databases in a no failure, update only environment. Then, the restrictions that we made are eliminated one at a time, and the impact on the system performance of doing this is evaluated. For the first step, we develop a new technique for analyzing the performance of update algorithms. This iterative technique is based on queueing theory. Several well known update algorithms are analyzed using this technique. The performance results are verified through detailed simulations of the algorithms. The results show that centralized control algorithms nearly always perform better than the more popular distributed control algorithms. This is a surprising result because the distributed algorithms were thought to be more efficient.

The performance results are also useful for identifying the critical system resources. This insight leads to the development of several new update algorithms with improved performance. In particular, the MCLA-h algorithm which we present performs better than all other update algorithms in most cases of interest. The MCLA-h algorithm is based on the new concept of "hole lists" which are used to increase the parallelism of updates. Several variations of the MCLA-h algorithms are also analyzed.

In order to investigate the validity of our results in a general system, we relax the assumptions made initially in the performance studies. We show that it is possible to make a centralized control algorithm (like the MCLA-h) resilient in the face of many types of failures. We show that the cost in terms of performance for doing this is roughly the same for all algorithms and thus, the original performance comparisons are still valid in the case of crash resistant algorithms.

We analyze distributed databases with partitioned data and multiple independent control mechanisms (called controllers). We describe transaction processing in this environment and we discuss how the performance results of the fully duplicated case can be used to design update algorithms in this environment.

We demonstrate how updates that do not specify their base set initially can be processed. We present three fundamental alternatives for processing these updates and we analyze their performance.

The operation of read only transactions (queries) and their interaction with the update algorithms has not been covered in the literature. In this thesis, we classify queries into free, consistent and current queries, and we present algorithms for processing each type of query under the different update algorithms.

KEY WORDS AND PHRASES: centralized locking, controller, crash recovery, database consistency, distributed database, partition, performance, query, queueing theory, simulation, read-only transaction, transaction, update algorithm.

This work was partially supported by a fellowship from the IBM Corporation, by a fellowship from Loving Grace Cybernetics, by the Advanced Research Projects Agency of the Department of Defense under contract MDA903-77-C-0322, by the SLAC Computation Research Group of the Stanford Linear Accelerator Center under the U.S. Energy Research and Development Administration contract No. EY-76-3C-03-0515, and by the Biotechnology Research Program of the National Institutes of Health under grant NIH RR-00785.

A mis padres.

Acknowledgments

Many individuals have helped make this thesis a reality.

I would especially like to thank my principal advisor, Gio Wiederhold, for introducing me to the area of distributed databases, as well as for his advice and patience. I also want to thank the other two members of my reading committee, Susan Owicki and Clarence Ellis, for their invaluable help.

In addition to Gio, Susan and Skip, many useful suggestions and ideas were provided by Peter Bishop, Ramez ElMasri, Jim Gray, Kjell Knutsen, Bruce Lindsay, Maurice Maybury, Tochimi Minoura, Daniel Ries, Tom Rogers, Pat Selinger, John Shoch, Reid Smith, Liba Svobodova, and others.

The research described in this thesis received financial support from various sources. Partial support was received from the Advanced Research Projects Agency under the KBMS project, from the IBM Corporation, from Loving Grace Cybernetics, from the National Institutes of Health under the SUMEX project, and from the SLAC Computation Research Group of the Stanford Linear Accelerator Center.

I am also greately indebted to a large number of friends who gave me moral support and encouragement. Without their help, this thesis would have never been completed. This list of friends (which by the way includes all the individuals mentioned so far) is too long to write here. But there are a few names that must be mentioned. Special thanks to Alicia, Voy, Laura, and Fernando for their friendship, patience ans support.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

In this chapter we give a brief overview of the area of distributed databases and we define the problem that we will address in this thesis. In section 1 we define what we mean by a distributed database. In the following two sections we list some of the potential advantages and disadvantages of these systems. In section 4 we list some of the current areas of research in the distributed database field. Then in section 5 we define the particular area that we will concentrate on in this thesis. We discuss the issues we would like to address, and we describe the approach taken by giving an outline of this thesis.

1. DEFINITIONS.

One of the most serious problems in the fast growing area of distributed databases is that there is no well defined vocabulary: the same words are given different meanings and diverse names are used for the same thing. This is especially true for the term "distributed database" itself, so in this section we will try to define it.

The first step is to define what we mean by a standard (non-distributed) database: A *database* is a collection of related data that is accessible by a computer. Usually the data is shared by several users with diverse objectives. A database must also have a set of procedures for handling the data. The operations on the data include storing, updating, searching and retrieval of the data items. The system that handles the database is called the database management system.

For our definition of distributed databases, we will try to give the most general one possible. This way we will be able to encompass all of the types of distributed databases. Our definition is the following:

> A *distributed database* is a system that allows integrated access to a collection of logically independent databases.

Notice that we did not mention the term "network of computers" in our definition. There will be a computing facility, called a node, associated with each database in the distributed database and there will be communication mechanisms between the nodes. However, a distributed database does not necessarily have to be spatially distributed, nor is it necessary to have a different computer for each database. Two or more of the databases may be physically located in a single computer. In this case, the communication mechanisms are straightforward (e.g. through shared memory). To differentiate a single non-distributed database in a single computer from a set of databases on a single machine, it is important that the set of databases be logically independent. That is, it is necessary that from a logical point of view they could as well be located on separate machines. (In the remainder of this thesis, we will still use terms like "network" or "remote site" to simplify explanations.)

By integrated access we mean that a transaction entered at any node can access data in any one of the databases. This is a minimum requirement for integrated access; in particular note that being able to update or add data at a remote node is not a requirement.

A typical example of a distributed database would be a system used by a large manufacturing company. The company has several sites and at each site there is a computer. All the computers are interconnected through a network. The databases at each of the sites might contain data on the local raw materials and finished products inventory, the planned production at the site, as well as some data on the employees that work there (e.g. name, address, shift, extra hours, etc.). The database at the company's headquarters might have data on all of the employees (e.g. name, salary, site where employed, etc.) and data on the purchase and sale orders. Typical operations with the database could be to find out the address of a given worker, to update the inventory, to shift production from one site to another, or to give employees of a certain classification (at any site) a raise.

2. ADVANTAGES OF DISTRIBUTED DATABASES.

Distributed database systems are by no means the final solution to all data management problems; they are only an alternative to the more common centralized database systems. Not all databases should be designed as distributed systems. Only by understanding the particular objectives of a given system and

by knowing the advantages offered by a distributed system, will it be possible to decide if distribution pays off. It is also important to understand the advantages and disadvantages of a distributed solution because emphasizing certain advantages will result in widely varying systems.

The potential advantages of distributed database systems are the following. We could also call this list the reasons for choosing a distributed system over a centralized one.

1) PERFORMANCE. By taking advantage of the available parallelism and of the increased compute power, we can speed up operations in the database system. Our gains can be of three types:
   a) Response times for searches can be decreased.
   b) If the data of immediate importance to the user is kept locally, then this data can be kept more up to date than the entire database since local updates can be done faster.
   c) Larger databases can be handled without degrading performance.
2) RELIABILITY. By having duplicate data at different nodes, the system will be more reliable. If one node goes down we can still access data from another node.
3) CONTROL and QUALITY of data. If the data is distributed among the users, they will have direct control of their own data while still being able to share it with other users. When a user is in charge of his own data, he will be responsible for it and will take better care of it. Therefore, the data in the system will be of higher quality.
4) SHARING of geographically distributed data. If the database already exists and is geographically distributed, then a distributed database system will interconnect the databases and allow the sharing of the data.
5) ECONOMY. If the database users are geographically distributed and if their interactions exhibit strong "locality", then it might be less expensive to do the processing locally. That is, the tele-communications costs can be higher than the tele-processing costs.
6) LOAD DISTRIBUTION. The distributed database will allow us to move programs and/or data from overloaded nodes to nodes with available capacity.
7) MODULARITY. A distributed database system can be modular and therefore easier to expand.
8) SECURITY. Distributed database systems have a potential for greater security because the databases can be kept in completely independent computers with access from other nodes in the network carefully controlled.

3. DISADVANTAGES OF A DISTRIBUTED DATABASE.

Now we will give a list of the potential disadvantages of data distribution.

1) COMPLEXITY. The main problem with distributed database systems is that they are considerably more complex than centralized systems. In addition to some of the common issues related to standard databases, there is an entire set of questions that are related to the data distribution. (These problems will be treated in section 4.) The higher complexity implies greater design costs and more sources for error.
2) HARDWARE COSTS. A good distributed database system inherently has more hardware than a centralized system. Some of the sources of extra hardware are the communication mechanisms, the replicated processors and the extra storage needed for redundant data. Although hardware prices are changing rapidly, most distributed alternatives will be more expensive.
3) LACK OF EXPERIENCE. There are currently only a few experimental and limited distributed database systems being designed. So there is none of the security implied by tested and widely used ideas.
4) LACK OF CENTRAL CONTROL. It is commonly stated that centralized control is an advantage of centralized database systems. J. Fry and E. Sibley state that centralized control "is necessary for efficient data administration" [FRY76]. However, the truth of this statement is debatable. If by "efficient" we refer to hardware efficiency (e.g. no wasted resources), then a centralized system would be advisable; but if we are talking about efficient service for the users, then a distributed system might be better. Paul G. Comba uses the following argument against centralized control [COMB75]: "A large complex enterprise does not stand still long enough for the database administrator and his staff to understand the information needs of every user and integrate them into a complete database specification. ... The only sensible way to proceed is for the users to participate directly in the specification and development of those parts of the database system that are intended to facilitate their work; and for the design/implementation process to proceed interactively."

4. CURRENT AREAS OF RESEARCH.

The field of distributed databases is a complex one where there are still a lot of unresolved issues. It is a relatively young field where some research

has been done but where much more is needed. We will now list and briefly describe the current areas of research and some of the main problems of distributed databases. Notice that these areas of research are not disjoint. (A detailed description of these areas can be found in some of the overview papers [ASCH74,GARC77,MARY77,ROTH77].)

1) PROGRAM AND DATA ALLOCATION. The problem here is to find the optimal location and the optimal number of copies of the program and data files in the distributed database. What is to be minimized are the combined storage, communication and processing costs. By choosing different sets of assumptions, several solutions of varying complexity have been obtained [CASE72,CHU69,MAHM76].

2) MAINTENANCE OF DUPLICATE COPIES. Since copies of the data may exist at different nodes, it is necessary to have algorithms that make sure that all copies are updated properly. Special mechanisms are needed to know where the duplicate copies, if any, exist. Several algorithms for different types of distributed databases have been suggested and work is in progress for proving the algorithms correct [JOHN75,ELLI77,THOM76].

3) CONCURRENCY CONTROL. In a distributed database system, several users may be attempting to read and/or update a set of data. In order to always provide users with a consistent view of the data, it is necessary to have concurrency control. This control, which can either be centralized or distributed, should include synchronization and locking mechanisms. Some work has been done defining the basic concepts [ESWA76] and analyzing the available options [GRAY77,ROSE78].

4) DEADLOCKS. Just like in any system where multiple users compete for access to a set of finite resources, in distributed database systems there is a possibility for deadlocks. There are two ways to deal with deadlocks: deadlock prevention, and deadlock detection and resolution. Both of these alternatives have been analyzed for general systems [COFF71] and in particular for distributed database systems [CHU74].

5) TRANSACTION PROCESSING. This area involves the design of algorithms that process transactions into strings of data manipulation commands. These algorithms are more complex than the usual algorithms for centralized databases. First of all, the local knowledge at the node where the transaction is processed might not be enough to understand the transaction so that help from other nodes is needed. Then there is the problem of locating the relevant data. If the data is duplicated, we must choose the copy to use. Finally, one must decide how to actually manipulate the data. There are

three options for this: transmit commands and transmit results back; move all of the necessary data to a node and work there; or a mixture of these two methods (i.e. filter data before moving). These problems are discussed in [STON77,WONG77].

6) DIRECTORY MANAGEMENT. A directory contains a description and the location of files (or relations) in the system. Directories can be global or local, distributed or centralized and they can have one or many copies. The tradeoffs involved with the different options are being analyzed [CHU75]. If the distributed database is dynamically changing, it is necessary to have mechanisms to add or delete names to the directory.

7) DATA AND PROGRAM TRANSLATION. In a non-homogeneous distributed database, it is mandatory to have translation mechanisms between the databases. Since it would be very inefficient to design an interface for every possible pair of dissimilar databases, it is necessary to design general procedures for translating data and programs. These procedures can include languages for describing the data and program formats plus definition of a common intermediate format [MERT74].

8) PRIVACY. There has been very little work done in the area of data privacy. It is necessary to design good ways of identifying users, both local and remote. It would also be nice to be able to restrict access not only by who the user is but by what his application is. For example, a user might not be allowed to access a particular employee's salary, but he may be permitted to look at the average salary of a group of employees.

9) RECOVERY. If a distributed database system is to be reliable, procedures for detecting errors and recovering from failures are required. It is important that when some database fails, the rest of the data is left in a consistent form. When a node comes up after a failure, it is indispensable to get its database up to date. Recovery can become extremely hard if failures cause the network to partition (i.e. to split up into several isolated pieces). These problems are treated in [ALSB76,GRAY77].

5. THESIS OBJECTIVE AND OUTLINE.

In this thesis we will concentrate on only one of the research areas described: the maintenance of replicated copies of data. In particular, we study the performance of update algorithms for replicated data in a distributed database. Of

course, it is impossible to isolate a certain research area, and in fact, in the thesis we also touch on the problems of concurrency control and recovery. However, we do attempt to concentrate on one single problem area as much as possible in order to reduce the complexity and length of the presentation.

In any real distributed database system, all the issues of section 4 must be considered together in order to design a complete system. Since we are not considering all issues here, we do not expect to obtain a complete system design out of this thesis. The objective of this work is simply to study and compare some of the fundamental management techniques for replicated data. It is hoped that by shedding some light on these issues we can help the designer of a complete distributed database system.

Since we will be looking at replicated data, it is important to understand why several copies of the same data may be stored at different nodes in the system. There are two main reasons for replicating data. One of the reasons for replicating data is to improve its availability. Another reason is to distribute the load by allowing transactions to read the data at different sites. The price that must be paid for the increased availability and the option of concurrent reads at different nodes is an increased cost for processing updates. Updating replicated copies of data is more expensive than updating a single copy of the data because in the replicated case updates must be performed on all copies. Furthermore, it is harder to coordinate conflicting updates when there are multiple copies to be modified than it is to coordinate the updates when there is a single copy to be updated.

In this thesis, we will not study the tradeoffs involved in replicating data. We will assume that the decision to replicate a subset of the data has been made. That is, it is either imperative that the data be available even in the face of failures, or it is expected that the number of updates to the data will be considerably smaller than the number of reads on the data. Once we decide to replicate the particular subset of the data, we need to design an algorithm for performing the updates. Here, we will address this last problem.

This thesis is divided into two main parts. In the first part (chapters 2 through 6) we make a set of assumptions that simplify the analysis of the various redundant data update algorithms. Some of these assumptions are that the database is completely replicated at each node, that all transactions are updates and that no failures occur in the system. In the second part (chapters 7 through 12) we relax the assumptions made and we study the effect of doing this on the results obtained in the first part.

Chapter 2 describes the database model we use. Some important concepts

like transaction and database consistency are also defined. (The database model is extended to include partitioned data in chapter 10.) Chapter 2 also lists the assumptions that are made in the first part of the thesis.

Chapter 3 presents the update algorithms that will be studied in this thesis. The new algorithms in this chapter (i.e., the WCLA, MCLA, MCLA-h, TWCLA, MEAS, MEAP) were actually developed after some of the other algorithms were analyzed. That is, the performance analysis of chapter 4 was useful in identifying the critical system resources. This in turn lead to the design of the new algorithms. However, in chapter 3 we present all the algorithms together in order to simplify the organization of the thesis.

The performance analysis of two of the update algorithms is described in chapter 4. The performance of the algorithms is studied through simulations as well as through a new iterative analysis technique based on queueing theory. The analysis of the other algorithms in presented in the appendices because these analyzes are similar to the ones in chapter 4. In chapter 5 we compare the results of the analysis with the simulation results. Some refinements of the analysis are also given. The fact that the simulation and analysis results agree fairly closely provides a good validation of both techniques. In chapter 6 we actually give the performance results for the algorithms. The results show that centralized control algorithms nearly always perform better than the more popular distributed control algorithms. This is a surprising result because the distributed algorithms were thought to be more efficient. In particular, the MCLA algorithm, which uses the novel concept of *hole lists* to increase parallel execution of updates, has the best performance in many cases of interest.

Chapter 7 starts the second part of the thesis by investigating the effects of the no failure restriction. We show that it is possible to make a centralized control algorithm resilient in the face of many types of failures. We show that the cost in terms of performance of doing this is roughly the same for all algorithms and thus, the original performance comparisons are still valid in the case of crash resistant algorithms. In particular, in chapter 7 we outline the basic mechanisms that are needed to make the MCLA-h algorithm resilient.

In chapter 8 we justify our decision to only study algorithms that are able to process arbitrary update transactions. We give examples of algorithms that take advantage of a particular transaction type in order to improve performance, and we discuss why it would be hard to study such algorithms.

Read only transactions (queries) are analyzed in chapter 9. We classify queries into free, consistent, and current queries, and we present algorithms for processing each type of query under the different update algorithms. In order

to understand consistent queries, we come back to the issues of consistency that were discussed in chapter 2, and we study the types of consistency provided by the update algorithms. We also discuss the performance of the query algorithms.

In chapter 10 we consider how updates that do not specify their read set initially can be processed. Some of the algorithms of chapter 3 have to be modified in this case. We present three fundamental strategies for transactions that cannot request their locks beforehand. We also study the performance of these strategies.

Up to chapter 10, we assume that the data in the system is completely replicated at each node in the system. In chapter 11 we relax this assumption by allowing partitioned data as well as multiple independent "controllers". A controller is a control mechanism that is responsible for the concurrency control of a given subset of the data. We present update algorithms for the partitioned data one controller case as well as for the partitioned data multiple controller case. We discuss how the performance results of chapters 4, 5 and 6 can be used to evaluate the performance of these algorithms. In chapter 11 we also present query algorithms for the partitioned data one controller and for the partitioned data multiple controllers cases. The performance of these query algorithms is also discussed. At the end of chapter 11, we come back to the crash recovery problem. We discuss how the crash recovery ideas of chapter 7 can be extended to the partitioned data multiple controller case.

Finally, in chapter 12, we present some conclusions and we identify some areas that need further research.

CHAPTER 2

THE DISTRIBUTED DATABASE MODEL

We start this chapter by defining a simple model for a single database and for transactions on this database. We also informally discuss the concept of database consistency for a single database. Then, in section 2, we extend these ideas to a distributed database. In section 3, we list all the assumptions that are embedded in this distributed database model. In addition, we list the other assumptions that are made in order to simplify the analysis of the update algorithms.

1. A SINGLE DATABASE MODEL.

Before we consider a distributed database, we must define a model for a single database. (Many of the ideas in this section are taken from [Eswa76].)

We view a single *database* as a collection of M shared named resources called items [Eswa76]. Each item has a name and a value associated with it. We use the notation d[i] to represent the value of item i. This is a very simple model; however, it is sufficient for studying the concurrency control and consistency issues we want to address.

An example of a very small database is given in figure 2.1. This database contains three items only. The names of these items are "deposits", "withdrawals" and "balance". The value associated with item "deposits", d["deposits"], is 100. This value refers to the total amount of deposits that have been made to a certain bank account. Similarly, the values of items "withdrawals" and "balance" represent the total withdrawals and the balance of this same bank account. When we talk about a database, it may not be convenient to use the complete names of the items as we have done in this example. Therefore, in other examples we may use the integers between 1 and M as the names of the items of an M item database. Hence, we may work with item 3 or item j (where $1 \leq j \leq M$) instead of working with item "balance".

Figures 2.1 and 2.2

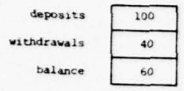


Figure 2.1. An example of a single database.

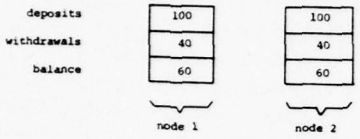


Figure 2.2. An example of a distributed database.

1.1 Consistency Constraints.

Associated with a database we have a collection of *consistency constraints* or assertions [Eswa76]. These constraints are predicates defined on the database which describe the relationships that must hold among the items of the database. For example, in the database of figure 2.1, we might have the consistency constraints "balance $>$ 0" and "deposits — withdrawals = balance". The values shown in the figure satisfy these constraints, so the database is said to be consistent.

We would like that the database always be consistent. However, due to updating activity, the consistency constraints must be temporarily violated. For example, if someone makes a 10 dollar deposit into the account described by the database of figure 2.1, we must add this amount to the "deposits" and to the "balance" items. Since these two operations cannot be performed simultaneously as a single atomic action, at some point the constraint "deposits — withdrawals = balance" will be false. But when both operations are completed, the database will be consistent again.

1.2 Transactions.

Since we are unable to guarantee consistency between actions, we group actions into *transactions* [Eswa76]. A transaction is the unit of consistency. That is, if a transaction T is run on a consistent database and without interference from other transactions, then T should leave the database consistent when it completes. A sample transaction for the database of figure 2.1 is "Deposit 10 dollars into the account". This transaction consists of several actions: (a) Read the value of item "deposits", (b) Read the value of item "balance", (c) Add 10 dollars to the value read for item "deposits", (d) Add 10 dollars to the value read for item "balance", (e) Store the new value computed for item "deposits" into the database, and (f) Store the new value computed for item "balance" into the database. Clearly, if the database is consistent to begin with, then the database will be consistent after these actions are performed (at least if they are performed without interference from other transactions.)

Since the database cannot be consistent at all times, then at least we would like that all transactions "get a consistent view of the database". By this we mean that the data read by any transaction should satisfy the consistency constraints. (More specifically, all consistency constraints that can be fully evaluated with the

data read by a transaction should be true. Consistency constraints that cannot be fully evaluated are irrelevant as far as this transaction is concerned.) Since transactions are the only entities that will ever read data from the database system, it is sufficient to guarantee that transactions will see a consistent database. For example, a user wishing to read some values out of the database will have to do so through a transaction. That user will get consistent data; the fact that the database may have been inconsistent for intervals before or after the transaction read its data is unimportant to the user.

In this thesis we view a transaction T as consisting of three steps:

(1) READ STEP. The transaction T reads the values for items $i_1, i_2, \ldots i_n$. (That is, T reads d[i_1], d[i_2], ... d[i_n].)

(2) COMPUTE STEP. Using the values obtained, T performs some arbitrary computations and comes up with a set of new values for a subset of the items read $i_1, i_2, \ldots i_m$, where $m \leq n$.

(3) WRITE STEP. The new values produced are stored in the database. (That is, T performs "d[j]:= new value for item j" for all items $j \in \{i_1, i_2, \ldots i_m\}$.)

The fact that we model transactions in this particular way has some implications that will be discussed in section 3. (Notice that we do not consider each step to be a single atomic operation.) Using the sample database of figure 2.1, the transaction T1: "Deposit 10 dollars into the account" consists of the following steps: (1) x: = d["deposits"], y: = d["balance"] (where x and y are local variables of T1); (2) x: = $x + 10$, y: = $y + 10$; and (3) d["deposits"]: = x, d["balance"]: = y.

1.3 Conflicts Among Transactions.

As we have stated, a transaction that is executed all by itself preserves database consistency. However, if several transactions are executed in parallel or concurrently, there is a possibility that consistency will be violated. By this we mean that some transactions may read inconsistent data. As we have stated earlier, we do not wish this to happen; All transactions should get a consistent view of the database. To see how transactions can interfere and cause a transaction to read inconsistent data, consider a second transaction T2: "Deposit 5 dollars into the account" that is executed at the same time as transaction T1 above. Suppose that transaction T2 is completely executed between the time T1 reads d["deposits"] and the time T1 reads d["balance"]. Thus, the value for "deposits" read by T1 will not reflect the deposit of 5 dollars made by T2, but the value

for "balance" read by T1 will reflect the 5 dollar deposit of T2. In other words, T1 reads 100 as the value for "deposits" and 65 as the value for "balance". (See figure 2.1.) Hence, T1 will store d["deposits"]: = 110 and d["balance"]: = 75, leaving the database inconsistent. (That is, 110 minus 40 is not equal to 75.)

Now, any transaction that follows can read the "deposits", the "withdrawals" and the "balance" and get an inconsistent view of the database. (As a fine point, notice that T1 sees a consistent view of the database. If T1 had also read "withdrawals", then it could have evaluated the consistency constraint "deposits — withdrawals = balance", and T1 would have observed that this consistency constraint was false. But since T1 did not read the "withdrawals" item, then it has no way of knowing that the database is inconsistent.)

1.4 Concurrency Control Mechanisms.

The problem illustrated with the previous example is a synchronization problem which appears in database management systems, in operating systems and in any system with concurrent (or parallel) programs. The solution to the problem is to have a *concurrency control mechanism* which somehow eliminates the destructive interference between transactions (or programs). The same synchronization problem, in the context of distributed databases, is the problem we will be addressing in this thesis. The solution to the synchronization problem for distributed databases is conceptually the same as for a single database. That is, we need a concurrency control mechanism for transactions in a distributed database system. But in practice, the concurrency control mechanisms for distributed databases, and in particular for replicated data, are implemented differently from the mechanisms for single databases. The difference in implementation mainly stems from the fact that in a distributed database the data is distributed and not under the control of a single computer or node.

Before we go on to describe our distributed database model in section 2, we will briefly mention two of the most common concurrency control mechanisms that are used in single database systems. One way to eliminate interference among transactions is to execute them one at a time with no parallelism. Since each transaction is executed in its totality all by itself, we guarantee that the database is always consistent after each transaction finishes. This implies that each transaction will see a consistent database. This concurrency control strategy is called *serialization* of the transactions.

The main performance disadvantage with the above strategy is precisely

that no transactions are executed in parallel. Since transactions that do not read or write any common items can in no way interfere with each other, we would like to execute such non conflicting transactions concurrently if possible. A *lock manager* is a concurrency control mechanism which detects if transactions reference (i.e., read or write) common items. Transactions that have no items in common are allowed to run concurrently by the lock manager, while conflicting transactions are delayed and serialized by the manager. The lock manager works by associating a lock with each item in the database. Before a transaction is executed, it must request locks from the manager for all items referenced by the transaction. The lock manager only grants a given lock to one transaction at a time. If a requested lock is being held by another transaction, the requesting transaction is delayed until the lock is released or returned. Thus, once a transaction obtains all requested locks, it has exclusive access to the referenced items. That is, only transactions which reference other items can be executed concurrently and there is no danger of interference. When a transaction completes, it returns the locks to the lock manager so that they may be assigned to some other transaction. More details as to how and why a lock manager works can be found in [Eswa76], [Gray77]. (These references also define formally the concepts of transaction and consistency.) Also notice that deadlocks are possible because transactions are competing for a finite set of resources (i.e., the locks). These deadlocks can be prevented or eliminated by the lock manager.

2. A DISTRIBUTED DATABASE MODEL.

In chapter 1 we defined a distributed database as a collection of databases. We will now assume that every database in the system is a complete copy of one database. In other words, there is a single database which is replicated at all nodes. This assumption is made to simplify the analysis of the update algorithms. The effect of relaxing this assumption is discussed in chapter 11.

Based on this assumption, we view a *distributed database* as a collection of M shared named resources called items. Each item has a name and a set of N values associated with it; each of these values is stored at a different node in the N node system. We represent the value of item i at node x by $d[i,x]$ $(1 \leq x \leq N)$. All the values for a given item should be the same because the database at each node should be identical. (That is, $d[i,x]$ should equal $d[i,y]$ for all nodes x,y.) However, due to the updating activity, the values may be

temporarily different. The collection of item values stored at a node is called the database (or the database copy) at that node.

Figure 2.2 gives an example of a two node distributed database. The names of the three items are "deposits", "withdrawals" and "balance". Each item has a value stored at each of the two nodes as shown in the figure. For example, d["balance", 2] = 60. As with a single database, we will sometimes use integers between 1 and M as the names of the items.

2.1 Consistency Constraints.

A distributed database also has associated with it a set of *consistency constraints.* As before, these constraints are predicates which describe the relationships that must hold among the items of the distributed database. These constraints are expressed in terms of the item names; the values stored in each node should satisfy the consistency constraints. For example, the distributed database of figure 2.2 may have the consistency constraints "balance $>$ 0" and "deposits — withdrawals = balance". In order for the distributed database to be consistent, the constraints must evaluate to true at each node. For example, d["balance", 1] and d["balance", 2] must both be greater than zero.

In addition to the consistency constraints we have described, we have an additional set of implicit constraints which state that the values of the same item should be equal. That is, $d[i,x] = d[i,y]$ for $1 \leq i \leq M$, $1 \leq x \leq N$, and $1 \leq y \leq N$.

2.2 Transactions.

Our model of a transaction is very similar to the transaction model given in section 1 for a single database. The main difference is that a transaction must now store the new values it produces at all nodes. A transaction T consists of the following steps:

(1) READ STEP. (At any node x.) The transaction T reads the values for items $i_1, i_2, \ldots i_n$ from one of the nodes. That is, T reads $d[i_1,x], d[i_2,x], \ldots d[i_n,x]$ for some node x.

(2) COMPUTE STEP. (At any node.) Using the values obtained, T performs some arbitrary computations and comes up with a set of new values for a subset of the items read, $i_1, i_2, \ldots i_m$, where $m \leq n$.

(3) WRITE STEPS. (One write step at every node.) The new values produced are stored in the distributed database. That is, T does "d[j, y]:= new value for item j" for all nodes y ($1 \leq y \leq N$), and for all items $j \in \{i_1, i_2, \ldots i_m\}$.

2.3 Conflicts Among Transactions.

We assume that a transaction that is executed without interference from other transactions, transforms a consistent distributed database into another consistent distributed database. However, if several transactions are executed concurrently, there may be interference and the consistency of the data may be violated. That is, transactions may interfere with each other and cause a transaction to get an inconsistent view of the database. Thus, just like in the single database case, we need a concurrency control mechanism that guarantees that all transactions see a consistent database.

There is one difference with the single database case. In a distributed database we have defined a specific transaction model which does not allow inter database reads. (See section 2.2.) That is, no transactions will ever read data at more than one node, and therefore, it will be impossible for any transaction to check the implicit consistency constraints. Recall that the implicit constraints state that the values of an item should be the same at all nodes. It is conceivable that we design a consistency control mechanism which guarantees that all transactions (as defined in section 2.2) see a consistent database at each node but which allows the values of a single item to be different at different nodes. This means that we have placed a second requirement on the concurrency control mechanism for distributed databases. One way to state this additional requirement is as follows: If at any point in time the system stops receiving new transactions, then all the values of a given item should converge to the same value. In other words, if no more new transactions arrive into the system, and if the system finishes processing all previous transactions, then the distributed database should be left in a state where the implicit consistency constraints are true. (This is called mutual consistency in [THOM76].)

2.4 Local Concurrency Control Mechanisms.

First let us assume that each node in the system has a *local concurrency mechanism* similar to the one described in section 1 for a single database. This

mechanism guarantees that a step of a transaction is executed as a single atomic operation at that node. (By a step we mean either the read step, the compute step or one of the write steps, as defined in section 2.2.) Thus, if a transaction T is reading data at node x (read step), T will read the data without any interference from other transactions. In other words, it will be impossible that some values and not others read by T reflect the output of some other transaction. This avoids problems like the one illustrated by the example of section 1.3. Similarly, the write step of a transaction T at node x will be performed as a single atomic operation at node x. Notice that the local concurrency control mechanisms are unable to prevent interference from other transactions between the read and write steps or between the write steps at different nodes. For example, between the time a transaction T finishes reading at node x and the time it starts writing at that same node, several other transactions may have read or written data into the database at node x.

2.5 Global Concurrency Control Mechanisms.

The local concurrency control mechanisms eliminate many of the potential conflicts between transactions. However, it is still possible that transactions interfere, even if the nodes have such local controls. We now give an example that shows how this can occur.

Assume that transaction T1: "Deposit 10 dollars into the account" is to be performed on the distributed database of figure 2.2. Suppose that this transaction arrives from a user to node 1. Node 1 executes the first two steps of T1, obtaining the new values of 110 for item "deposits" and 70 for item "balance". The write step of T1 at node 1 is executed leaving d["deposits", 1] = 110 and d["balance", 1] = 70. To perform the write step of T1 at node 2, a message is sent to node 2 instructing it to store the new values into the database at node 2. The situation at this point is illustrated in figure 2.3. In this figure, the message is shown on its way to node 2. (Notice that at this instant some of the implicit consistency constraints have been violated. For example, d["deposits", 1] is not equal to d["deposits", 2]. This does not represent a problem yet because T1 has not completed.)

Before the message arrives at node 2, a second transaction T2: "Withdraw 5 dollars from the account" arrives at node 1. This second transaction is processed in a similar way. Figure 2.4 shows the situation after T2 has been executed at node 1 and its message to node 2 is also on its way.

Figures 2.3, 2.4 and 2.5

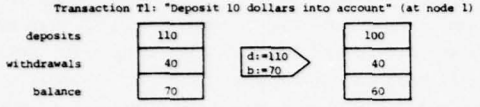


Figure 2.3. A sample transaction.

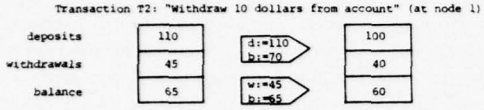


Figure 2.4. Transaction T2 is processed at node 1.

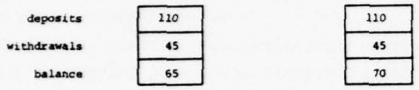


Figure 2.5. The final result.

Now suppose that the two messages arrive in the wrong order at node 2. If the commands from the second message (i.e., the withdrawal) are executed before the commands from the first message (i.e., the deposit), we obtain the system of figure 2.5. Now the database at node 2 is inconsistent and any subsequent transaction that reads the "deposits", "withdrawals" and "balance" items at node 2 will see an inconsistent database (i.e., 110 — 45 is not 70). Therefore, we see the need for a *global concurrency control mechanism* that does not permit this to happen. Notice that in figure 2.5 one of the implicit consistency constraints has been violated (i.e., d["balance", 1] is not equal to d["balance", 2]) and both transactions have completed. The global concurrency control mechanism should also eliminate this problem (even though no transactions will ever be able to detect this situation).

In this thesis we will study some of these global concurrency control mechanisms which we call update algorithms. The algorithms we study are presented in chapter 3. But before we go into that chapter, we must realize exactly what assumptions and restrictions have been embedded in the distributed database model we have defined in this section. These restrictions, as well as some other additional assumptions, will be discussed in the next section. (We will return to the issues of consistency and transactions in distributed databases in chapter 9. In that chapter we show that some of the update algorithms that will be presented in chapter 3 actually satisfy the two requirements we formulated in this section.)

3. ASSUMPTIONS.

In this section we will discuss the assumptions that are made in order to simplify the analysis of the update algorithms. In section 3.1 we list the assumptions that were implicitly made when choosing the distributed database model of section 2. Then in section 3.2, we describe a set of further assumptions we make. Later on in the thesis, many of these assumptions will be eliminated. We believe that it is simpler to start with a restricted situation and then to generalize, than it is to start the analysis directly with a general system.

3.1 Implicit Assumptions.

Embedded in the distributed database model of section 2 were the following assumptions:

a) The databases are completely replicated at each node in the system. We make this assumption because the main emphasis of this thesis is to study the management of replicated data. Of course, in most real distributed database systems, data will not be replicated at all nodes, and in chapter 11 we discuss systems where data is not necessarily replicated everywhere (i.e., we discuss partitioned data). In that chapter we also show how the performance results obtained in this thesis can be extended to the partitioned data case. However, let us point out that in a limited number of systems, data will actually be fully replicated. For example, a distributed system directory is one distributed database where the same data may be stored at all nodes.

b) The update algorithms must be able to process the arbitrary transactions defined in section 2. As will be described in chapter 8, it is possible to design specialized update algorithms that take advantage of particular transaction types. In this thesis we will not study these specialized algorithms, and in chapter 8 we will justify our decision.

c) All transactions know that the database is fully replicated at each node. Although this may seem an unimportant assumption, it actually allows us to avoid two important issues: transaction processing and directory management. (See section 4 in chapter 1.) If a transaction knows that a value for any item can be found at any node, then there is no need to consult a directory. Furthermore, if a transaction can find all the data it needs at any single node, then no decisions as to where and how the data is to be read must be taken. Transactions simply can read the data they need at one node. Since we consider the research areas of transaction processing and directory management to be beyond the scope of this thesis, we will not attempt to eliminate this assumption. In chapter 11 where we look at partitioned data, we will continue to sidestep the issues of directory management and transaction processing.

d) Transactions specify at their inception the items they will reference. That is, we assume that a transaction knows what items it will read and what items it will modify before it performs any computations. This allows us to design update algorithms where transactions lock all the items they reference as an initial step. The implications of eliminating this assumption are discussed in chapter 10.

e) The write set of a transaction is a subset of the read (or base) set. The read (or base) set of a transaction is the set of items whose values are read by the transaction. Similarly, the write set of a transaction is the set of items whose values are modified by the transaction. The fact that the write set is a subset of the read set simplifies the presentation of some of the update algorithms. This is not a serious restriction. It is simple to force all transactions to read any item they will modify. Of course, the value does not really have to be read if it will not be used; the system simply handles the item as if it had been read. (In some algorithms, the timestamp of an item will have to be read even if the value is not. See chapter 3.) In this thesis we will not eliminate this assumption. (Notice that in a blocked system, performance does not change anyhow since a block has to be read in order to update an item.)

f) Transactions write out their results to the databases after all data has been read and all computations performed. That is, transactions have a final write phase where only writes are performed. This assumption simplifies the update algorithms tremendously, especially when the algorithms are made crash resistant. (See chapter 7.) It is relatively easy for transactions to operate in this fashion. Transactions simply save their output values as they are produced in temporary storage; when the transaction completes its computations, the values are written from the temporary storage into the databases as a last step. We will make this assumption throughout this thesis.

g) No items are added to or deleted from the distributed database. In this thesis we completely avoid the problems of dynamically creating new items and eliminating old items. These problems have been discussed in the literature [Eswa76]. Similar solutions can be designed for a distributed database, but we will not discuss these here. (For example, in the locking algorithms of chapter 3, we can use predicate locks [Eswa76], or we can lock the index page that points to the item that is to be created or eliminated.)

h) All nodes have local concurrency control mechanisms. This assumption again simplifies all the update algorithms. Since it is reasonable to expect that any node in the system will have these local controls, we will make this assumption throughout this thesis.

3.2 Some More Assumptions.

Finally, we make three more assumptions about the operation of the system:

i) There is a communication system which allows any node to communicate with any other node. That is, an update algorithm can hand the communication system a message with any node as a destination, and the communication system will deliver this message. The communication system does not guarantee that messages will arrive in the same order that they were sent. The communication system will detect transmission errors and lost messages. However, after unsuccessfully attempting to send a message a number of times, the communication system may give up and tell the user (i.e., the update algorithm) that it has been unable to deliver the message.

j) No failures occur in the system. We assume that the communication system never fails to deliver a message, and that the N nodes in the system are in operation at all times. In chapter 7 we will study the possible types of failures and we will show how the update algorithms can be made crash resistant. In that chapter we will also study the performance of the crash resistant algorithms.

k) All transactions are update transactions. We assume that all transactions modify at least one item. In chapter 9 we will consider read only transactions (queries) and how the update algorithms can be simplified to handle them. Up to chapter 9, we will use the terms "transaction", "update transaction" and simply "update" interchangeably. Notice that we will use "update" as a noun meaning "update transaction". When we wish to refer to the action of modifying values in a database, we use the term "perform an update" (instead of the verb "update") in order to avoid confusion.

We have now completed the list of assumptions and are ready to look at some of the update algorithms. But in chapter 4 we will make some more assumptions regarding the performance of the distributed database system.

CHAPTER 3

THE ALGORITHMS

In this chapter we will present a set of update algorithms. Each of these algorithms is a global concurrency control mechanism of the type described in chapter 2. The set of algorithms is in no way exhaustive: we have only chosen a small set of representative algorithms. In addition, we also present some new algorithms that were developed as part of this thesis research. The performance of most of the algorithms we include in this chapter will be analyzed in chapters 4, 5 and 6.

The goals of the update algorithms are to (1) guarantee that all transactions observe a consistent database, and (2) guarantee that the values of an item converge to the same value if no new transactions are received. (See section 2 in chapter 2.) In this chapter we will only give informal arguments as to why the algorithms presented comply with the two requirements. In chapter 9 we will show how these informal ideas can be made formal. We give complete formal proofs for two of the algorithms in this chapter (i.e. the MCLA-h and the DVA algorithms).

The update algorithms can be divided into two classes: the centralized and the distributed control algorithms. In the centralized control algorithms, one special node, the central node, is assigned the concurrency control function. These centralized algorithms are presented in section 1. In the distributed control algorithms, there is no distinguished central node and all nodes share the concurrency control function. The distributed control algorithms are given in section 2.

1. THE CENTRALIZED CONTROL ALGORITHMS.

1.1 The Complete Centralization Algorithm With Acknowledgments, CCAA.

When we discussed single database systems in section 1 of chapter 2, we mentioned that one way to eliminate interference between transactions was to execute them one at a time. In a distributed database, we can do exactly the same thing. We call this solution the complete centralization algorithm with acknowledgments or CCAA. (This solution is also called the primary copy algorithm in [Alsb76]). We select one of the nodes as the "central node". This node will be in charge of serializing the update transactions. All transactions are forwarded to the central node where they are executed one at a time. All data needed by a transaction will be read at the central node. Similarly, all computations will be performed there. When the update values are ready, the central node broadcasts the new values to all nodes in the system. The central node will wait until the values are stored at all nodes before starting to process a new transaction. Since there is absolutely no possibility of interference among transactions, we can guarantee that all update transactions see consistent data.

We now give a brief description of the CCAA algorithm:

(1) Update transaction A arrives at node x from a user.

(2) Node x forwards transaction A to the central node.

(3) When the central node receives an update transaction A, it places it in a queue. Update transaction A waits in the queue until its turn to be executed comes up.

(4) When A's turn comes, it is executed at the central node. (At this point, all previous transactions have completed at all nodes.) The values requested by A are read from the database at the central node, the computations are carried out, and the new values are stored in the local database.

(5) "Perform update A" messages are sent out by the central node to all other nodes giving them the new values that must be stored at each site.

(6) Each node that receives a "perform update A" message, stores the new values produced by A into its database. Then an acknowledgment message is sent back to the central node.

(7) When the central node receives acknowledgments for the "perform update A" messages from all the nodes in the system, then it knows that A has completed everywhere. Thus, the central node gets the next update transaction that is

waiting in its queue and processes it. (See step 4.) (End of CCAA algorithm.)

One important advantage of the CCAA algorithm is that it is very simple. On the other hand, there are three potential disadvantages with the CCAA algorithm. The first problem is that update transactions are executed serially, with absolutely no parallelism. Thus, we do not expect this algorithm to be very efficient.

The two remaining potential disadvantages are due to the fact that there is a centralized control node. Since these disadvantages occur in all the centralized control algorithms, we will discuss them in the following section.

1.2 Potential Disadvantages of the Centralized Control Algorithms.

Another potential problem of the CCAA algorithm is that if the central node crashes, then no more transactions can be processed. This problem also arises in all the other centralized control update algorithms. Since for the time being we have assumed that no failures occur in the system (see section 3.2 of chapter 2), we do not have to worry about this problem at this point. In chapter 7 we will consider failures and how they affect the performance of the algorithms. However, it is appropriate to make a few short comments here regarding failures so that readers may have an idea of what is coming up in chapter 7. It is possible to make the CCAA algorithm, as well as the other centralized control algorithms, resilient in the face of failures. The main idea is to have a protocol for electing a new central node when the old central node crashes. The new central node can collect all the state information from the active nodes in the system, and based on this, it can complete any unfinished update transactions and start processing new ones. When we analyze the performance of the update algorithms (in chapters 4, 5 and 6) we will study the algorithms presented in this chapter and not the crash resistant algorithms of chapter 7.

A second problem with all the centralized control algorithms is that the central node is a performance bottleneck because all update transactions must pass through that special node. Such a bottleneck can significantly degrade system performance. In this thesis we will study this problem. We will show when the bottleneck problem arises and how serious a problem it is for the centralized control algorithms.

1.3 The Complete Centralization Algorithm (With No Acknowledgments), CCA.

The complete centralization algorithm with acknowledgments (CCAA) can be made more efficient by eliminating the acknowledgments to the "perform update" messages sent by the central node. The acknowledgments of the CCAA algorithm (step 6) were used by the central node to find out when the write steps of a transaction had completed at every node. (See section 2.2 in chapter 2.) But it is not necessary for the central node to wait until these steps complete; it is sufficient for the central node to guarantee that the write steps of transactions are performed at every node in the same order as they were performed at the central node. This can be done by assigning a *sequence number* to each update transaction that is executed at the central node. The sequence number assigned to an update transaction is an integer equal to one plus the sequence number of the previous update transaction processed. The sequence number of a transaction is appended to all the "perform update" messages for that transaction, and is used to order the storage of the new values (the write steps) into the database at each node. Since with this sequence number mechanism the steps of all transactions are executed in the same order as in the CCAA algorithm, the CCA algorithm is logically equivalent to the CCAA algorithm.

In summary, the CCA algorithm operates as follows:

(1) Update transaction A arrives at node x from a user.

(2) Node x forwards transaction A to the central node.

(3) When the central node receives an update transaction A, it places it in a queue. Transactions from this queue are executed one at a time at the central node. That is, the values requested by A are read from the local database, the computations are carried out, and the new values are stored in the local database. (Update transactions can be executed in parallel at the central node as long as a local concurrency control guarantees that the effect on the database is as if transactions were performed one at a time.) A sequence number is assigned to transaction A. This number represents the order, with respect to other transactions, in which A was executed.

(4) "Perform update A" messages are sent out by the central node to all other nodes giving them the new values that must be stored at each site. The sequence number of A is appended to these messages. After the central node sends out these "perform update A" messages, it is done with A and can start processing any other update transactions on its queue.

(5) When a node y receives a "perform update A" message, it waits until

it has processed all "perform update" messages from transactions with lower sequence numbers. (Notice that the largest sequence number processed so far must be remembered by all nodes.) Then node y stores the new values into its local database, as indicated by the message. (End of the CCA algorithm.)

Since the CCA algorithm is clearly more efficient and only slightly more complex than the CCAA algorithm, we will only study the CCA algorithm in this thesis. (In addition to being more efficient, the CCA algorithm has one other advantage over the CCAA algorithm: It is simpler to make the CCA algorithm crash resistant than it is to make the CCAA algorithm crash resistant. This is actually a topic for chapter 7, but since we will not consider the CCAA algorithm further, we make this comment at this point. Notice that if any noncentral node in the CCAA algorithm is down, the central node cannot process transactions because it is unable to get the required acknowledgments. This does not happen in the CCA algorithm. Furthermore, the sequence numbers of the CCA algorithm are very useful for discovering transactions that were missed when a node was down.)

1.4 The Centralized Locking Algorithm With Acknowledgments, CLAA.

We will now investigate other centralized control approaches in order to try to improve the performance of the CCA algorithm. If we look at the CCA algorithm, we realize that the central node is the system bottleneck because it has the highest load. If we could somehow reduce the amount of work that is done at the central node, we could improve the performance of the system.

In the CCA algorithm, the central node is performing two distinct functions: (a) the central node is reading the data and performing the computations for all update transactions, and (b) the central node provides the necessary concurrency control for the transactions (i.e., it serializes the transactions). In the algorithm we will propose now, the centralized locking algorithm with acknowledgments ,CLAA, we will move function (a) to the other nodes in order to reduce the load at the central node. Function (b), which is naturally performed at the central node, will remain there.

In the CLAA algorithm, the central node will provide concurrency control by managing locks for the items in the database. (The central node acts just like the lock manager described in section 1.4 of chapter 2.) Before an update transaction is executed, it will request locks for the items it references. When the locks are granted by the central node, the transaction will be able to proceed

knowing that no other update transaction will interfere.

In the CLAA algorithm, an update transaction A that arrives at node x is processed as follows:

(1) Node x requests from the central node locks for all the items referenced by transaction A.

(2) The central node checks all of the requested locks. If all can be granted, then a "grant" message is sent back to node x. If some items are already locked, then the request is queued. There is a queue for each item and a request only waits in one queue at a time. To prevent deadlocks, all transactions request locks for their items in the same predefined order.

(3) Once node x gets all of the requested locks, it can proceed with the transaction. (At this point, node x knows that all previous transactions that referenced items referenced by A have completed everywhere in the system.) The items are read from the local database, and the update values are computed. A "perform update A" message is sent to all other nodes informing them of the update. Node x updates the values stored in its local database.

(4) When the other nodes (including the central node) receive "perform update A" messages, they perform the indicated update on their copy of the database. After a node has processed the "perform update A" message, it sends an acknowledgment message to node x.

(5) When node x receives acknowledgments for the "perform update A" messages from all the nodes in the system, then it knows that A has completed everywhere. Thus, node x can send a "release locks of A" message to the central node.

(6) When the central node receives the "release locks of A" message, it releases the locks of the involved items. Transactions that were waiting on these locks are notified and can continue their locking process at the central node. (End of CLAA algorithm.)

1.5 The Centralized Locking Algorithm (With No Acknowledgments), CLA.

We can now use the same idea that was used to eliminate the need for acknowledgments in the CCAA algorithm to eliminate the acknowledgments of step 4 of the CLAA algorithm. As a further simplification, the "release locks" message (step 5 of the CLAA algorithm) can be merged with the "perform update" message (step 3) to the central node. If we do this we obtain the centralized locking algorithm with no acknowledgments or CLA. In the CLA algorithm, the

central node assigns a sequence number to each update transaction after it has obtained its locks. This sequence number is sent to the transaction originating node via the "grant" message and is then appended to all "perform update" messages. The sequence numbers are used to order all the steps of the transactions so that no interference can occur.

We now give an outline of the CLA algorithm:

(1) An update transaction A arrives at node x. Node x requests from the central node locks for all the items referenced by the transaction.

(2) The central node checks all of the requested locks. If all can be granted, then a "grant" message is sent back to node x. If some items are already locked, then the request is queued. There is a queue for each item and a request only waits in one queue at a time. To prevent deadlocks, all transactions request locks for their items in the same predefined order.

(3) When node x gets the "grant" message for A (together with A's sequence number), it must wait until all transactions with lower sequence number than A's sequence number have completed at node x. (This was not required in the CLAA algorithm. This wait is now needed because in the CLA algorithm (without acknowledgments), the fact that a transaction holds locks on the items it references does not imply that all previous conflicting transactions have completed everywhere.) After all transactions with lower sequence number than A's sequence number have completed at node x (i.e., after the "perform update" messages for these transactions have been received and processed), the items requested by A are read from the local database and the update values are computed. A "perform update A" message (with A's sequence number) is sent to all other nodes informing them of the update. Node x updates the values stored in its local database.

(4) When another node receives its "perform update A" message, it waits until it has processed all "perform update" messages from transactions with lower sequence number. (Notice that the largest sequence number processed so far must be remembered by each node.) Then the indicated update is performed on the local database. When the central node receives its "perform update A" message, it releases the locks of the involved items and then performs the update on the local database as indicated above. Transactions that were waiting on the released locks are notified and can continue their locking process at the central node. (End of CLA algorithm.)

Some readers might suspect that by serializing the steps of transactions with sequence numbers we have lost some of the parallelism that was obtained by using a lock manager. We show this to be true in the next section.

1.6 Sequence Numbers Produce Unnecessary Delays.

The centralized locking algorithm as stated above may produce unnecessary delays in update transactions due to the sequence number restriction. An example is the best way to illustrate this problem.

Suppose that a large update transaction (i.e., one involving many items) arrives at node 1. A lock request is sent to the central node. At the central node, the locks are granted and the transaction is assigned a sequence number, say number 10. The grant message is sent to node 1 where the transaction is executed (assuming that node 1 has processed all updates with sequence numbers less than 10). Executing transaction 10 consists of reading all items in its read set and doing some computations with the values read. Since we assumed that this transaction referenced many items, executing the transaction at node 1 will take a long time.

Suppose that while transaction 10 is being executed at node 1, another transaction arrives at node 2. Node 2 sends a lock request to the central node. Let us assume that this new transaction has no items in common with transaction 10 or any other transactions which are still in progress. Then the central node can grant the requested locks and assigns sequence number 11 to this transaction. A grant message is then sent to node 2 indicating that it can proceed with transaction 11. But node 2 will not be able to execute the transaction because it has not seen transaction 10 yet (i.e., because of the sequence number rule). However, we know that transactions 10 and 11 have no items in common and that they could be performed concurrently. Unfortunately, node 2 does not know this fact.

As far as node 2 knows, the following sequence might have occurred: The locks of transaction 10 were granted, the update performed at all nodes except node 2 and the locks released at the central node. The "perform update" message to node 2 (step 4 in the CLA algorithm) has been delayed and is on its way. Then transaction 11 arrived. It conflicts with transaction 10, but since the locks of transaction 10 have been released, transaction 11 can proceed. Thus transaction 11 has obtained its locks but it cannot be performed at node 2 until node 2 has performed update 10.

Going back to our original situation, if we want node 2 to be able to proceed with transaction 11 while transaction 10 is being executed at node 1, we must give node 2 additional information that permits it to distinguish the current case from the hypothetical case where transactions 10 and 11 conflict. This additional information is available at the central node. There are several ways in which the central node can give node 2 this information. In this thesis we will discuss three

ways in which this can be done. The algorithm that uses the first method (called the WCLA algorithm) will be presented in section 1.7; the algorithm that uses the second alternative (called the MCLA algorithm) is given in section 1.8; the third method is used by the TWCLA algorithm of section 1.10. Several variations of the MCLA algorithm are considered in section 1.9.

1.7 The Centralized Locking Algorithm With "Wait-For" Lists, WCLA.

In the WCLA algorithm, the central node keeps track of the last update transaction that referenced each item in the database. In other words, the central node keeps a table, LAST(i), where LAST(i) is the sequence number of the last update transaction that locked item i. Then, when an update transaction A obtains its locks, the central node constructs a "wait-for" list for transaction A. This list, which we will call wait-for(A), includes the sequence number of all update transactions that A must wait for before being executed. Wait-for(A) is simply the list of the LAST(i) entries for all items i referenced by A. The wait-for(A) list is appended to the grant message to A's originating node x. Before node x executes transaction A, it must wait until all "perform update" messages for transactions in wait-for(A) have been processed locally. Notice that node x will only wait for transactions whose resulting values are absolutely necessary *for executing A. In our example, update transaction 11 will not be delayed by* transaction 10 because transaction 10 did not conflict with transaction 11 and hence is not in the wait-for list of transaction 11. Wait-for(A) must also be appended to all "perform update" messages for A, so that the new update values produced by A can be stored at all nodes in the proper sequence and without unnecessary delays.

There are two potential overhead sources in the WCLA algorithm. One is the processing that is needed before an update can be performed. That is, before performing an update, a node must check that all "perform update" messages for transactions in the wait-for list of the update have been seen. To do this, nodes need to have a list of the sequence numbers of all previously processed "perform update" messages. This list may be very long, but there are many ways to compact it. Thus, we expect this list to fit in main memory at each node, and the CPU time needed to check the wait-for list against this list of performed updates should be relatively small.

A more serious source of overhead is the construction of the wait-for lists at the central node. This node must keep a sequence number (i.e., LAST(i)) for

each item in the database, and in most cases this information will not fit in main memory. Thus, in order to read or modify this information, the central node must use an IO device. This is undesirable because we are trying to reduce the processing loads at the critical central node.

1.8 The Centralized Locking Algorithm With Hole Lists, MCLA.

In this section we present an alternative to the WCLA algorithm which does not have the IO overhead at the central node associated with wait-for lists. The idea again is to send additional sequencing information with the grant messages, but we choose information which is more easily accessible at the central node.

Let us use the term *hole list* for the list of update transactions in progress (i.e., locks granted but not released) at the central node. (We use the term hole list because each entry in the list is a hole or a missing entry in the list of transactions that have released their locks.) When the locks of an update transaction are granted, the transaction's sequence number is added to the hole list. When an update releases its locks at the central node, its sequence number is removed from the hole list.

Now consider the relationship between an update transaction A which has just obtained all its locks at the central node and the hole list existing at that instant. If update transaction B is in the hole list, then A and B can not have referenced common items (else A could not have gotten its locks). Therefore, A does not have to wait for B. In other words, the hole list existing at the instant when A obtains its locks is a *do-not-wait-for* list because it contains the sequence number of transactions that can be executed in parallel with A. If we append the hole list to the "grant" message to A's originating node x, then transaction A can be executed at node x even if node x has not performed the updates in the hole list. In our example, sequence number 10 would be in transaction 11's hole list, so transaction 11 will not be delayed.

Notice that there may be other update transactions which are not in the hole list but do not conflict with A either. For example, a transaction C which does not conflict with A, but released its locks before A got its locks is in this category. We then see that the hole list is a partial "do-not-wait-for" list. If we compare the hole list for an update transaction A with a complete list of all the transactions that do not conflict with A, we find that the hole list contains the more recent entries in the complete list. However, the older transactions in the complete list have probably already been processed at all nodes and are therefore

not capable of producing delays like the one illustrated in section 1.6. So the hole list will probably be enough to eliminate almost all unnecessary delays. As a matter of fact, if the transmission delays are uniform (as we will assume in our performance model of chapter 4), the use of a hole list will eliminate all unnecessary delays. This is true because in this case all the "perform update" messages for transactions not in A's hole list will arrive at A's originating node before the "grant" message arrives at that node.

In summary, hole lists are used as follows. When an update transaction A obtains its locks at the central node, a sequence number S(A) and a copy of the hole list H(A) are appended to the "grant" message for A. Transaction A will be executed at A's originating node only when all transactions with lower sequence number than S(A) but not in H(A) have been seen locally. The sequence number S(A) and the hole list H(A) are also appended to all "perform update" messages so that the values produced by A can be stored at all nodes in the proper sequence. That is, before a node stores the values produced by A, it must have stored all values for updates with lower sequence number than S(A) but not in H(A).

The advantage of the MCLA algorithm over the WCLA algorithm is that the hole list can be kept in main memory and is easy to update. Thus, the IO overhead for locking in the MCLA algorithm is almost zero. (In most cases, the lock table can also be kept in main memory as a hash table.) The disadvantage of the MCLA algorithm is that it does not eliminate all unnecessary delays. But for a system where communication delays have a small variance, the hole list mechanism will eliminate almost all unnecessary delays.

1.9 Limited Hole Lists.

In the MCLA algorithm of section 1.8, we assumed that the complete hole list copy can be sent in the "grant" and the "perform update" messages of every update transaction. In some cases, it might not be possible or desirable to transmit hole lists of arbitrary length in messages, so we must set a practical limit for the size of the hole list copy. Let us call this preset limit h. The hole list at the central node will have to be complete; however, the copies made of it for transmission will be cut to size h in case they are longer than h.

There are several alternatives for handling the case of an overflowing hole list copy. In the next three subsections we will describe three options.

1.9.1 The MCLA-h Algorithm.

We call the first alternative for handling limited hole list copies the MCLA-h algorithm. The basic idea in the MCLA-h algorithm is the following one. When an update obtains all of its locks, the central node checks the size of the hole list. If the size of this list is less than or equal to h, then a copy of the hole list is added to the "grant" message (and later to the perform update messages) and the update can proceed. If the size of the hole list is greater than h, then the update transaction is deferred at the central node. The update and its copy of the hole list are placed in a deferred queue. Then, as holes disappear from the list (i.e., as updates release their locks), we also remove those holes from the copies of the hole list of deferred updates. As soon as a deferred update has a hole list copy of size less than or equal to h, the update is removed from the deferred queue and it is allowed to proceed.

We will now give a brief outline of the MCLA-h algorithm:

(1) An update transaction A arrives at node x. Node x requests from the central node locks for all the items referenced by the transaction.

(2) The central node checks all of the requested locks. If some items are already locked, then the request is queued. There is a queue for each item and a request only waits in one queue at a time. To prevent deadlocks, all transactions request locks for their items in the same predefined order.

(3) When transaction A obtains all of its locks, the central node assigns it a sequence number. Then the central node makes a copy of the current hole list and assigns this copy to transaction A. This hole list copy for A consists of the sequence numbers of the update transactions that are currently holding locks at the central node. The transactions in A's hole list are the transactions that A does not have to wait for before being performed. Transaction A's sequence number is added to the hole list at the central node.

(4) If the number of entries in A's hole list copy is less than or equal to the limit h, then a "grant" message (which includes A's sequence number and hole list copy) is sent to node x. If the hole list copy has more than h entries, then the "grant" message for A is delayed at the central node. As the transactions in A's hole list copy release their locks at the central node, their sequence number is removed from A's hole list copy. As soon as the hole list copy of A contains h (or less) entries, the "grant" message for A is sent out to node x.

(5) When node x gets the "grant" message for A (together with A's sequence number and hole list copy), it must wait until all transactions with lower sequence number than A's sequence number but that are not in the hole list copy of A have

completed at node x. Once the "perform update" messages for the necessary transactions are received at node x, the items requested by A are read from the local database and the update values are computed. A "perform update" message (with A's sequence number and hole list copy) is sent to all other nodes. Node x updates the values stored in its local database.

(6) When another node receives its "perform update A" message, it waits until it has processed all "perform update" messages from transactions with lower sequence number but that are not in the hole list copy of A. The indicated update is then performed on the local database. When the central node receives its "perform update A" message, it releases the locks of the involved items. Transactions that were waiting on the released locks are notified and can continue their locking process at the central node. The sequence number of A is removed from the hole list. Transaction A's sequence number is also removed from the hole list copies of any transactions that were delayed because their hole list copy was too large. (If any of these copies now have h entries, the corresponding "grant" message is sent.) The local database at the central node is also updated. (End of the MCLA-h algorithm.)

Since the MCLA-h algorithm is one of the most important algorithms we will study in this thesis, we also give a detailed description of this algorithm in appendix 1. Notice that a MCLA-infinity algorithm is simply an algorithm that has no limit for the size of the hole lists, while a MCLA-0 algorithm only grants locks to an update when all previous updates with locks granted have completed.

The performance of the MCLA-0 algorithm should be very similar to the performance of the original centralized locking algorithm described at the beginning of this report. However, these algorithms are not identical since in the MCLA-0 algorithm the updates are delayed at the central node, while in the other algorithm, the updates are delayed at the originating node (after having been granted locks) waiting for updates with lower sequence numbers.

1.9.2 The Truncating Alternative.

A second alternative for handling hole list copies of a limited size is simply to truncate the lists that are too long. Hence, instead of delaying the update transactions at the central node before sending the "grant" message (as described above), we simply eliminate some entries from the hole list and send the "grant" message immediately. Notice that we can just take out sequence numbers out of the hole list because it is a do-not-wait-for list. The fact that some sequence numbers (i.e., holes) are missing from the hole list of a transaction simply means

that the transaction will be delayed at its originating node. Of course, if the "perform update" messages for the holes that were not transmitted happen to arrive before the "grant" message does, then there will be no delay.

The problem with this method is how to decide what holes to cut off the hole list copy. If we could know in advance which will be the first holes to disappear from the list (e.g., which updates in the hole list will release their locks first), we could truncate those holes from the list and save time. However, if we do not know what holes will disappear first from the hole list, it might be better to delay updates at the central node. The two alternatives we have described so far, delaying at the central node and truncating the hole list, will be compared in chapter 6.

1.9.3 Bit Maps.

A third alternative for handling limited hole list copies is to use bit maps to compact the hole list. Since most updates that have gotten their locks but not released them (i.e., holes) will probably have sequence numbers numerically close to the latest sequence number issued, we can use a small bit map (e.g., 32 bits) to represent the hole list. However, since there can always be holes with very small sequence numbers (caused by updates that take a very long time to complete), a small bit map might not cover all holes. Therefore, a hybrid method is suggested where part of the hole list is represented by a bit map and the remaining holes are represented by a list of sequence numbers. Hopefully, in most cases the bit map will be sufficient to cover all holes, making the handling and transmission of hole lists very efficient. Clearly, in some cases, the compacted hole list will be too large and will have to be cut. The options for doing this are the same as for the noncompacted case.

1.10 The Centralized Locking Algorithm With "Total-Wait-For" Lists, TWCLA.

In section 1.7 we considered the use of wait-for lists to convey additional sequencing information. In this section we study a different type of wait-for list which we call the "total-wait-for" list. Let the *total-wait-for list* be the list of *all* transactions which have released their locks at the central node. That is, the sequence number of an update transaction is added to the total-wait-for list when the transaction releases its locks at the central node. An update transaction T

which just obtained its locks could have conflicted with any transaction on this list, but does not conflict with transactions not on this list. Thus, the total-wait-for list existing at the time when T obtains all its locks is the list of *all* transactions that T must wait for before being executed.

The way we have described the total-wait-for list, it would grow indefinitely. However, elements can be deleted from the list: Say an update A releases its locks and its sequence number is added to the total-wait-for list. When A obtained its locks, a copy of the total-wait-for list that existed at that time was appended to A. Suppose that A's total-wait-for list contained the sequence numbers of updates $B_1, B_2, \ldots B_n$. Since A is now in the total-wait-for list, we can at this point delete the sequence numbers for updates $B_1, B_2, \ldots B_n$ from the list because by waiting for A we automatically have to wait for $B_1, B_2, \ldots B_n$. Therefore, the sequence number of an update B_i remains on the total-wait-for list only from the time B_i releases its locks to the time when another update A with the sequence number of B_i in its total-wait-for list releases its locks.

As with the hole list, a copy of the total-wait-for list is added to all "grant" and "perform update" messages. Before a node performs an update A, it must make sure that it has previously processed all updates on A's total-wait-for list. Again, there are several alternatives available if the size of the total-wait-for list must be limited to a size h.

One alternative is to delay updates at the central node until their total-wait-for list shrinks to a size less than or equal to h. Unfortunately this strategy does not work as well as it did for hole lists because in the case of total-wait-for lists, we cannot delete any elements without adding new ones. Thus, it might be a long time before the total-wait-for list shrinks. Another alternative is to truncate the total-wait-for list and to send the "grant" message immediately. Since we cannot simply drop elements off the total-wait-for list (as we could do with the hole list), we must be careful. One way to handle truncated total-wait-for lists would be to only remove the elements with the smallest sequence numbers and to use the following update execution rule: Before a node performs update A, it must make sure that it has previously processed all updates on A's total-wait-for list and it has also processed all updates with sequence number less than the smallest sequence number in A's total-wait-for list.

2. THE DISTRIBUTED CONTROL ALGORITHMS.

2.1 The Distributed Voting Algorithm, DVA.

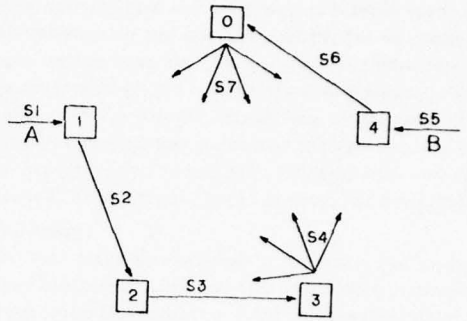
Another solution to the redundant update problem is a distributed voting algorithm suggested by Thomas [THOM76]. We call this algorithm the DVA algorithm. In this thesis we only consider the daisy chain version of this algorithm. We assume that the timestamp of a transaction is generated when the transaction is accepted. (The reason for doing this is discussed at the end of this section.) The DVA is a relatively complex algorithm, so we urge the reader to study [THOM76] carefully. Here we will only give an extremely brief outline of the algorithm. Most of the material in this outline is taken directly out of [THOM76].

Let us assume that each node in the system has a perfect clock. Let us also assume that all the clocks are synchronized. By reading the time from its clock, and by appending a node identification number to this time, every node can produce a unique timestamp. A timestamp, ts(T), assigned to each update transaction T processed by the system. (We will shortly see how this timestamp is assigned.) Each item value in the system also has a timestamp associated with it. When transaction T modifies a value, the timestamp of the value becomes ts(T). Hence, the timestamp of a value records the last time when the item was modified. (It is actually not necessary to have perfect clocks; they can be "simulated" with special counters [THOM76]. Here we assume that nodes do have perfect clocks in order to simplify the presentation.)

In the DVA algorithm there is no central controller; the nodes communicate among themselves and decide what updates can be performed. The nodes in the system form a dasy chain or ring. Before a transaction can be performed, it must move along this chain obtaining votes. After a transaction gets a majority of "OK" votes, it can be performed. A transaction may also receive a "reject" vote, in which case it may not be performed.

When an update transaction A arrives at a node, it immediately reads the items desired (and their timestamps) from the local database. Then the new update values are computed. Next comes the voting phase where A visits the nodes along the chain. Each node votes on A using the voting rule given below. (As the transaction moves along the chain, it carries with it the timestamps of the base set items that were read at the originating node.) After each vote, a node uses the request resolution rule to decide if A can be performed. The update application rule, also given below, describes how the "accept" messages for A (i.e., "perform update A" messages) are processed at each node. The steps followed by an update transaction in the DVA algorithm are illustrated through a simple

Figure 3.1



We illustrate the steps followed by update transactions in the DVA algorithm with the following two sample transactions A and B:

(s1) Transaction A arrives at node 1 and gets its first OK vote.
(s2) Transaction A visits node 2 where it gets its second OK vote.
(s3) Transaction A visits node 3 where it gets its third OK vote and is accepted. (Notice that 3 votes constitute a majority in this 5 node system.)
(s4) "Accept A" (or "perform update A") messages are sent to all nodes.
(s5) Transaction B arrives at node 4 and gets its first OK vote.
(s6) Transaction B visits node 0 and gets a reject vote. (B read obsolete timestamps or conflicted with another transaction.)
(s7) "Reject B" messages are sent to all nodes. (Node 4 re-starts transaction B from scratch at a later time.)

Figure 3.1. The DVA Algorithm: An Example.

example in figure 3.1. We now give the rules used by the DVA algorithm:

1) Voting Rule. Two update transactions *conflict* if the intersection of the base set of one and the write set of the other is not empty. (See section 3.1 in chapter 2.) Each update transaction is assigned a *priority* equal to the node identification number of its originating node. Between the time a node votes for a transaction and the transaction is resolved, the transaction is said to be *pending* at that node. The voting rule consists of five steps:
   a) Compare the timestamps for the transaction base set items with the corresponding timestamps in the local database.
   b) Vote "Reject" if any base set item is obsolete (i.e., if we find a timestamp which is more recent than the one that was read at the transaction's originating node).
   c) Vote "OK" is each base set item is current and the transaction does not conflict with any pending transactions at the node.
   d) Vote "Deadlock Reject" if each base set item is current but the transaction conflicts with a pending request of higher priority.
   e) Otherwise, defer voting and remember the transaction for later reconsideration.
2) Update Resolution Rule. After voting on a transaction, each node uses this rule to decide what must be done next. The update resolution rule consists of four parts:
   a) If the vote was "OK" and a majority of "OK" votes for the transaction exist, then the transaction is accepted. A timestamp is assigned to the transaction at this time. The timestamp must be greater than any timestamp seen in the voting process. (See chapter 9.) "Accept" messages (with the new update values and the new timestamp) are sent to all nodes.
   b) If the vote was "Reject", then reject the transaction by sending out "reject" messages to all nodes.
   c) If the vote was "Deadlock Reject" and a majority consensus is no longer possible (i.e., the transaction received a majority of "Deadlock Reject" votes), then reject the transaction by sending out "reject" messages to all nodes.
   d) Otherwise forward the transaction and the votes accumulated so far to the next node in the chain.
3) The Update Application Rule. When a node learns that a transaction, A, has been resolved, it uses the update application rule to either perform the update or to reject it.

   a) If the node receives an "accept A" message, the new values which are not obsolete are stored in the local database. That is, for each item in A's write set, the node compares the item value timestamp, t, with the timestamp of A, ts(A). If t is less than ts(A), then the item is modified as indicated and the timestamp for the item is set to ts(A). If t is greater than ts(A), then no modification is performed since the value is obsolete. All conflicting transactions that were deferred at the node because of A are rejected.
   b) If the node receives a "reject A" message, then the node uses the voting rule to reconsider conflicting requests that were deferred because of A.

(In the DVA algorithm we have described, the timestamp of a transaction A is assigned when A is accepted. An alternative is to generate the timestamp for A after A reads the values for the items in its base set at its originating node. The alternative we choose makes the proofs of chapter 9 simpler. In a failure environment, a transaction may be accepted more than once. This means that the same transaction might have different timestamps. We believe that this does not represent a serious problem because the effect of having a transaction accepted twice with different timestamps is equivalent to the effect of the transaction being accepted once.)

2.2 The Ellis Ring Algorithm, OEA.

Like the centralized locking and the distributed voting algorithms we have studied up to now, the Ellis ring algorithm (OEA) makes sure that all nodes receive the same updates and guarantees database consistency in a completely duplicated distributed database. (The "O" in the name OEA stands for "original". This is to distinguish it from the modified Ellis type algorithms that will be presented.) The Ellis ring algorithm is a distributed control algorithm. Here we will briefly describe the operation of the algorithm and refer the reader to [ELLI77] for the details and a proof of the correctness of the algorithm.

Each database in the system has a state associated with it. The state can be *idle*, *passive* or *active*. The state information can be viewed as a three way lock for the complete database at that node. An idle state corresponds to an unlocked database, while the passive and the active states correspond to special types of locked databases. A database is active when an update that originated there is in the process of locking all databases. A database is passive when it is not active but knows that an update that originated at another node is in the

process of locking all databases. Whenever a database is active or passive, the processing of all other updates that originate at that node is delayed until the database becomes idle again. A first-in first-out queue, the *internal* queue, is used for these waiting updates.

Before an update can be executed, it must obtain passive or active locks at all nodes in the system. This is done by forming a ring (or daisy chain) with all nodes and by having each update move along this chain obtaining the locks. When an update finishes this process and arrives at its originating node once more, then it can be performed. To perform the update, it is sent once more along the chain. Each node in turn performs the update on its local database and sets the state to idle. (In some cases the state will not be changed; see below.)

All updates are given a priority when they originally arrive. This priority is the node number of the originating node. When several updates are concurrently in the locking process, their priorities are used to order the updates as follows. When update A (which originated at node p) is in the locking process and arrives at a node q which is in the active state, it knows that there is another update B (which originated at q) which is also in the locking process. Therefore, if the priority of A is less than the priority of B (i.e., $p < q$) then A waits in an *external* queue at node q until update B is performed. If on the other hand, A has higher priority (i.e., $p > q$), then A can continue knowing that B will be delayed at node p.

Assuming that A is delayed (i.e., $p < q$), then when update B is performed, the locks must not be released since A will need them. (This guarantees update A a turn; i.e., A will not be "starved" by other later updates.) Thus, a special flag in update B, "update final", is set to false to indicate to all nodes that they should only perform the update and not set their state to idle. When update B arrives at node q after having been performed, q's state is set to passive and update A is released from the external queue. At that point, update A already has all the locks but it still has to complete its loop around the ring. When A arrives at node p, the update can be performed and all the locks can be released. (Assuming that no other update was in p's external queue.)

2.2.1 Outline of the Ellis algorithm.

These are the steps that must be followed when processing an update A that arrives at node p:

(1) If state(p) = passive or active then some other update(s) is in the process of updating the database. Therefore, A is delayed by placing it at the end of the

internal queue at node p. If state(p) = idle, then we can proceed: Set state(p) to active and send A to node successor(p) to obtain other locks. (Successor(p) is the node that follows p in the ring.)

(2) When A arrives at node q in the ring, we check state(q): If state(q) = idle, we set it to passive. If state(q) = passive, we change nothing. If state(q) = active and $p < q$, we change nothing. If state(q) = active and $p > q$, we delay A by placing it in the external queue at node q. Unless A was delayed, we send it on to node successor(q). If successor(q) is not p, then we repeat this step at node successor(q); otherwise we perform step 3 at node p.

(3) When A arrives at node p after having visited all nodes once, we are ready to perform the update. First, update A is performed on the local database. Then, if the external queue at p is empty, we set the "update final" flag in A to true; otherwise we set it to false. State(p) is not changed yet, and we send A to node successor(p) to perform the update (step 4).

(4) When update A arrives at node q to be performed, we perform the indicated update on the local database. If the update final flag in A is true, then we set state(q) to idle; else we do not change state(q). If successor(q) = p then we perform step 5 at node p; otherwise we repeat this step at node successor(q). If state(q) was set to idle and the internal queue at node q is not empty, then we remove one entry from the queue and start processing that update at step 1.

(5) When A arrives at node p after being executed at all nodes, we check the queues at p: If the external queue is not empty, we remove the update from the queue and send it to node successor(p) (step 2). State(p) is set to passive. If the external queue is empty, then we set state(p) to idle and check the internal queue. If it is non-empty, then we remove one entry and start processing it at step 1. After step 5, update A has been completed. (End of OEA algorithm.)

2.2.2 Comments on the Ellis Algorithm.

Notice that there is never more than one update waiting in a given external queue. Also notice that updates originating at the highest priority node are never delayed in external queues, while updates originating at the lowest priority node could be delayed in any external queue (except the one at that node).

The operation of the Ellis ring algorithm is based on the assumption that an update traveling in the ring cannot overtake or pass another update that is ahead of it in the ring. This means that all requests for service at every node must be handled in a first-in first-out fashion. Similarly, messages must arrive at a node in the same order they were sent to it. (If the communications network does not

have this property, it can be added by using sequence numbers for messages.)

To see why this assumption is important, consider the following example for an n node ring. Update A is being performed (update final is true) while updates B and C wait at internal queues in nodes $n-1$ and n respectively. When A's "perform update" arrives at node $n-1$, state($n-1$) is set to idle and B is permitted to continue (step 5). Now suppose that B arrives at node n before A's "perform update" arrives there. In this case, B finds state(n) = passive and continues on (step 2). Since node n is the only one where update B could have been delayed again, we know that B will continue around the ring and will obtain all locks. Later, A's "perform update" arrives at node n. State(n) is set to idle, update C is released, and state(n) is set to active (step 5). Update C also goes on to lock all databases. Thus, both B and C will be performed concurrently. Since this can cause problems, we must not allow B to be processed at node n before A's "perform update" is seen at node n.

2.3 Advantages and Disadvantages of the Ellis Ring Algorithm.

The Ellis ring algorithm has some advantages over the centralized algorithms of section 1 and the distributed voting algorithm of section 2.1. One advantage is that each node only has to know about two other nodes in the ring: the predecessor and the successor nodes (except if failures occur). Unlike the previous centralized locking algorithms, no prior knowledge of the items referenced by an update is needed. (See chapter 10.) Therefore an update in the Ellis algorithm can decide what items to update after it has obtained its locks. (Updates in the previous algorithms would not need prior knowledge either if they simply locked (or referenced) all items in the database.) Another advantage of the Ellis algorithm is that it requires very little state information. No timestamps or locks for individual items are needed. Thus no time will be spent reading timestamps or locks from an IO device because all state information can be kept in main memory.

The Ellis ring algorithm has two major disadvantages. First, updates must lock the complete database. This eliminates the possibility of concurrently performing updates that do not conflict. Except in special circumstances, this is a serious drawback. Another disadvantage of this algorithm is that updates must travel along the ring twice. This introduces transmission delays not found in the other algorithms. We will now discuss modifications to the Ellis algorithm that solve the first problem and reduce the magnitude of the second one.

2.4 The Modified Ellis Ring Algorithm, MEAS.

In this section we describe a modified Ellis ring algorithm (MEAS) which allows updates that do not conflict to execute concurrently. The basic idea is to have state information associated with each item at each database. Thus, the state of item i at node j, state(i,j), can be idle, passive or active. Similarly, we provide internal and external queues for waiting on each item at each node. And now, each item referenced by an update will be locked. (A priori knowledge of the base set is now required.)

There are two alternatives for locking the items in an update. One is to lock each item sequentially, i.e., attempt to lock item i only when locks for item $i-1$ have been secured at all nodes. Clearly, this alternative is undesirable because it requires that the update circulate around the ring once for each item to be locked. Therefore, we only consider the second alternative.

The second alternative is to lock all the items as we visit each node in the ring. That is, when an update A arrives at node p, we set state(i,p) to active for all items i referenced by A. If for some item k, state(k,p) is already active or passive, the we wait on k's internal queue at node p. After A has obtained all local locks, it travels along the ring requesting locks for all items referenced. If any item is not available, then A must wait on the item's external queue. When A returns to node p, it has locked all referenced items at all nodes and the update can be performed. If no other updates are waiting in the external queue of item i (referenced by A), then A's locks can be released at all nodes. If on the other hand an update is waiting, then the locks for that item are not released. In other words, each item in A will have an "update final" flag which will indicate to each node that performs the update what items are to be released.

This alternative allows non-interfering concurrent updates and is more efficient than the first alternative. Unfortunately, there are two problems that we must deal with before this solution works properly: update overtake and deadlocks.

2.4.1 Update Overtake in the Modified Algorithm.

Since updates can be delayed as they move along the ring, it is possible for one update to overtake or pass one ahead of it. This violates an assumption that was made for the original Ellis algorithm. One of the problems that can occur is illustrated by the following example.

Consider a four node network. Update A, referencing items i and j, originates at node 1. It sets state(i,1) and state(j,1) to active. At node 2, A locks item i

(i.e., sets state(i,2) to passive) but is delayed on item j. Meanwhile, update B, referencing item i only, originates at node 3. It is not delayed and sets state(i,3) to active, sets state(i,4) to passive and leaves state(i,1) and state(i,2) as they were (active and passive respectively). (See step 2 in original Ellis algorithm.) Thus, update B overtakes update A which is waiting at node 2 on item j. Since node 3 sees no updates in i's external queue at that node, then update B is performed with item i's "update final" flag set to true. State(i,2) will therefore be set to idle, but A still thinks that it has a lock for item i at node 2. This situation can lead two updates to modify item i at the same time.

In the modified algorithm, once an update B has obtained all locks for an item i, it cannot find out if there are other updates attempting to lock the item simply by checking the external queue for that item. If there is an update in the external queue, then there is no problem: no locks are released for item i when B is performed. However, if the external queue is empty, we cannot simply release all locks for item i (e.g., set state(i,all nodes) to idle). We would like to leave alone any locks that are being held by an update different from B and to release any locks that are exclusively held by update B.

One way to do this is to remember at each node who has locked what items. A simpler solution can be obtained by noticing that only updates with a lower priority than B's could also be holding locks on the items referenced by B. (If an update C with priority greater than or equal to B's priority had locks on item i, it must have set state(i,x) to active, where x is a node number greater than on equal to B's originating node number. This means that B cannot get past node x and thus it cannot have all the locks for item i.) Therefore, we only need to remember the minimum priority of the updates that have obtained locks for item i. Furthermore, this only has to be done for passive locks since active locks (other than the one held by B) must have been obtained by updates with lower priority.

Therefore, we define "lowest priority(i,x)" to be the smallest priority of the set of updates that passive locked item i at node x. An update "passive locks" item i at node x when it either sets state(i,x) to passive or when it finds state(i,x) already passive and continues. Lowest priority(i,x) is only defined when state(i,x) is passive. When an update B is performed with update final flag set to true for item i, passive locks where lowest priority(i,x) is less than the priority of B will not be released. All other locks will be released.

This modification allows updates to be delayed while requesting locks. To see that the modified algorithm operates correctly, consider the state of the system after an update B has been performed with update final flag set to true for item

i. Update B is performed correctly since it did obtain all the locks for item i. All other updates that reference item i are left as if they had arrived after B's completion and had gotten their locks then. Therefore, the other updates should be able to continue from this state and finish correctly.

2.4.2 Deadlocks in the Modified Ellis Algorithm.

The original Ellis algorithm avoided deadlocks by having updates only wait for higher priority nodes to become available. For the modified algorithm we must extend this idea in order to avoid deadlocks.

Deadlocks can be avoided by a-priori ordering all the items in each database. We assign a sequence number to each item and each item should have the same number in all databases. Then we form a global sequence number for each item by concatenating its node number with the item's sequence number. For example, item number 105 in node 3 has global sequence number 3-105. Deadlocks can then be avoided by using the following rule: "An update should not wait for an item with global sequence number x to become available while holding locks on an item with higher global sequence number than x."

If at each node we request locks by ascending sequence number, then the above rule can be enforced. However, there is a special case we must deal with. Suppose that an update A is waiting in the external queue at node p for item i because another update B has locked item i (i.e., state(i,p) = active). When B is performed, the update final flag for item i will be set to false and i's locks will not be released. When B has been performed, state(i,p) is set to passive and update A is allowed to continue. Notice that at this point update A "inherits" locks for item i at all nodes from update B. Thus, update A is holding "forward" locks which must be released if A ever has to wait for an item with lower global sequence number. For example, if later A has to wait for item j at node q (where q is greater than p but less than or equal to n, the number of nodes), then the locks that A holds for item i at nodes $q+1$, $q+2$,... n must be released. If $j < i$, then the lock for item i at node q must also be released.

2.4.3 The Complete Modified Algorithm.

Appendix 2 gives a step by step description of the modified Ellis ring algorithm.

2.5 The Modified Ellis Algorithm With Parallel Updates, MEAP.

In this section we describe one last modification to the Ellis ring algorithm. This modification allows an update to be performed in parallel once it has obtained all its locks. This modification can be used in the original algorithm or in the modified algorithm of Appendix 2. However, here we only consider the modified algorithm of Appendix 2. We call the new algorithm the MEAP algorithm.

The basic idea is to take advantage of the fact that the network we are considering can transmit messages from any node to any other node. (If this is not so, and the network is actually a ring, then we cannot make any improvements.) So when update A has obtained locks for all its items at all nodes (procedure Locks-obtained in Appendix 2), then it can send messages to all other nodes in parallel informing them that they can perform update A. However, in order not to violate the assumption that updates cannot overtake updates ahead of them, we must make sure that an update B that has been released by A is not processed at a node that has not performed update A.

One strategy that does this is the following one. Each node numbers all updates that originate at that node and that are to be performed. Additionally, all nodes remember the last update they have performed from each node. (For example, node 4 remembers: I have processed up to update number 403 from node 1, up to update number 100 from node 2, etc.) Furthermore, any messages from node n to node m must include the "status" of node n, the is, the list of updates that have been performed by node n. (For example, if node 4 above sends a message, the message must say: "Node 4 has processed through update number 403 from node 1, through update number 100 from node 2, etc.) When node m receives the message from node n, it will the delay processing the message until it has processed all of the updates processed by node n. This guarantees that no update will be performed or processed out of sequence. (Notice that no acknowledgments are needed for the "perform update" messages.) Update sequence numbers for updates at node n can be periodically be reset to 0. This process requires careful synchronization which is controlled by node n.

Under normal network operation, updates should seldom get out of sequence and therefore very few message processing delays will be incurred. Of course, the algorithm is now more complex and the messages transmitted are larger since they include "status" information, but this overhead should be offset by the time saved with the parallel updates. If the number of nodes in the system is large, the "status" information for each node can become very large, making parallel updates uneconomical.

The MEAP is the last algorithm we consider in this chapter. In the next chapter, we will analyze some of the algorithms presented in this chapter.

CHAPTER 4

PERFORMANCE ANALYSIS

In this chapter we analyze the performance of some of the update algorithms that were presented in chapter 3. In section 1 we describe a simple performance model of a distributed database. In section 2 we give a brief overview of the analysis technique, and in section 3 we give some queueing theory results that will be needed for the analysis of the algorithms. Then, in section 4, we analyze the MCLA algorithm. The analysis of the DVA algorithm is described in section 5. The analysis technique we describe in this chapter can also be used to analyze most of the other algorithms. However, in this thesis we will not analyze the other algorithms because their analysis is similar to the analysis of the first two algorithms. The analysis of the CLA algorithm and of the Ellis type algorithms can be found in [GARC78].

1. THE PERFORMANCE MODEL.

Figure 4.1 shows the model that was used to represent the distributed database system at each node. The model was designed to be as simple as possible while still displaying the principal characteristics of such a system. Requests for service arrive at a node from three sources: the users at the node, the network and the node itself. Each request for service can be of two types: a request for CPU time only and a request for IO service followed by CPU time. Both the IO and the CPU servers at each node service at most one request at a time, so that first-in-first-out queues are provided for waiting requests. Once the request is serviced, it can generate further requests, either at the same node or at other nodes. The request can also change various CPU controlled queues. The entries in these queues may cause more requests for service at a later time.

To illustrate the operation of the model, we will briefly describe how a vote request (in the DVA algorithm) is processed. An update transaction arrives at a

node and requests a vote. In order to vote, the node needs the local timestamps of all the items involved. Assuming that the timestamps are stored in an IO device, the request must first obtain service from the IO server. The service time is proportional to the number timestamps read. (The fact that the timestamps are stored in an IO device and not in main memory can easily be changed through a parameter to be described later.) Next the request will proceed to the CPU server where it will receive CPU time roughly proportional to the number of steps needed in the voting procedure. Then, depending on the outcome of the voting procedure, the transaction will either move on and request service at another node, or it will be deferred locally because of a conflict with another transaction. In case the transaction is deferred, it will wait until its fate can be decided. When it is time to decide its fate, the transaction will request more CPU time at the node.

The operation at a real distributed database node is probably not as simple as we have described it. The IO operations for a request will be interleaved with the CPU computations for the same request. However, we believe that the effect of a collection of small IO operations and small CPU operations for a service request is approximately equivalent to the effect of lumping all of the IO operations into one and of similarly lumping the CPU operations.

Another difference between our model at each node and a real computer is that in our model there is no multi-processing. This means that the requests are serviced serially instead of having several requests receiving service concurrently. Nevertheless, since most of the requests for service are small, the scheduling algorithm should not have a noticeable effect on the response times and the results obtained with the model should be comparable to the results obtained with a multi-processing model.

In a multi-processing environment, the different operations in progress can interfere with each other. Therefore a local concurrency control is needed at each node for both algorithms. This extra overhead can be added to our model by varying some of the model parameters that will be described in the next section.

1.1 The Parameters.

The model parameters describe a particular instantiation of our model. In order to obtain useful results, it is very important to select a small number of meaningful parameters. If too many parameters are chosen, there will be too many variables and it will be hard to understand the interrelationship of all the

Figure 4.1

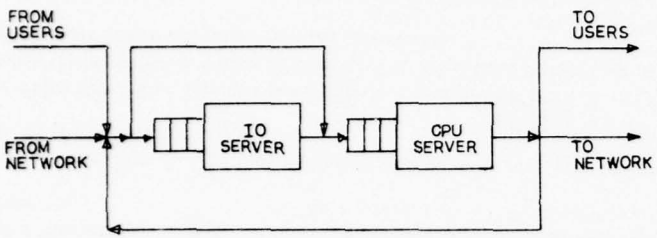


Figure 4.1. The performance model at each node.

parameters and the system performance. If the parameters are not meaningful and intuitive, it will be hard to relate them to real systems in order to assign them actual values.

The parameters we selected for the model are the following:

1) Mean interarrival time of updates at each node, A_r. We assume that the update interarrival time is exponentially distributed, that is, we assume Poisson arrivals. The mean of the distribution is A_r. This means that on the average $1/A_r$ updates per second arrive at each node. Poisson arrivals arise in many real life systems with a large number of independent users, and seem to be a good assumption for our case too.
2) Mean base set, B_s. We assume that the number of items referenced by an update transaction (the base set) has a discrete exponential distribution. A discrete exponential distribution is obtained by making a continuous exponential distribution discrete. (This will be explained in detail in section 2.) The mean of the continuous exponential distribution is B_s. The fact that the base set is exponentially distributed means that updates that reference a small number of items will occur more frequently than updates that reference many updates. We assume that an update transaction references random items in the database.

   We believe that the discrete exponential distribution is one of several reasonable distributions for the base set. Other ones are the Erlang and the normal distributions. However, in this thesis we will only consider the discrete exponential distribution

   Out of the referenced items, some will be read-only items while the rest (at least one) will be read/write. We will assume that the number of read/write items (e.g., items in the write set) in an update is uniformly distributed between 1 and the number of items in the base set. On the average, about half of the items referenced by an update will be read-only. (See section 3.5.)
3) The number of items, M. This parameter describes the total number of items in the system.
4) The number of nodes, N. The number of nodes and databases in the system is N.
5) The network transmission time, T. In order to simplify the simulation, we assume that the time it takes any message to go from any node to any other node is a constant T. This is an acceptable assumption if the communications network is lightly loaded or if the distributed database message traffic is only a small fraction of the total network traffic. In both of these cases, the

message delay is independent of the number of database messages generated. On the other hand, in some networks the transmission time may be very sensitive to message size, to network load and to the distance between nodes. Our constant transmission time does not model these cases as well.

6) CPU time slice, C_s. The CPU time slice is the time it takes any CPU server to do a "small" computation. For example, in C_s seconds, a processor can check and set one lock (in memory), compare two timestamps (in memory) and maybe add something to a queue, send a lock request to the central node, etc.

7) CPU update compute time, C_u. If an update references x items, then the time to compute the actual update values is xC_u. That is, once a transaction has been accepted or all of its locks have been granted, it will need xC_u CPU seconds before a message with the update values can be sent to all nodes.

8) IO time slice, I_s. This is the time it takes to read or write a lock or a timestamp from an IO device. For example, if a transaction at a node needs to read x timestamps, then it must get xI_s seconds of service time from the IO server.

9) IO item update time, I_d. The time needed to read or write one item value from or to the IO device is I_d.

10) Retry delay time, R_t. The last parameter is a special one since it only applies to the distributed voting algorithm. The retry time is the time a node must wait before retrying a rejected transaction.

In selecting the parameters, several additional assumptions have been made (e.g., constant transmission time). The list of assumption has grown considerably by now. (The great number of assumptions that have been made simply illustrate the great number of factors that must be considered in designing a distributed database.) With so many assumptions, how relevant can we expect the results obtained from the simulation to be? This is hard to say, but we do have one thing in our favor. We are only trying to compare the performance of different algorithms; we are not trying to predict the exact performance of a given system. Each of the factors we have considered (e.g., constant transmission time) may alter our results, but it will probably affect all the algorithms in a similar way. So even if we are unable to guarantee exact results, we will hopefully be able to discover general trends for each of the algorithms. (We will later study the effect of each assumption on the simulation results.)

1.2 The Performance Measures.

There are many variables one can choose to evaluate the performance of the system. In this thesis we will study the following variables:

1) Update response time. The response time of an update is defined as the difference between the finish time and the time when the update arrived at the originating node. There are several ways of defining the finish time; here we will consider the update to be finished when the originating node has finished all work on the update. At that point the node can inform the user that his or her update has been completed. Notice that at the finish time, other nodes might not be done with the update. The average response time of update transactions, $\overline{R}$, will be the main performance variable in this thesis.
2) Number of messages. Another important performance variable is the number of messages that must be sent per update transaction. The messages to and from a user are considered internal messages to a node and are not counted here. A broadcast message is counted as $N-1$ messages, where N is the number of nodes.
3) IO and CPU utilization. The IO and CPU utilizations at each node are also of interest. The utilization is defined as the fraction of the available time that a server is busy.

1.3 Using the Model.

The performance model we have described was used to study the algorithms of chapter 3. Two techniques were used for this: simulation and analysis.

Detailed event driven simulators were built to study the algorithms. Each simulator has an update transaction generator that produces transactions as described in the model. (See section 1.1.) The items referenced by each transaction are selected at random from the M items available. The simulator then mimics the operation of the system as it processes the transactions. Of course, the simulator does not read or write the data corresponding to a transaction; it only mimics this by requesting the necessary IO and CPU time from the servers. However, the simulator does keep track of such things as granted locks, the timestamps of the item values, and the deferred transactions. During the simulation, statistics like the average response time of transactions and the number of messages are collected.

In addition to the simulators, the update algorithms were also analyzed using a technique which is described in the following sections. The objective of this analysis was (1) to double check the performance results obtained from the simulators and (2) to obtain additional insight into the operation of the system.

For the analysis of the algorithms we make one final assumption. We assume that the CPU update time (C_u) and the CPU time slice (C_s) are both 0. This assumption is made to simplify the analysis; the extension to the case where C_u and C_s are greater than 0 should be straightforward. Furthermore, the new assumption can be justified because in most of our cases of interest, C_s and C_u are both much smaller than the IO service times (I_s and I_d) and the network transmission time (T). As long as C_u and C_s are small compared to I_s, I_d and T, they do not affect the system performance.

2. OVERVIEW.

The technique for analyzing the performance of the algorithms is iterative. First we assume that update requests never conflict with each other. Under this assumption, we derive some performance measures like the average wait time at each IO queue and the average response time of each update. Then using these measures, we compute the probability of conflict between updates. This probability is used to estimate the extra IO load at each node and the delays incurred by the conflicting requests. With these values, we recompute the average wait time and average response time. This process is repeated until the average response time value between iterations does not change noticeably (or until the computations diverge).

Throughout the computations, many simplifications are made and thus the results obtained are only approximate. However, most of the simplifications made are reasonable. The fact that the approximate results coincide fairly well with the simulation results, gives credibility to this last statement. (See chapter 5.)

In the next section, we derive some results that are useful in the analysis of the algorithms. Then we proceed to study each algorithm separately and in detail.

3. USEFUL RESULTS.

3.1 Independence of Nodes.

The interconnection of all nodes forms a network of queues [KLEI75]. The analysis of such a network is very complex for the general case. However, we simplify the analysis here by assuming that each node can be analyzed independently. This simplification can be justified as follows.

Jackson [JACK57] studied an arbitrary network of queues with each node having an exponential service time and receiving requests from outside the system in the form of a Poisson process. Jackson showed that each node in such a system behaves as if it were a M/M/1 system with a Poisson arrival rate equal to the sum of of all the original arrival rates at that node. (M/M/1 is a system with Poisson arrivals and 1 exponential server.)

In our case, all of the external arrivals are Poisson, but the service times at each node are not exponential. Fortunately, service times are "roughly" exponential because all IO service times are proportional to the number of items referenced in the update that is being serviced. Recall that the number of items referenced in an update is a discrete approximation to the exponential distribution. (The actual service time distributions at each node will be derived later on.) Thus, we hold that Jackson's result is still valid and we will analyze each node independently. The arrival rate at each node will be the sum of the external arrival rate (i.e., new updates from users) plus the arrival rate due to service requests from other nodes. (For further justification of this assumption see [GARC78].)

3.2 The M/G/1 Queue.

We only use the exponential service time assumption to decompose the network into a set of independent queueing systems. In the rest of the analysis, we use the actual service time distribution at each node. Thus each system is a M/G/1 one (G means general service time distribution) and not a M/M/1 one. The average wait time, $\overline{W}$, in such systems is given by [KLEI75]:

$$\overline{W} = \frac{\rho \overline{X}(1+C_b^2)}{2(1-\rho)} \tag{1}$$

where $\overline{X}$ is the mean of the service time distribution, C_b^2 is the squared coefficient of variation of service time and ρ is the utilization factor. The utilization factor is simply

$$\rho = \lambda \overline{X} \tag{2}$$

where λ is the Poisson arrival rate at the system. The coefficient of variation can be computed as

$$C_b^2 = \frac{\overline{X^2} - (\overline{X})^2}{(\overline{X})^2} \tag{3}$$

where $\overline{X^2}$ is the second moment of the service time. Notice that both the mean and the second moment of the service time distribution are needed to compute the average wait time.

The average number of requests waiting in the queue (not including the one in service) is give by [KLEI75]:

$$\overline{N_q} = \rho^2 \frac{(1+C_b^2)}{2(1-\rho)} \tag{4}$$

3.3 The IO Server.

The service time requested by each transaction depends on the type of request. For example, a vote request at a node (in the distributed voting algorithm) will need IO time proportional to the number of timestamps that it will read, while a perform-update request will need IO time proportional to the number of items in the write set. We model this behavior by an n-stage parallel server (figure 4.2), where each individual server services one type of request only and the probabilities at each branch are the fraction of the total requests that are of the given type. Server i is described by its mean service time $\overline{X_i}$, its second moment $\overline{X_i^2}$, and the probability α_i that an arrival goes to that server. Only zero or one request can be in the service facility at each node at a time.

We now derive the mean service time $\overline{X}$ and the second moment of the service time $\overline{X^2}$ for the complete facility at a node. From the operation of the n servers, we know that the probability distribution function of the overall service

Figure 4.2

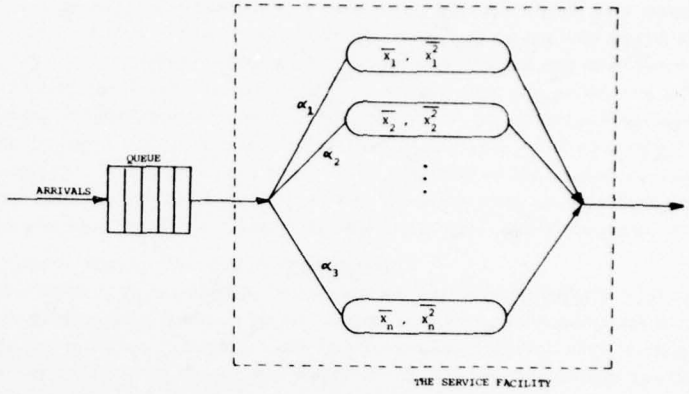


Figure 4.2. The n-stage parallel server at each node.

time is

$$f(x) = a_1 f_1(x) + a_2 f_2(x) + \ldots + a_n f_n(x) \tag{5}$$

where $f_i(x)$ is the probability distribution function (pdf) of service time i. From this equation we find the mean service time

$$\overline{X} = \int x f(x)\,dx = a_1\overline{X_1} + a_2\overline{X_2} + \ldots + a_n\overline{X_n} \tag{6}$$

and similarly,

$$\overline{X^2} = \int x^2 f(x)\,dx = a_1\overline{X_1^2} + a_2\overline{X_2^2} + \ldots + a_n\overline{X_n^2} \tag{7}$$

3.4 The Base Set.

In order to use equations (6) and (7), we need $\overline{X_i}$ and $\overline{X_i^2}$ for each type of service request. And since these values depend on the number of items in the base and write sets, we must study the probability function of the number of items referenced in an update.

Let Y (a discrete random variable) be the number of items in the base set of an update request and let $f_Y(i)$ be the probability function of Y. We assume that $f_Y(i)$ is a discrete exponential distribution. (See section 1.1.) This distribution is related to a continuous exponential distribution (with mean B_s) as follows. Let X be a random variable with a continuous exponential distribution $f_X(x)$. Then the probability that $Y = i$ is the probability that X is between $i-1$ and i. That is,

$$f_Y(i) = Pr[i-1 < X < i] = \int_{i-1}^{i} f_X(x)\,dx, \qquad i \geq 1$$

Since X is exponentially distributed,

$$f_X(x) = \frac{1}{B_s}\exp(-x/B_s)$$

where B_s is the mean of the continuos exponential distribution. As we will see, B_s is close to the mean $E[Y]$ but is not the same. Thus,

$$f_Y(i) = \int_{i-1}^{i} \frac{1}{B_s}\exp(-x/B_s)\,dx = \exp(-i/B_s)[\exp(1/B_s) - 1]$$

Now we can obtain the mean and the second moment of Y:

$$E[Y] = \sum_{i=1}^{\infty} i f_Y(i) = [1 - \exp(-1/B_s)]^{-1} \tag{8}$$

$$E[Y^2] = \sum_{i=1}^{\infty} i^2 f_Y(i) = \frac{[1 + \exp(-1/B_s)]}{[1 - \exp(-1/B_s)]^2}$$

$$= [1 + \exp(-1/B_s)](E[Y])^2 \tag{9}$$

The details of these derivations are in [GARC78]. Notice that if $B_s \gg 1$, then the mean $E[Y]$ is approximately B_s, the mean of the original exponential distribution. ($\exp(-1/B_s)$ can be approximated by $1 - 1/B_s$.)

3.5 The Write Set.

We now derive the mean and the second moment of the write set distribution. Let discrete random variable Z be the number of items in the write set. This number should be a random fraction of the number of items in the base set. However, Z should be at least 1 because all transactions must modify at least one item. In other words, Z can be defined by

$$Z = \lfloor 1 + RY \rfloor$$

where Y is the number of items in the base set and R is a random number between 0 and 1. That is, Z is an integer that is uniformly distributed between 1 and Y.

This means that discrete random variable Z is a function of discrete random variable Y and continuos random variable R, where $f_Y(i)$ is defined above and

$$f_R(x) = 1, \qquad 0 \leq x \leq 1.$$

Using iterated expectation [PAPO75],

$$E[Z] = E[E[Z \mid Y]],$$
$$E[Z^2] = E[E[Z^2 \mid Y]],$$

so we obtain that

$$E[Z] = \frac{E[Y] + 1}{2} \tag{10}$$

$$E[Z^2] = \frac{1}{3}E[Y^2] + \frac{1}{2}E[Y] + \frac{1}{6} \tag{11}$$

(See [GARC78] for details).

Using the above results, we can now proceed to analyze the update algorithms.

4. THE MCLA ALGORITHM.

In this section we describe the analysis of the MCLA algorithm (or the MCLA-h algorithm with h = infinity). During the discussion that follows, one must keep in mind that the hole lists used by this algorithm eliminate all unnecessary delays. (See chapter 3.) Hence, the only delays we study in the analysis of the MCLA algorithm are the queueing delays at the IO servers and the delays waiting for locks at the central node.

4.1 No Conflicts Case.

The analysis of the MCLA algorithm under the no conflicts assumption is divided in two parts: the central node analysis and the other nodes' analysis. We first describe the analysis of a non-central node because it is the simpler case.

4.1.1 The Non-Central Nodes.

There are two types of IO service requests at a non-central node. The first type is a request to read the items in the base set of an update (RRBS). This occurs when a node receives a grant locks message from the central node because at that point the node must compute the update. One RRBS occurs for each update request that arrives from the users at that node, so RRBS arrive at a rate of λ per second. (λ is the inverse of the interarrival time A_r.)

The second type of request is a request to perform update (RPU). These arrive at the node for every update that is accepted in the system. Since there are N nodes, RPUs arrive at a rate of $N\lambda$ per second.

Each RRBS requires (I_d)(Number of items in update) seconds of service, so that the mean service time for this request is $I_d\,\mathrm{E}[Y]$. (Recall that I_d is the IO time needed to read or write one item.) Similarly, the second moment of this service time is $I_d\,\mathrm{E}[Y^2]$. On the other hand, RPUs have have a mean service time of $I_d\,\mathrm{E}[Z]$ and a second moment of $I_d\,\mathrm{E}[Z^2]$. The total arrival rate at the node is

$\lambda(N+1)$ and the probability of a RRBS is $1/(N+1)$, while the probability of a RPU is $N/(N+1)$.

Using equations (6) and (7), we obtain the mean and the second moment of the service time at a non-central node:

$$\overline{X_{nc}} = \left(\frac{1}{N+1}\right) I_d\,\mathrm{E}[Y] + \left(\frac{N}{N+1}\right) I_d\,\mathrm{E}[Z] \tag{12}$$

$$\overline{X_{nc}^2} = \left(\frac{1}{N+1}\right) I_d^2\,\mathrm{E}[Y^2] + \left(\frac{N}{N+1}\right) I_d^2\,\mathrm{E}[Z^2] \tag{13}$$

and using equation (1), we obtain the average wait time at a non-central node

$$\begin{aligned}\overline{W_{nc}} &= \frac{\rho\overline{X_{nc}}(1+C_b^2)}{2(1-\rho)} \\ &= \frac{(N+1)\lambda\overline{X_{nc}^2}}{2\left[1-(N+1)\lambda\overline{X_{nc}}\right]}\end{aligned}$$

since

$$C_b^2 = \frac{\overline{X_{nc}^2} - \left(\overline{X_{nc}}\right)^2}{\left(\overline{X_{nc}}\right)^2}$$

and

$$\rho = (N+1)\lambda\overline{X_{nc}}.$$

(The λ in equation (1) is now replaced by $(N+1)\lambda$, which is the total Poisson arrival rate at the node.)

4.1.2 The Central Node.

We now analyze the central node in the MCLA algorithm. All update requests that arrive at the system ($N\lambda$ per second) must request locks from the central node (RL). The IO time needed to do this at the central node is 2 I_s (number of items in update). The factor of 2 is included because the locks must first be read and then they must be set. Thus, the mean and the second moment of RL service requests are $2I_s\,\mathrm{E}[Y]$ and $4I_s^2\,\mathrm{E}[Y^2]$ respectively. After the locks have been granted, the update transactions need to read the base set in order

to compute the update. This is done at the node where the update originated (RRBS). The central node will also have to perform this type of work since λ updates per second originate at the central node. As stated in the previous section, the mean and the second moment of RRBSs are $I_d\,\mathrm{E}[Y]$ and $I_d^2\,\mathrm{E}[Y^2]$. Finally, all transactions must also release their locks at the central node and perform the update (RRLPU). These request arrive at a rate of $N\lambda$ per second. RRLPUs need

$$I_s(\text{size of base set}) + I_d(\text{size of write set})$$

seconds of IO time. Since the sizes of the base and write sets are not independent, the derivation of the mean and the second moment of RRLPUs has to be done carefully. The details are given in [GARC78] and the result is

$$\text{mean service time of RRLPU's} = I_s\,\mathrm{E}[Y] + I_d\,\mathrm{E}[Z] \tag{15}$$

$$\text{second moment of RRLPU's} = I_s^2\,\mathrm{E}[Y^2] + I_s I_d\Big(\mathrm{E}[Y] + \mathrm{E}[Y^2]\Big) + I_d^2\,\mathrm{E}[Z^2] \tag{16}$$

The total Poisson arrival rate at the central node is $\lambda(2N+1)$ and the probabilities for RL, RRBS, and RRLPU are $N/(2N+1)$, $1/(2N+1)$ and $N/(2N+1)$ respectively. Now we can use equations (6) and (7) to find the mean and the second moment of the complete service time at the node:

$$\overline{X_c} = \left(\frac{N}{2N+1}\right)2I_s\,\mathrm{E}[Y] + \left(\frac{1}{2N+1}\right)I_d\,\mathrm{E}[Y] + \left(\frac{N}{2N+1}\right)(I_s\,\mathrm{E}[Y] + I_d\,\mathrm{E}[Z]) \tag{17}$$

$$\overline{X_c^2} = \left(\frac{N}{2N+1}\right)4I_s^2\,\mathrm{E}[Y^2] + \left(\frac{1}{2N+1}\right)I_d^2\,\mathrm{E}[Y^2] + \left(\frac{N}{2N+1}\right)\left[I_s^2\,\mathrm{E}[Y^2] + I_s I_d\Big(\mathrm{E}[Y] + \mathrm{E}[Y^2]\Big) + I_d^2\,\mathrm{E}[Z^2]\right] \tag{18}$$

Substituting into equation (1), we find the average wait time at the central node:

$$\overline{W_c} = \frac{\rho\overline{X_c}(1+C_b^2)}{2(1-\rho)} = \frac{(2N+1)\lambda\overline{X_c^2}}{2\left[1-(2N+1)\lambda\overline{X_c}\right]} \tag{19}$$

Figure 4.3

FIGURE 4.3

STEPS OF A TRANSACTION

MCLA CENTRALIZED LOCKING ALGORITHM
--- No Conflicts Case ---

ORIGIN: Non-central node x.

S T E P	Average time for step.
Request arrives at node x	0
Transmission of lock request to central node	$\overline{T}$
Locks are granted	$\overline{Wc}$ + 2 Is E[Y]
Transmission of "grant" message to node x	$\overline{T}$
Read base set items	$\overline{Wnc}$ + Id E[Y]
Send "perform update" to all nodes, messages immediately arrives at node x	0
Perform update at node x	$\overline{Wnc}$ + Id E[Z]
Average response time	$\overline{Rnc}$

ORIGIN: Central node.

S T E P	Average time for step
Request arrives at central node	0
Transmission of lock request to central node	0
Locks are granted	$\overline{Wc}$ + 2 Is E[Y]
Transmission of "grant" message to node	0
Read base set items	$\overline{Wc}$ + Id E[Y]
Send "perform update" message to all nodes, message immediately arrives at central node	0
Perform update and release locks	$\overline{Wc}$ + Is E[Y] + Id E[Z]
Average response time	$\overline{Rc}$

4.1.3 The Average Response Time.

Once the average wait times at each node are known, the average response time of an update can be computed simply by studying the steps followed by an update. Since we are assuming that the network is an independent set of queueing systems, the average total response time is the sum of the average response times at each node. Figure 4.3 shows the steps followed by a transaction that originates at the central and non-central nodes. The average total response time for transactions originating at the central node is

$$\overline{R_c} = 3\overline{W_c} + 3I_s\mathrm{E[Y]} + I_d(\mathrm{E[Y]} + \mathrm{E[Z]}) \tag{20}$$

while the value for transactions originating at non-central nodes is

$$\overline{R_{nc}} = 2T + \overline{W_c} + 2\overline{W_{nc}} + 2I_s\mathrm{E[Y]} + I_d(\mathrm{E[Y]} + \mathrm{E[Z]}). \tag{21}$$

Since there are $N-1$ non-central nodes and only one central node, the average response time for all requests is

$$\overline{R} = \left(\frac{N-1}{N}\right)\overline{R_{nc}} + \left(\frac{1}{N}\right)\overline{R_c}. \tag{22}$$

This completes the analysis of the central locking algorithm when no conflicts among the updates occur.

4.2 The MCLA Algorithm – Conflicts.

We have obtained expressions for the average wait time at each node and the average response of updates when no conflicts occur. Next, we use these results to estimate the probability and the effect of conflicts, and use these values in turn to recompute the wait and response times.

The analysis that follows is only approximate. One reason for this is that a more detailed analysis is probably much harder. A second reason is that these results will have a small effect on the performance of most systems since the fraction of updates that conflict should be very small.

The main assumption we make in analyzing the algorithm under conflicts is that updates that conflict are "average" updates. By an average update we mean an update A such that A has exactly E[Y] items in its base set, A has exactly E[Z]

items in its write set and whenever A waits at a node, it waits exactly the average wait time at that node. This simplifies the conflict analysis greatly. This would be a valid assumption if the average effects of update conflicts were equivalent to the effects of average updates conflicting. Unfortunately, this is not quite true. For example, if we know that an update has had a conflict, then chances are that its base set size is greater than average. This particular point will be analyzed chapter 5, but for the time being we go ahead and use this assumption.

4.2.1 Probability of Conflict.

Our first step is to estimate the probability that two updates conflict, $\Pr(C)$. That is, assume that update A has been granted at the central node the locks for all of its $\mathrm{Y_A}$ items. Then we need the probability $\Pr(C)$ that a second update B arriving at the central node will not be able to get a lock for one of its $\mathrm{Y_B}$ items. The probability that the first item of B coincides with an item of A is $\mathrm{Y_A}/M$, where M is the total number of items in the database. The probability that the other items in B coincide with one of the items of A is also about $\mathrm{Y_A}/M$ (assuming, of course, that $\mathrm{Y_A}$ and $\mathrm{Y_B} \ll M$ and that the probability of two hits is very small). Since there are $\mathrm{Y_B}$ items, the probability of any conflict is

$$\Pr(C) = \sum_{i=1}^{\mathrm{Y_B}} \frac{\mathrm{Y_A}}{M} = \frac{\mathrm{Y_A Y_B}}{M}.$$

In the rest of our analysis, we use the average value of Pc:

$$\Pr(C) = \mathrm{E}\left[\frac{\mathrm{Y_A Y_B}}{M}\right] = \frac{(\mathrm{E[Y]})^2}{M} \tag{23}$$

($\mathrm{Y_A}$ and $\mathrm{Y_B}$ are independent.)

4.2.2 Probability of Waiting at Central Node.

Equation (23) only gives us the probability of a conflict between two updates. However, what we really need is the probability that update B will have to wait for locks at the central node ($\Pr(W)$). Let $\overline{J}$ be the average number of updates that are holding locks at the central node. Then,

$$\Pr(W) = \Pr(C)\overline{J}. \tag{24}$$

To estimate $\overline{J}$ we need to know how long a transaction holds its locks. If the transaction originated at a non-central node, the locks are held while the originating node computes the update and until the central node is notified:

L_{nc} = (transmission from central to originating node)
+ (time to compute update values)
+ (transmission from originating to central node)
+ (time to release locks).

On the average this becomes

$$\overline{L_{nc}} = T + \overline{W_{nc}} + I_d \mathrm{E[Y]} + T + \overline{W_c} + I_s \mathrm{E[Y]} + I_d \mathrm{E[Z]}. \tag{25}$$

For a transaction originating at the central node, no transmissions are involved, so

$$\overline{L_c} = \overline{W_c} + I_d \mathrm{E[Y]} + \overline{W_c} + I_s \mathrm{E[Y]} + I_d \mathrm{E[Z]}. \tag{26}$$

The average lock time for any update is

$$\overline{L} = \left(\frac{N-1}{N}\right)\overline{L_{nc}} + \left(\frac{1}{N}\right)\overline{L_c}, \tag{27}$$

since there are $N-1$ non-central nodes and only one central one.

Using Little's formula [KLEI75], we can obtain the average number of transactions holding locks at a given instant:

$$\overline{J} = (\text{arrival rate of lock requests})\overline{L},$$

or

$$\overline{J} = N\lambda\overline{L}. \tag{28}$$

This value can be substituted into equation (24) to obtain $\Pr(W)$.

4.2.3 Cost of Conflicts.

To estimate the cost of each conflict, we will assume that an update transaction will at most conflict with one other transaction during its execution. This is reasonable to do since the probability of two or more conflicts is much smaller that the probability of one conflict. A delayed transaction must wait until the lock it needs is released. On the average this time will be $\overline{L}/2$ since we assume

that the conflicting lock request arrives at a random point in time with respect to the transaction that holds the lock. After waiting, the transaction must read and set the remaining locks. We assume that the conflict could have occurred with any lock, so that the number of remaining locks is uniformly distributed between 0 and $\mathrm{Y}-1$ items (Y is the number of items in the base set). The mean and the second moment of the number of remaining locks are

$$\left(\frac{\mathrm{E[Y]}-1}{2}\right)$$

and

$$\left(\frac{\mathrm{E[Y^2]}}{3} - \frac{\mathrm{E[Y]}}{2} + \frac{1}{6}\right)$$

respectively. (See [GARC78] for details.) Reading and setting each of these locks takes $2I_s$ seconds. Therefore, the mean and the second moment of the IO service time are:

$$I_s(\mathrm{E[Y]}-1)$$

and

$$4I_s^2\left(\frac{\mathrm{E[Y^2]}}{3} - \frac{\mathrm{E[Y]}}{2} + \frac{1}{6}\right). \tag{29}$$

We can now recompute the average IO wait time at the central node (see equation 19). We now have a new type of request: the request to lock remaining items (RLR). The arrival rate of RLRs at the central node is $\Pr(W)N\lambda$ which is the fraction of the lock requests that are not granted the first time through. The total Poisson arrival rate at the central node is $(2N+1+\Pr(W)N)\lambda$. Equations (17) and (18) are modified accordingly to give

$$\begin{aligned}\overline{X_c} = &\left(\frac{N}{2N+1+\Pr(W)N}\right)2I_s\mathrm{E[Y]} + \left(\frac{1}{2N+1+\Pr(W)N}\right)I_d\mathrm{E[Y]} \\ &+ \left(\frac{N}{2N+1+\Pr(W)N}\right)(I_s\mathrm{E[Y]} + I_d\mathrm{E[Z]}) \\ &+ \left(\frac{\Pr(W)N}{2N+1+\Pr(W)N}\right)I_s(\mathrm{E[Y]}-1)\end{aligned} \tag{30}$$

$$\overline{X_c^2} = \left(\frac{N}{2N+1+\Pr(W)N}\right)4I_s^2\mathrm{E}[\mathrm{Y}^2] + \left(\frac{1}{2N+1+\Pr(W)N}\right)I_d^2\mathrm{E}[\mathrm{Y}^2]$$
$$+\left(\frac{N}{2N+1+\Pr(W)N}\right)\Big[I_s^2\mathrm{E}[\mathrm{Y}^2] + I_sI_d\big(\mathrm{E}[\mathrm{Y}]+\mathrm{E}[\mathrm{Y}^2]\big) + I_d^2\mathrm{E}[\mathrm{Z}^2]\Big] \qquad (31)$$
$$+\left(\frac{\Pr(W)N}{2N+1+\Pr(W)N}\right)4I_s^2\left(\frac{\mathrm{E}[\mathrm{Y}^2]}{3} - \frac{\mathrm{E}[\mathrm{Y}]}{2} + \frac{1}{6}\right).$$

When these new values are substituted into equation (19), we obtain the new average wait time at the central node. In turn, the new value for $\overline{W_c}$ is substituted into equations (20), (21) and (22) to give us a new average response time $\overline{R}$. However, to this value we must add the expected value of the delay due to conflicts. Since a request that conflicts is delayed $\overline{L}/2$ seconds plus the time to set the remaining locks, the new response time $\overline{R'}$ is given by:

$$\overline{R'} = \overline{R} + \Pr(W)\big(\overline{L}/2 + \overline{W_c} + I_s(\mathrm{E}[\mathrm{Y}]-1)\big). \qquad (32)$$

Here, $\overline{R}$ is the value obtained from equation (23) with the new value of $\overline{W_c}$.

The procedure we have just described can be repeated (starting at equation 23) until the increase in $\overline{R}$ is negligible. The procedure to do this is as follows:

```
Initialize; Pr(W) := 0; L := 0; oldR := 0;
SolveSystem; << Result is R, Wc and Wnc >>
Do until "R is close to oldR"
    begin
    ConflictAnalysis; << Result is Pr(W) and L >>
    oldR := R;
    SolveSystem; << result is new R, Wc, and Wnc>>
    end;
```

The complete program is given in appendix 3.

4.2.4 Convergence and Saturation.

A natural question to ask at this point is: Under what conditions does the iterative analysis we have just described converge? That is, in what cases will the difference between "oldR" and $\overline{R}$ actually approach zero and in what cases will this difference increase? We have not investigated this issue for the following reason. If the algorithm of appendix 3 does not converge within a few steps (e.g.,

3 or 4), then the number of conflicts in the system must be significant. This invalidates the main assumption of the analysis, so there is no point in studying whether the analysis algorithm will eventually converge: in either case the result is not valid. Thus, in the program of appendix 3 we give up after 5 iterations. If the program gives up for this reason, the analysis cannot provide good results.

In practice, this seems to be an adequate rule. In all the test cases where the analysis program gave up, the system was fairly close to saturation already (as verified with the simulator). Thus, the analysis can be used to estimate the point where the system becomes saturated.

Also notice that in some equations (like equation (19)) it is possible that the denominator reach zero, or even become negative. In such cases, the system is clearly saturated and our analysis assumptions are invalid. The program of appendix 3 will report this situation.

With this section we conclude the analysis of the MCLA algorithm.

5. THE DISTRIBUTED VOTING ALGORITHM (DVA).

5.1 No Conflicts Case.

We now analyze the performance of the distributed voting algorithm (DVA) under the assumption of no conflicts. The type of analysis is very similar to the one performed with the centralized locking algorithm.

In the distributed voting algorithm there is no central node, so the analysis is simplified. At any node in the system there are three types of service requests. The first type is a request to read the items and timestamps in the base set (RRIT). These requests are generated by new updates as they arrive at their originating nodes. The time needed to read one item and one timestamp is $I_d + I_s$, so that the mean and second moment of RRITs are

$$(I_d + I_s)\mathrm{E}[\mathrm{Y}] \quad \text{and} \quad (I_d + I_s)^2\mathrm{E}[\mathrm{Y}^2]$$

respectively. The arrival rate of RRITs is λ per second because one RRIT occurs for each new update that arrives at a node (and assuming no rejected updates, of course).

The second type of request for service is a request to vote on an update (RV). This type of request involves reading all of the timestamps for the items in the base set. Since reading one timestamp takes I_s seconds of IO time, the mean and the second moment of RVs are

$$I_s \mathrm{E}[Y] \quad \text{and} \quad I_s^2 \mathrm{E}[Y^2].$$

Assuming that all nodes vote OK on all updates, a given node must vote on all the updates that originate at that node. That node must also vote on all of the updates originating at the $N_m - 1$ previous nodes in the daisy chain, where N_m is the number of votes needed for a majority. Thus the arrival rate of RVs at a node is $N_m\lambda$.

Finally, there are requests to perform an update (RPU). These requests involve writing the items in the write set and updating the timestamps for these items. Since for each item in the write set we need $I_d + I_s$ seconds of IO time, the mean and the second moment of RPUs are

$$(I_d + I_s)\mathrm{E}[Z] \quad \text{and} \quad (I_d + I_s)^2 \mathrm{E}[Z^2].$$

The arrival rate of RPUs at each node is $N\lambda$ because all updates must be performed at all nodes.

The total arrival rate at a node is $(1 + N_m + N)\lambda$ and the probabilities of RRIT, RV and RPU are $1/(1+N_m+N)$, $N_m/(1+N_m+N)$ and $N/(1+N_m+N)$ respectively. We can now use equations (6) and (7) to obtain the mean and the second moment of the service time at node i:

$$\overline{X_i} = \left(\frac{1}{1+N_m+N}\right)(I_s + I_d)\mathrm{E}[Y] + \left(\frac{N_m}{1+N_m+N}\right) I_s \mathrm{E}[Y] + \left(\frac{N}{1+N_m+N}\right)(I_s + I_d)\mathrm{E}[Z] \tag{33}$$

$$\overline{X_i^2} = \left(\frac{1}{1+N_m+N}\right)(I_s + I_d)^2 \mathrm{E}[Y^2] + \left(\frac{N_m}{1+N_m+N}\right) I_s^2 \mathrm{E}[Y^2] + \left(\frac{N}{1+N_m+N}\right)(I_s + I_d)^2 \mathrm{E}[Z^2] \tag{34}$$

Using equation (1), we obtain the average IO wait time at node i:

$$\overline{W_i} = \frac{\rho \overline{X_i}(1 + C_b^2)}{2(1-\rho)} = \frac{(1+N_m+N)\lambda \overline{X_i^2}}{2\left[1 - (1+N_m+N)\lambda \overline{X_i}\right]} \tag{35}$$

Figure 4.4

FIGURE 4.4

STEPS OF A TRANSACTION

DISTRIBUTED VOTING ALGORITHM
--- No Conflicts Case ---

ORIGIN: Node i.

S T E P	Average time for step.
Request arrives at node i	0
Read items and timestamps	$\overline{W_i}$ + (Is + Id) E[Y]
Vote at node i	$\overline{W_i}$ + Is E[Y]
Transmission to node i+1	T
Vote at node i+1	$\overline{W_{i+1}}$ + Is E[Y]
Transmission to node i+2	T
Vote at node i+2	$\overline{W_{i+2}}$ + Is E[Y]
.	
.	
.	
Transmission to node i + Nm - 1	T
Vote at node i + Nm - 1, update accepted	$\overline{W_{i+Nm-1}}$ + Is E[Y]
Transmission of accept message to node i	T
Perform update at node i	$\overline{W_i}$ + (Is + Id) E[Z]
Average response time	$\overline{Ri}$

[Note: Arithmetic (e.g., i + x) is modulo N.]

(Notice that if there are no conflicts, $\overline{W_i}$ is independent of i. However, this is not true in the general case and we therefore include the i subscript here.)

The next step is to use the result of equation (35) to compute the average response time. Figure 4.4 shows the steps followed by an update transaction. The average response time is simply the sum of the average times taken at each step (by our independence assumption):

$$\overline{R_i} = \overline{W_i} + (I_s + I_d)\,\mathrm{E[Y]} + \sum_{j=i}^{i+N_m-1} \left(\overline{W_j} + I_s\,\mathrm{E[Y]} + T\right) + \overline{W_i} + (I_s + I_d)\,\mathrm{E[Z]}, \tag{36}$$

$$\overline{R} = \frac{\sum_{j=0}^{N-1} \overline{R_j}}{N}. \tag{37}$$

This ends the analysis of the distributed voting algorithm with no conflicts.

5.2 The DVA Algorithm – Conflicts.

The analysis of conflicts in the distributed voting algorithm is more complex than the equivalent analysis for the MCLA centralized locking algorithm. The technique used is the same; however, there are many cases to consider in the distributed voting algorithm. When there are no conflicts, all nodes in the distributed algorithm have the same behavior. Unfortunately, when conflicts occur, each node performs differently since decisions are made according to the node's priority (i.e., its position in the daisy chain).

In the following, we assume that we have an estimate for the average wait time at node i ($\overline{W_i}$). (See section 5.1.) Furthermore, we assume that the number of updates that conflict is small compared to the ones that don't conflict. We also assume that an update will conflict with at most one request before it is completed. (That is, at most, an update will be retried once.) Finally, we assume that all conflicting updates are average updates (just as was done for the centralized algorithm).

5.2.1 The Size of the Pending List.

The first step is to estimate the following values:

pt[i, j] = the average time that an update that arrives at node j with i OK votes will remain on j's pending list, assuming that the update is completed without any conflicts.

q[i, j] = the average size of the pending list at node j due to update requests that arrived at j with i OK votes (only considering requests without any conflicts).

For example, pt[0, 1] is the average time that an update that originated at node 1 will stay on the pending list at node 1, assuming that the update completes without any delays.

Since λ requests per second arrive at node j with i votes ($0 \le i \le N_m - 1$), then by Little's formula

$$\mathrm{q}[i, j] = \lambda\,\mathrm{pt}[i, j] \tag{38}$$

To compute pt[i, j], we consider the interval between the time an update is placed on the pending list and the time it is removed. When the update is placed on the pending list at j, it has $i + 1$ votes. Therefore it will remain on until it receives $N_m - (i + 1)$ more votes and it is accepted. Thus, if $(i + 1) \neq N_m$,

$$\mathrm{pt}[i, j] = \sum_{k=1}^{N_m-(i+1)} \left(T + \overline{W_{j+k}} + I_s\,\mathrm{E[Y]}\right) + T + \overline{W_j} + (I_d + I_s)\,\mathrm{E[Z]}. \tag{39}$$

The last two terms represent the update time needed to remove the update from the pending list. If $(i + 1) = N_m$, then no more votes are needed and

$$\mathrm{pt}[i, j] = \overline{W_j} + (I_d + I_s)\,\mathrm{E[Z]} \tag{40}$$

The next step is to study how a conflict between two updates can occur and to estimate the extra IO load and delay in each case. There are basically two types of conflicts: (1) an update, A, arriving at a node might find that its timestamps are obsolete or (2) update A might conflict with another request on the pending list.

5.2.2 Obsolete Timestamps.

Obsolete timestamps occur when a conflicting update B is accepted while update A is being processed. Update B must have obtained its N_m OK votes at

Figure 4.5

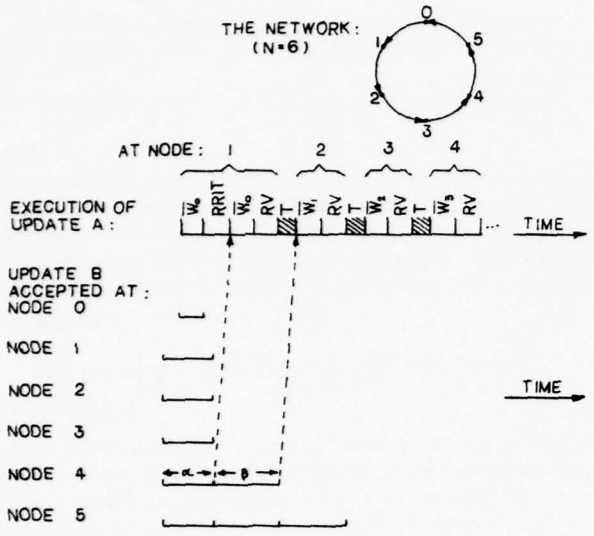


Figure 4.5. How and when obsolete timestamps occur. (Update A originates at node 0.)

nodes where A has not been processed (otherwise A would have been deferred) and similarly A must have received its OK votes ($0 \leq$ votes $\leq N - N_m$) at nodes where B has not been processed.

For example, figure 4.5 shows how and when obsolete timestamps can occur in a 6 node system. The top line shows the steps involved in processing an update request A that arrives at node 0. Below this line, we show the time intervals where the acceptance of a conflicting request B would cause A to see obsolete timestamps. These intervals are dependent on the node where B is accepted; therefore we show the intervals for all six nodes. We illustrate the case where B is accepted at node 4: If B is accepted at any time in interval α (see fig. 4.5), then the acceptance message will arrive at node 0 just before A is voted on. This means that A will see old timestamps. If B is accepted in interval β , then the accept message will arrive at node 1 before A is voted on and A will see obsolete timestamps in this case too. If B is accepted before α, then the update acceptance message will arrive at node 0 before A even reads the items, so when A reads the items it will obtain the new values. If B is accepted after β, then A will not see old timestamps at node 2 because it will be delayed (or deadlock rejected) at node 1. Notice that for B to be accepted at node 4, it must be on the pending list at node 1 and it will therefore be impossible for A to proceed past node 1. (At this point we only consider updates that see old timestamps; cases where A conflicts with an update in the pending list are treated later.)

From this example, we can write down what happens in the general case. The probability that an update that originates at node i will obtain obsolete timestamps on its vote request (RV) at node i itself is:

$$\text{Pold}[i,i] = \Pr(C')\left\{\overline{W_i}\lambda + (N-1)\left[\overline{W_i} + (I_s + I_d)\,\text{E}[Y]\right]\lambda\right\} \qquad (41)$$

where $\Pr(C')$ is the probability of a conflict. In this case, a conflict occurs when one element of the write set of the accepted update coincides with one item of the base set of the other update. Thus,

$$\Pr(C') = \frac{\text{E}[Y]\,\text{E}[Z]}{M} \qquad (42)$$

(Compare equation (42) to equation (23).) The value in curly brackets in equation (41) is the average number of updates that are accepted in the critical intervals (e.g., interval α of figure 4.5).

What is the cost of such an occurrence? The update that failed must be retried. Therefore, all IO operations up to the point where the update was rejected

will be overhead. This includes one RRIT and one RV at node i. Thus, the arrival rate of these requests must be modified:

$$\text{arrival rate of RRIT}[i] := \text{arrival rate of RRIT}[i] + \text{Pold}[i,i]\lambda \quad (43)$$

$$\text{arrival rate of RV}[i] := \text{arrival rate of RV}[i] + \text{Pold}[i,i]\lambda \quad (44)$$

(The initial values of arrival rate of RRIT[i] and arrival rate of RV[i] are the values in the previous section.) The rejected request will be delayed by

$$\overline{W_i} + (I_s + I_d)\,\mathrm{E}[\mathrm{Y}] + \overline{W_i} + I_s\,\mathrm{E}[\mathrm{Y}] + R_t$$

(and R_t is the retry time). Therefore, when we recompute the average response time (equation 37), we must add a term to account for this delay. We will do this by defining a variable "delay[i]" which will accumulate all of the delays for updates that originated at node i. Variable delay[i] is initially set to zero and when we are done recomputing $\overline{R}$ (equation 37), we will add this term. Thus,

$$\begin{aligned}\text{delay}[i] :=\ & \text{delay}[i] \\ & + \text{Pold}[i,i]\left[\overline{W_i} + (I_d + I_s)\,\mathrm{E}[\mathrm{Y}] + \overline{W_i} + I_s\,\mathrm{E}[\mathrm{Y}] + R_t\right] \end{aligned} \quad (45)$$

(where Pold is defined in equation 41.)

Up to now, we have only considered updates that are rejected at the originating node i due to obsolete timestamps. Next we consider such rejections at other nodes. This can happen at nodes $i+j$ for $1 \le j \le (N - N_m)$ only. (See figure 4.5.) (The computation of node numbers (e.g., $i+j$) is always modulo N.) The probability that the update that originated at node i is rejected at $i+j$ because of old timestamps is:

$$\text{Pold}[i,i+j] = \Pr(C')\left\{[(N - N_m + 1) - j]\left[\overline{W_{i+j-1}} + I_s\,\mathrm{E}[\mathrm{Y}] + T\right]\lambda\right\} \quad (46)$$

The effect of such a conflict is that the update progress so far (through voting at node $i+j$) will be wasted and the update must be retried. This is taken into account as follows:

$$\text{arrival rate of RRIT}[i] := \text{arrival rate of RRIT}[i] + \text{Pold}[i,i+j]\lambda, \quad (47)$$

$$\text{arrival rate of RV}[k] := \text{arrival rate of RV}[k] + \text{Pold}[i,i+j]\lambda \quad \text{for } i \le k \le i+j, \quad (48)$$

$$\begin{aligned}\text{delay}[i] :=\ & \text{delay}[i] + \text{Pold}[i,i+j]\Big\{\overline{W_i} + (I_s + I_d)\,\mathrm{E}[\mathrm{Y}] + \overline{W_i} \\ & + I_s\,\mathrm{E}[\mathrm{Y}] + \sum_{k=i+1}^{i+j}\left(T + \overline{W_k} + I_s\,\mathrm{E}[\mathrm{Y}]\right) + T + R_t\Big\} \end{aligned} \quad (49)$$

Recall that equations (46) through (49) must be repeated for $1 \le j \le (N - N_m)$, while equations (41) through (49) must be repeated for each node $0 \le i \le N-1$.

5.2.2 Conflicts with Pending Requests.

The other type of conflict occurs when an update A arrives at a node with current timestamps, but it conflicts with an update on the pending list at that node. We divide this analysis into two parts: the conflict occurs at A's originating node i (on first vote) and the conflict occurs at the other nodes ($i+1$, $i+2$, etc.).

The first case is the following one. Update A arrives at node i. It reads the items and timestamps and then i votes on A. During the voting, node i notices that A conflicts with an update B. Assume that B originated at node $i-h$ where $0 \le h \le N_m - 1$. (As always, $i-h$ is performed modulo N.) The probability that this happens is the probability that A and B conflict times the average number of requests from $i-h$ that are on the pending list at i. That is,

$$\Pr(C'')\,\mathrm{q}[h,i] \quad (50)$$

where q[h,i] is defined in equation (38) and Pr(C'') is the probability of a conflict. There are actually two ways in which A and B can conflict. An item in the write set of B can coincide with an item in the base set of A. The probability of this is

$$\Pr(1) = \frac{\mathrm{E}[\mathrm{Z}]\,\mathrm{E}[\mathrm{Y}]}{M} \quad (51)$$

and if B is ever accepted, then A must be rejected. The other way that a conflict can occur is for an item in the write set of A to coincide with an item in the read set of B (i.e., the base minus the write set). The probability for this is

$$\Pr(2) = \frac{\mathrm{E}[\mathrm{Z}]\,\mathrm{E}[\mathrm{Y} - \mathrm{Z}]}{M} = \frac{\mathrm{E}[\mathrm{Z}](\mathrm{E}[\mathrm{Y}] - \mathrm{E}[\mathrm{Z}])}{M} \quad (52)$$

and in this case A may continue after B is accepted. Therefore,

$$\Pr(C'') = \Pr(1) + \Pr(2) \quad (53)$$

Thus, with probability $(\Pr(1) + \Pr(2))\,\mathrm{q}[h,i]$ update A (origin i) will conflict with B (origin $i-h$) at node i ($0 \le h \le N_m - 1$). There are two actions that the distributed voting algorithm will take in this case: Either A is delayed at i

or A receives a DR (deadlock reject) vote. We study the effects of the actions separately.

If $i - h$ is greater than i ($i - h$ modulo N), then A will be delayed at i. In this case, with probability $\Pr(1)\,q[h,i]$, A will wait until B is accepted and then A will be rejected. A's delay will be (A's wasted time) + (average time for completion of B), or

$$\text{delay}[i] := \text{delay}[i] + \Pr(1)\,q[h,i]\left\{\overline{W_i} + (I_s + I_d)\,E[Y] + \overline{W_i} + I_s E[Y] + R_t + \frac{\text{pt}[h,i]}{2}\right\}. \quad (54)$$

(See equation (39). We assume that B's remaining time is uniformly distributed between 0 and pt$[h,i]$.) One RRIT and one RV will have to be repeated for A, so

$$\text{arrival rate of RRIT}[i] := \text{arrival rate of RRIT}[i] + \Pr(1)\,q[h,i]\lambda \quad (55)$$
$$\text{arrival rate of RV}[i] := \text{arrival rate of RV}[i] + \Pr(1)\,q[h,i]\lambda \quad (56)$$

On the other hand, with probability $\Pr(2)\,q[h,i]$, A will only be delayed and not rejected. In this case, we will have to vote again on A, so

$$\text{arrival rate of RV}[i] := \text{arrival rate of RV}[i] + \Pr(2)\,q[h,i]\lambda \quad (57)$$
$$\text{delay}[i] := \text{delay}[i] + \Pr(2)\,q[h,i]\left\{\overline{W_i} + I_s E[Y] + \frac{\text{pt}[h,i]}{2}\right\} \quad (58)$$

Next we consider what happens if A is deadlock rejected at node i. This happens when $i - h$ is less than i, and in this case something curious occurs. Update B has received h OK votes before node i, one OK vote at node i and will go on to receive OK votes at nodes $i+1$, $i+2$,... $i+(N_m - h - 1)$. This is assuming, of course, that B does not conflict with any other requests. Here we assume that this is true because, as we stated earlier, the number of updates that have conflicts is small and therefore, chances are that B will complete with no delays. Also notice that update B is "ahead" of A in the daisy chain and when A arrives at node $i+1$ (after its DR at node i), it will again see B on the pending list at that node. Therefore A will get another DR vote at $i+1$. This chase will continue until B is accepted and A arrives at a node where B has been removed from the pending list. When this occurs, with probability $\Pr(1)\,q[h,i]$, A will be rejected (because its timestamps are obsolete) and with probability $\Pr(2)\,q[h,i]$, A will be able to continue.

To estimate the effects of this chase, we assume that the time before B is accepted at all nodes is pt$[h,i]/2$. Then we see how far A can go in this time. The following program segment computes the delays and the extra loads involved:

```
<< At this point, A has just started execution. "Exec-time" will
  be the time that A has been in execution. "k" is the node
  where A is currently at. >>
Exec-time := W̄_i + (I_s + I_d)E[Y] + W̄_i + I_s E[Y];
k := i; << node i >>
skip := 0; << skip will count the number of DR votes. >>
arrival rate of RRIT[i] := arrival rate of RRIT[i] + Pr(1) q[h,i]λ ;
arrival rate of RV[i] := arrival rate of RV[i] + Pr(1) q[h,i]λ ;
<< Now A has been deadlock rejected for the first time. >>
Remaining-time := pt[h,i]/2;
While remaining-time > 0 do
    begin
    << A advances to next node. >>
    k := k + 1; << modulo N >>
    skip := skip + 1;
    exec-time := exec-time + T + W̄_k + I_s E[Y];
    arrival rate of RV[k] := arrival rate of RV[k] + Pr(1) q[h,i]λ ;
    remaining-time := remaining-time - (T + W̄_k + I_s E[Y]);
    end;
<< With prob. Pr(1) q[h,i], A was rejected at last vote, so: >>
delay[i] := delay[i] + Pr(1) q[h,i](exec-time + T + R_t );
<< With prob. Pr(2) q[h,i], A will continue. Update A wasted
    skip - i votes, so A will have to go past node i + N_m - 1 in
    order to obtain its N_m votes. Therefore, >>
delay[i] := delay[i] + Pr(2) q[h,i] (exec-time - W̄_i - (I_s + I_d) E[Y]);
For l:= (i + N_m - 1) + 1 until (i + N_m - 1)+skip do
    arrival rate of RV[l] := arrival rate of RV[l] + Pr(2) q[h,i]λ ;
```

Up to now we have only considered the case where A conflicts with B at A's originating node i. Now we consider the case where A conflicts with pending request B at nodes $i+1$, $i+2$, etc. In the previous case, A could conflict at i with requests originating from several nodes. However, in this case A can only conflict with pending request B at node $i+g$ if B originated at node $i+g$ (for $1 \le g \le N_m - 1$). To see this, imagine that B did not originate at node $i+g$. Then, if it is in the pending list at $i+g$, it must also be on the pending

list at $i+g-1$ and A would have conflicted with B at that node and not at $i+g$. This argument is only valid under the normal operational conditions we are considering, where no updates or messages are delayed in the communication lines.

The analysis of this case is similar to the previous one. The details are shown in appendix 4 where we present a complete program to compute the average response time. (See last part of procedure "Conflicts".) However, there is one special case we must consider. This occurs when the number of nodes, N, is even and when A conflicts with a pending request on its last vote at node $i+N_m-1$. This is a special case because if A is delayed at that node, it holds N_m-1 votes and the request it conflicted with, B (origin at $i+N_m-1$), will not be able to obtain a majority of votes. (Majority in this case is $N_m=N/2+1$.) We analyze this special case in a separate way.

First, we estimate the probability that this special case occurs. We do this through an example. Figure 4.6 shows the execution of update A which started at node 0 of a four node system. Below this, is the execution of B, the conflicting request that originated at node 2. The relative position of the time axis of B shown in figure 4.6 is not important; as a matter of fact, it helps if one views B's axis as being able to slide back and forth with respect to A's axis.

We wish to find the time interval where B could arrive in order for the conflict to occur. There are two conditions that must be met if A is to conflict with pending request B at node 2. First, B must be on the pending list. That is, B's first vote at 2 must have occurred before A arrived at 2; otherwise A could complete. This first condition implies that point P_1 must precede point P_4 in time: $P_1 > P_4$. (See figure 4.6.) The second condition is that A originates at node 0 before B arrives there; otherwise B completes before A ever gets to node 2. This condition means that $P_2 > P_3$. Since $P_4 = P_3 + D_A$ and $P_2 = P_1 + D_B$ (see figure 4.6), our two conditions imply that

$$P_3 - D_B < P_1 < P_3 + D_A.$$

The probability of the special case occurring is the probability of a conflict between A and B ($Pr(C'')$) times the number of possible B "candidates". Since updates like B originate at node 2 at a rate of λ per second, an estimate for the number of candidates is $\lambda(D_A + D_B)$ and the probability of a special case is $Pr(C'')\lambda(D_A + D_B)$. From this example, we can generalize and find that the probability of the special case is

Figure 4.6

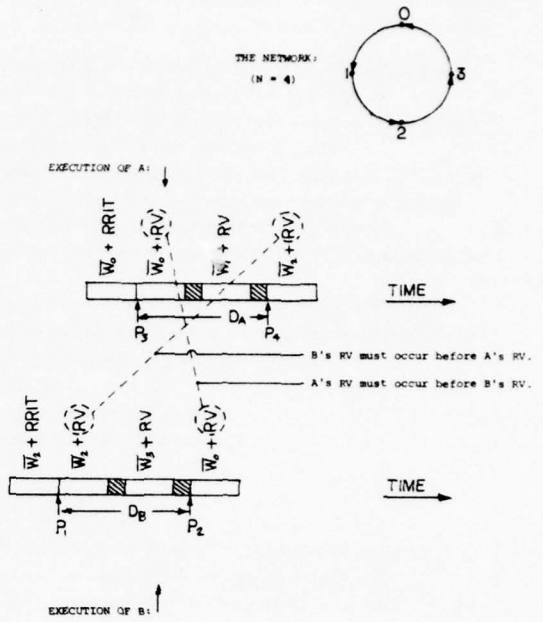


Figure 4.6. Example for the special case.
(Update A originates at node 0; update B originates at node 2.)

$$\Pr(S) = \Pr(C'')\lambda \sum_{k=0}^{N-1} \left(\overline{W_k} + I_s \mathrm{E}[Y] + T\right). \tag{59}$$

The effects of the conflict depend on whether A is delayed or deadlock rejected at node $i + N_m - 1$. If A has a higher priority than B (i.e., $i \leq i + N_m - 1$), then A will be delayed at node $i + N_m - 1$ until B gets its $N_m - 1$ OK votes and its $N_m - 1$ DR votes and B is rejected. After voting again at node $i + N_m - 1$, A will be able to continue. We assume that B's remaining time to rejection is uniformly distributed between 0 and the time for N votes ($N_m - 1 + N_m - 1$), so that

$$\begin{aligned}\text{delay}[i] := \text{delay}[i] + \Pr(S)\Bigg\{&\frac{1}{2}\sum_{k=0}^{N-1}\left(\overline{W_k} + I_s \mathrm{E}[Y] + T\right) \\ &+ \overline{W_{i+N_m-1}} + I_s \mathrm{E}[Y]\Bigg\}.\end{aligned} \tag{60}$$

The extra IO load is

$$\text{arrival rate of RV}[i+N_m-1] := \text{arrival rate of RV}[i+N_m-1] + \Pr(S)\lambda. \tag{61}$$

On the other hand, if A has lower priority than B ($i > i + N_m - 1$), update A will be deadlock rejected at node $i + N_m - 1$ and at the following nodes until A is rejected. In this case,

$$\begin{aligned}\text{delay}[i] := \text{delay}[i] + \Pr(S)\Bigg\{&\overline{W_i} + (I_s + I_d)\,\mathrm{E}[Y] \\ &+ \sum_{k=0}^{N-1}\left(\overline{W_k} + I_s \mathrm{E}[Y] + T\right) + R_t\Bigg\}\end{aligned} \tag{62}$$

$$\text{arrival rate of RRIT}[i] := \text{arrival rate of RRIT}[i] + \Pr(S)\lambda \tag{63}$$

$$\text{arrival rate of RV}[k] := \text{arrival rate of RV}[k] + \Pr(S)\lambda, \quad \text{for } k := 0, 1, 2, \ldots N-1. \tag{64}$$

This completes the analysis of the special case.

We have now computed the effects of the most important types of conflicts. The effects are of two types. First, the arrival rates of RRIT and RVs at each

node have increased. This means that the average wait time at each node must be recomputed (eq. 35). Secondly, we have computed the expected value of the delay of a single request (delay[i]). Therefore, this value must be added to the average response time of requests that originate at node i (eq. 36). After performing these two steps, we have new estimates for the average wait times and the average response time. This procedure can be repeated until the values converge. The details of this procedure are shown in the program of appendix 4. The result of executing this procedure is the average response time for updates in the distributed voting algorithm. (The comments on convergence and saturation of section 4.2.4 also apply to the DVA algorithm.)

In the next chapter we will make some improvements on the analyses we have presented here. Then, in chapter 6, we present the performance results for the update algorithms.

CHAPTER 5

COMPARISON OF THE ANALYSIS AND SIMULATION TECHNIQUES

In this chapter we compare the simulation and analysis techniques that were presented in chapter 4. In section 1 we compare the performance results that were obtained for the MCLA and DVA algorithms. In section 2 we describe some refinements to the MCLA analysis that are intended to improve the accuracy of the results. Then in section 3 we present the results obtained with the refined analysis and we compare these results to the MCLA simulation results. The reasons why even the refined analysis results differ from the simulation results in certain parameter ranges are given in section 4. In section 5 we show how the analysis of the DVA algorithm can also be improved. Finally, in section 6, we briefly mention the advantages and disadvantages of each technique.

1. COMPARISON OF THE MCLA RESULTS.

The analytic technique for studying the performance of the MCLA and the DVA algorithms is divided in two parts. (See chapter 4). Initially, the algorithms are analyzed assuming that no conflicts among the updates occur and then the conflicts are taken into consideration. In order to check both parts of the analysis, we initially compare the results of the first part of the analysis with special modified simulators that produce no conflicts among the update transactions. Then, in section 1.2, we study the results of the second part of the analysis in relation to the results produced by the original simulators.

1.1 Comparison of the Results When No Conflicts Occur.

The simulators of the MCLA and the DVA algorithms were modified so that no conflicts occurred among the updates. This was simply done by making the

outcome of the test for conflicts be always false. No other parts of the simulators were changed.

The results obtained from the modified simulators for the case of 6 nodes are shown in table 5.1. In this table, $\overline{R_s}$ is the average response time for updates, s^2 is the sample variance of the observed response times and n is the number of observations. For the MCLA algorithm, the IO utilization at the central node is given, while for the DVA algorithm, the average IO utilization at the nodes is given. The results obtained from the first part of the analysis are also shown in this table ($\overline{R_a}$). The column labeled "%DIFF." gives the difference between the simulation average response time and the analytic average response time as a percentage of the simulation average response time.

To give an idea of the accuracy of the simulation results, we compute the 90 percent confidence interval for the results shown in table 5.1. (See [FREU71].) First we assume that the n samples taken from the simulation are n independent samples from a distribution with mean μ and standard deviation σ. (The samples are not quite independent because the value of a sample may affect some of the other values. That is, the samples may be autocorrelated. However, when the number of samples is large, we can assume that the samples are independent and obtain satisfactory results; see [GORD78].) Statistic $\overline{R_s}$ is the average of these n samples and we are interested in knowing how close to the true mean, μ, our estimate $\overline{R_s}$ is. From the central limit theorem [FREU71], we know that the distribution of the sample mean (of which $\overline{R_s}$ is a sample) can be considered normal for a large number of samples regardless of the original distribution. Therefore, we can obtain the 90 percent confidence interval for $\overline{R_s}$ as:

$$\left(\overline{R_s} - Z_{.05}\frac{\sigma}{\sqrt{n}} \quad , \quad \overline{R_s} + Z_{.05}\frac{\sigma}{\sqrt{n}}\right) \tag{1}$$

where constant $Z_{.05}$ is 1.65. Since we do not know the standard deviation, σ, we must approximate this as the square root of the sample variance s^2 obtained from the simulation. Thus, the 90 percent confidence interval is approximately

$$\left(\overline{R_s} - 1.65\sqrt{\frac{s^2}{n}} \quad , \quad \overline{R_s} + 1.65\sqrt{\frac{s^2}{n}}\right) \tag{2}$$

This interval can be interpreted as follows: If we run the simulation for a given case 100 different times (each time starting from different initial conditions), then 90 out of the 100 times, the interval given above will include the true mean

Table 5.1

TABLE 5.1

COMPARISON OF ANALYTIC AND SIMULATION RESULTS.
--- NO CONFLICTS CASE ---

N = 6, Bs = 5, M = 1000, T = .1, Rt = 1, Is = Id = 0.025.

MCLA Centralized Locking Algorithm.

Ar	$\overline{Ra}$	$\overline{Rs}$	s^2	n	Utilization†	% Diff.*	90% C. I.**
15	0.769	0.768	0.385	8956	0.284	-0.13	1.25
10	0.829	0.834	0.399	10740	0.310	+0.60	1.21
7	0.936	0.951	0.577	7630	0.432	+1.58	1.51
5	1.194	1.248	1.231	9655	0.613	+4.33	1.49
4	1.747	1.852	3.256	10742	0.765	+5.67	1.55

Distributed Voting Algorithm.

Ar	$\overline{Ra}$	$\overline{Rs}$	s^2	n	Utilization‡	% Diff.*	90% C. I.**
15	1.526	1.496	0.855	5331	0.116	-2.01	1.48
10	1.609	1.575	0.911	5331	0.174	-2.16	1.37
7	1.735	1.770	1.084	2291	0.255	+1.98	2.83
5	1.951	1.978	1.314	3157	0.351	+0.96	1.71
4	2.287	2.233	1.622	9668	0.445	+1.16	1.15

† IO utilization at central node (from simulation).

‡ Average IO utilization at all nodes (from simulation).

* % Difference = 100*($\overline{Rs}$ - $\overline{Ra}$)/$\overline{Rs}$.

** 90 % confidence interval is 1.65*sqrt(s*s/n)*100/$\overline{Rs}$.

of the response time distribution, μ. The 90 percent confidence intervals for the cases in table 5.1 are given in the last column of that table. The interval is given as a percentage of $\overline{R_s}$ in order to make comparisons with the "%DIFF." column easier.

The cases shown in table 5.1 seem to be typical cases and the differences between the simulation and the analytic results are relatively small. However, notice that in many of the cases, the analytic result $\overline{R_a}$ is not in the confidence interval. Also notice that when the utilization is greater than 0.60, the difference between the analytic and the simulation results is considerably larger. The significance of these observations will be discussed later; first we compare the techniques when update conflicts are taken into account.

1.2 Comparison of Results When Conflicts Occur.

Table 5.2 compares the simulation results to the analytic results when conflicts among updates are considered. The simulators used did not have the modification described in the previous section and the analytic results were obtained using the complete technique described in chapter 4. (See appendices 3 and 4.) The cases given in table 5.2 are for various combinations of the number of nodes N and the interarrival time A, for both the centralized and the distributed algorithms. The column heading are the same as before.

Notice that the difference between the analytic and the simulation results is now larger. The difference increases as the utilization increases. However, it still seems that in most of the cases tested (in addition to the cases shown in table 5.2), the difference was relatively small as long as the utilization at all nodes was less than 0.50. The one exception to this rule were the cases where the ratio B_t/M was larger than 0.01, as is illustrated in table 5.3. This table gives the difference between the analytic and the simulation results as M was varied while B_t was held constant. Notice that some of the differences are larger than what we would expect from table 5.2 since the utilization is less than 0.50.

The fact that the analytic results differ from the simulation results as more conflicts occur (i.e., as B_t/M increases), suggests that the analysis for conflicts is underestimating the cost of conflicts occurring. In the next section, we refine some aspects of the conflict analysis of chapter 4 in order to try to reduce the difference with the simulation results.

Table 5.2

TABLE 5.2

COMPARISON OF ANALYTIC AND SIMULATION RESULTS.
--- CONFLICTS CONSIDERED ---

Bs = 5, M = 1000, T = .1, Rt = 1, Is = Id = 0.025

MCLA Centralized Locking Algorithm.

N	Ar	$\overline{Ra}$	$\overline{Rs}$	s^2	n	Utilization†	% Diff.*	90% C. I.**
6	15	0.772	0.800	0.339	8956	0.285	+3.45	1.27
6	10	0.835	0.855	0.502	9654	0.308	+2.29	1.39
6	7	0.951	1.010	0.918	9204	0.443	+5.86	1.63
6	6	1.043	1.138	1.415	8957	0.522	+8.34	1.82
6	5	1.237	1.415	2.787	9055	0.627	+12.6	1.98
9	10	0.951	1.005	0.893	9668	0.457	+5.33	1.58
9	7	1.288	1.553	3.714	9287	0.667	+17.1	2.13

Distributed Voting Algorithm.

N	Ar	$\overline{Ra}$	$\overline{Rs}$	s^2	n	Utilization†	% Diff.*	90% C. I.**
6	15	1.548	1.537	1.089	5331	0.117	-0.72	1.53
6	10	1.646	1.675	1.605	5331	0.177	+1.73	1.71
6	7	1.796	1.871	2.263	5331	0.256	+4.01	1.82
6	5	2.068	2.229	3.508	5328	0.365	+7.58	1.90
9	15	1.892	1.903	1.817	5327	0.158	+0.58	1.60
9	7	2.378	2.529	3.913	5722	0.351	+6.29	1.71

† IO utilization at central node (from simulation).

‡ Average IO utilization at all nodes (from simulation).

* % Difference = 100*($\overline{Rs}$ - $\overline{Ra}$)/$\overline{Rs}$.

** 90 % confidence interval is 1.65*sqrt(s*s/n)*100/$\overline{Rs}$.

Table 5.3

TABLE 5.3

COMPARISON OF SIMULATION AND ANALYTIC RESULTS.
--- EFFECT OF HIGH LOCK ACTIVITY ---

Conflicts considered,
N = 6, Bs = 5, Ar = 10, T = .1, Rt = 1, Is = Id = 0.025.

MCLA Centralized Locking Algorithm.

M	$\overline{Ra}$	$\overline{Rs}$	Utilization †	% Diff.*
1000	0.835	0.855	0.308	+2.34
400	0.846	0.893	0.313	+5.26
200	0.863	0.946	0.318	+8.77
100	0.897	1.044	0.328	+14.08

Distributed Voting Algorithm.

M	$\overline{Ra}$	$\overline{Rs}$	Utilization ‡	% Diff.*
1000	1.646	1.675	0.180	+1.73
400	1.701	1.839	0.185	+7.50
300	1.732	1.898	0.187	+8.75
200	1.793	2.043	0.193	+12.24

† IO utilization at central node (from simulation).

‡ Average IO utilization at all nodes (from simulation).

* % Difference = 100*($\overline{Rs}$ - $\overline{Ra}$)/$\overline{Rs}$.

2. Conflict Analysis Revisited - The MCLA Algorithm.

In this section, we consider some "second order" effects of conflicts in the MCLA algorithm. Some of the results obtained here will also be applicable to the DVA algorithm, but in order to simply the presentation, we will consider the distributed algorithm later.

2.1 Probability of Conflict.

In chapter 4, we found that the probability that two updates conflicted (e.g., their base sets intersected) was given by

$$\Pr(C) = \frac{(\mathrm{E}[\mathrm{Y}])^2}{M} \tag{3}$$

(see equation (23) in chapter 4). The derivation of this equation did not state clearly for what values of B_s (the base set parameter) and M (the number of items) the equation was valid. We will now derive a better approximation of $\Pr(C)$ and we will investigate the ranges of B_s and M where it is valid.

Suppose that we have two updates, A and B. Let discrete random variable Y be the number of items in the base set of A and let discrete random variable X be the number of items in B's base set. Let us represent the event "A and B conflict" by C and the event "A and B do not conflict" by NC. The probabilities of these events occurring are $\Pr(C)$ and $\Pr(NC)$ respectively, and $\Pr(C)$ equals $1 - \Pr(NC)$. Update B conflicts with A if at least one item in B coincides with an item in A's base set.

First we will obtain an expression for $\Pr(NC)$ given that we know that the value of Y is i and the value of X is j. We write this expression as $\Pr(NC \mid \mathrm{Y} = i \text{ and } \mathrm{X} = j)$. The process of two updates conflicting can be viewed as sampling without replacement [FREU71]. We have a set (update A) with M elements of which i elements are labeled "success" and $M - i$ are labeled "failure". We will sample from this set j times. Each of the j times, we select an element of the set at random and if we have a success then we have a conflict. A particular element of the set (i.e., an item) cannot be selected more than once (i.e., no replacement). That is, we are selecting a subset of j elements and we are interested in the probability that all elements selected are failures ($\Pr(NC)$).

There are $\binom{M}{j}$ ways in which a subset of j elements can be chosen from a set of M elements and each choice has a probability of $1/\binom{M}{j}$. Recall that $\binom{M}{j}$

is the number of combinations of j objects selected from a set of M objects and is given by

$$\binom{M}{j} = \frac{M!}{j!\,(M-j)!} \tag{4}$$

The number of subsets (of j elements) that have no successes is the number of ways in which we can choose the j elements from the $M - i$ failures, or $\binom{M-i}{j}$. Therefore the probability of not getting a conflict is given by

$$\Pr(NC \mid \mathrm{Y} = i \text{ and } \mathrm{X} = j) = \frac{\binom{M-i}{j}}{\binom{M}{j}} \tag{5}$$

To compute the probability of no conflict $\Pr(NC)$ from $\Pr(NC \mid \mathrm{Y} = i \text{ and } \mathrm{X} = j)$, we use the so called "rule of elimination" [FREU71]:

$$\Pr(NC) = \sum_{j=1}^{M} \sum_{i=1}^{M} \Pr(\mathrm{Y} = i) \Pr(\mathrm{X} = j \mid \mathrm{Y} = i) \Pr(NC \mid \mathrm{Y} = i \text{ and } \mathrm{X} = j) \tag{6}$$

Since X and Y are independent, $\Pr(\mathrm{X} = j \mid \mathrm{Y} = i)$ is $\Pr(\mathrm{X} = j)$. Taking into account the limits for i and j, and equation (5), we obtain:

$$\Pr(NC) = \sum_{j=1}^{M} \sum_{i=1}^{M-j} \Pr(\mathrm{Y} = i) \Pr(\mathrm{X} = j) \frac{\binom{M-i}{j}}{\binom{M}{j}} \tag{7}$$

In chapter 4 we showed that

$$\Pr(\mathrm{Y} = i) = \Pr(\mathrm{X} = i) = \left(\frac{1-a}{a}\right) a^i \tag{8}$$

where $a = \exp(-1/B_s)$. Using this fact and equation (4), we get:

$$\Pr(NC) = \frac{(1-a)^2}{a^2} \sum_{j=1}^{M} a^j \sum_{i=1}^{M-j} a^i \frac{(M-i)(M-i-1)\ldots(M-i-j+1)}{M(M-1)(M-2)\ldots(M-j+1)} \tag{9}$$

Table 5.4

TABLE 5.4

COMPARISON OF P(C) WITH ITS ESTIMATE.

Bs	M	$\frac{(E[Y])^2}{M}$	P(C) †	% Diff.*
1	2000	0.00125	0.00125	0.01
1	1000	0.00250	0.00250	0.00
1	200	0.01251	0.01247	0.30
1	100	0.02502	0.02486	0.64
1	20	0.1251	0.1209	3.47
1	10	0.2502	0.2334	7.20
5	10000	0.00304	0.00303	0.32
5	5000	0.00608	0.00604	0.72
5	1000	0.03043	0.02928	3.93
5	500	0.06086	0.05644	7.83
5	100	0.3043	0.2236	36.08
5	50	0.6086	0.3629	67.70
10	20000	0.00552	0.00547	0.81
10	10000	0.01104	0.01085	1.71
10	2000	0.05521	0.05082	8.62
10	1000	0.1104	0.09453	16.81
10	200	0.5521	0.3181	73.56
10	100	1.1042	0.4683	135.75
20	40000	0.01051	0.01032	1.79
20	20000	0.02102	0.02027	3.65
20	4000	0.1051	0.08936	17.61
20	2000	0.2102	0.1573	33.61
20	400	1.0510	0.4352	141.50
20	200	2.1021	0.5839	260.00

† Computed using equations (9) and (10) of chapter 5.

* Computed as *100*(estimate - P(C))/P(C).

Once we evaluate equation (9), we can find P(C) simply as

$$\Pr(C) = 1 - \Pr(NC) \tag{10}$$

It seems hard to find a closed form expression for equation (9), so we evaluate it and equation (10) using a numerical method. A program to evaluate with high accuracy equations (9) and (10) was written [GARC78]. Table 5.4 compares the values of $\Pr(C)$ obtained by means of the program with the estimated values given by equation (3). From these results it seems that equation (3) is a good approximation to $\Pr(C)$ only when $(E[Y])^2/M$ is less than 0.05.

Notice that the values of $\Pr(C)$ given in table 5.4 are in all cases less than the estimated values of equation (3). This is unfortunate since this will increase the difference between the analytic and the simulation results in table 5.3. For cases with $(E[Y])^2/M < 0.05$, the decrease in $\Pr(C)$ is very small, so that the differences in table 5.3 will hardly increase. However, when $(E[Y])^2/M > 0.05$, the increased deviation of the simulation and analytic results will be noticeable. We will discuss the implications of this later.

2.2 Size of the Base Set Given Conflict.

In chapter 4, we assumed that an update that conflicted had average characteristics. In particular, we assumed that the size of its base set was given by

$$E[Y] = (1 - \exp(-1/B_s))^{-1} = (1 - a)^{-1} \tag{11}$$

(where $a = \exp(-1/B_s)$) and this value was used in the rest of the analysis. However, since we know that the update has conflicted, we expect the base set to be larger than average. In this section, we derive an expression for the average base set of an update given that it has conflicted. To do this, we assume that $(E[Y])^2/M$ is less than 0.05.

Suppose that update A has arrived at the central node to request locks and it has conflicted with update B. As in the previous section, we let discrete random variable Y be the number of items in A's base set and discrete random variable X be the number in B's base set. Also let C be the event "update A conflicted with B". We are interested in the expected value of Y given event C, i.e., $E[Y \mid C]$. We compute this from the following equation:

$$E[Y \mid C] = \sum_{i=1}^{\infty} i \Pr(Y = i \mid C). \tag{12}$$

The expression $\Pr(Y = i \mid C)$ is the probability that A has i items in its base set given that it has conflicted. To compute $\Pr(Y = i \mid C)$, we use Bayes' rule [FREU71]:

$$\Pr(Y = i \mid C) = \frac{\Pr(Y = i)\Pr(C \mid Y = i)}{\sum_{k=1}^{\infty} \Pr(Y = k)\Pr(C \mid Y = k)} \tag{13}$$

The denominator in equation (13) is the probability of a conflict $\Pr(C)$. The value of $\Pr(Y = i)$ is $(1 - a)a^i/a$. (See equation (8).)

To compute $\Pr(C \mid Y = i)$ for equation (13), we must know how many items are in update B (that is the update A has conflicted with). In other words, we know $\Pr(C \mid Y = i)$ given that X, the number of items in B, is j. This last expression is $\Pr(C \mid Y = i \text{ and } X = j)$. Since $(E[Y])^2/M$ is less than 0.05, we approximate this by:

$$\Pr(C \mid Y = i \text{ and } X = j) = \frac{ij}{M} \tag{14}$$

This equation was used in chapter 4 to derive the approximation to $\Pr(C)$ (equation (3)). Since that approximation was good when $(E[Y])^2/M$ was less than 0.05, we expect equation (14) to be a good approximation for this case too. (In other words, if $(E[Y])^2/M < 0.05$, then for the most probable values of i and j, equation (14) will be a good approximation.)

From equation (14), we compute $\Pr(Y = i \mid C)$ as follows:

$$\Pr(C \mid Y = i) = \sum_{j=1}^{\infty} \Pr(C \mid Y = i \text{ and } X = j)\Pr(X = j), \tag{15}$$

or

$$\Pr(C \mid Y = i) = \frac{i\,E[X]}{M} \tag{16}$$

where E[X] is of course given by

$$E[X] = E[Y] = \sum_{j=1}^{\infty} j \Pr(X = j). \tag{17}$$

Now we substitute equation (16) into equation (13) and obtain

$$\Pr(Y = i \mid C) = \frac{\Pr(Y = i)i\,E[X]/M}{\sum_{k=1}^{\infty} \Pr(Y = k)k\,E[X]/M} \tag{18}$$

or

$$\Pr(Y = i \mid C) = \frac{i \Pr(Y = i)}{E[Y]} \tag{19}$$

Substituting into equation (12), we obtain the desired result:

$$E[Y \mid C] = \frac{\sum_{i=1}^{\infty} i^2 \Pr(Y = i)}{E[Y]} = \frac{E[Y^2]}{E[Y]}, \quad \left((E[Y])^2/M < .05\right). \tag{20}$$

Using the value of E[Y] given in equation (11) and the value of $E[Y^2]$ given by equation (9) in chapter 4, we can also write this as

$$E[Y \mid C] = \frac{1 + a}{1 - a}. \tag{21}$$

In a similar fashion we can obtain the second moment of Y,

$$E[Y^2] = \frac{6a^2}{(1 - a)^2} + \frac{6a}{(1 - a)} + 1, \quad \left((E[Y])^2/M < .05\right). \tag{22}$$

(See [GARC78] for details.)

Incidentally, notice that the denominator of equation (18) is the probability of a conflict P(C). From equation (17), we find that

$$\Pr(C) = \frac{E[Y]\,E[X]}{M} \tag{23}$$

which is exactly the approximation of equation (3) for the case $(E[Y])^2/M < 0.05$. It is also possible to show that $E[X \mid C] = E[Y \mid C]$. That is, equations (21) and (22) not only apply to the update that conflicted, but also to the update it conflicted with.

To give an idea of the difference between equations (21) and (11), we give an example. A reasonable value for the base set parameter, B_s, is 5 items. Then E[Y] is 5.52 while $E[Y \mid C]$ is 10.03. That is, the average base set of the updates that conflict is almost twice the size of the average base set of all updates.

Notice that in equations (21) and (22), M, the total number of items, does not appear. This is to be expected, since these equations are only valid for the case where M is very large compared to E[Y] (i.e., $(E[Y])^2/M < 0.05$). As the

value of M decreases and B_s is held constant, we expect the real value of $E[Y \mid C]$ to decrease and approach $E[Y]$ because more and more updates with smaller base sets will conflict as M decreases.

If discrete random variable Z is the number of items in an update's write set, then the mean and the second moment of Z given that a conflict occurred are given by

$$E[Z \mid C] = \frac{E[Y \mid C] + 1}{2} \quad \text{and} \tag{24}$$

$$E[Z^2 \mid C] = \frac{E[Y^2 \mid C]}{3} + \frac{E[Y \mid C]}{2} + \frac{1}{6}. \tag{25}$$

The details are given in [GARC78].

The values of $E[Y]$, $E[Y^2]$, $E[Z]$ and $E[Z^2]$ were used in chapter 4 for two purposes: to compute the probability of conflict and to estimate the cost of conflicts (delays and extra IO time). Now that we have the values of $E[Y \mid C]$, $E[Y^2 \mid C]$, $E[Z \mid C]$ and $E[Z^2 \mid C]$, we use these new values instead of the original ones to compute the cost of conflicts because these computations refer only to updates that have conflicted. However, when computing the probability of conflict or when dealing with updates which we do not know have conflicted, we still use the original values.

2.3 Other Improvements to the Conflict Analysis.

In order to try to reduce the gap between the simulation and the analytic results, we now consider some other "second order" effects. We expect the improvements (or deterioration) to be small, but in any case it is important to make sure that these changes are as small as we suspect. We still assume that $(E[Y])^2/M$ is less than 0.05.

For example, in chapter 4 we approximated the average time that an update A had to wait for a locked item by $E[L]/2$ (or $\overline{L}/2$), where $E[L]$ was the time that an average update held its locks. (See equations (27) and (32) in chapter 4.) If update A is waiting for a lock, the lock is held by an update, B, that A conflicted with, so instead of $E[L]/2$ we should now use $E[L \mid C]/2$, where $E[L \mid C]$ is the average lock time given that a conflict has occurred. The value of $E[L \mid C]$ is computed by using $E[Y \mid C]$ and $E[Z \mid C]$ instead of $E[Y]$ and $E[Z]$ in equations (25), (26) and (27) in chapter 4:

$$E[L_{nc} \mid C] = T + \overline{W_{nc}} + I_d E[Y \mid C] + T + \overline{W_c} + I_s E[Y \mid C] + I_d E[Z \mid C] \tag{26}$$

$$E[L_c \mid C] = \overline{W_c} + I_d E[Y \mid C] + \overline{W_c} + I_s E[Y \mid C] + I_d E[Z \mid C] \tag{27}$$

$$E[L \mid C] = \left(\frac{N-1}{N}\right) E[L_{nc} \mid C] + \left(\frac{1}{N}\right) E[L_c \mid C]. \tag{28}$$

This estimate of $E[L \mid C]$ assumes that no other update is waiting for the item, i.e., that A will be able to continue once the item "i" is free. However, there might be a third update A′ waiting for item i with a higher priority than A. Therefore, the lock wait time for A will be larger than $E[L \mid C]$ on the average.

The expected value of this extra delay is the probability that A′ is waiting with a higher priority times the extra time that A′ will keep item i locked. The last term can be approximated by $E[L \mid C]$. The probability that A′ is waiting is the probability that an update A′ arrived between the time B arrived and the time A arrived and that update A′ referenced item i. The probability that i is in the base set of A′ is approximately $E[Y]/M$, and the time interval between the arrival of B and A is on the average $E[L \mid C]/2$. Since updates like A′ arrive at a rate of $N\lambda$, the probability that the extra delay occurs is

$$\frac{E[L \mid C]}{2} N\lambda \frac{E[Y]}{M} \tag{29}$$

Thus, the expected value of the delay is

$$E[L \mid C]\left\{\frac{E[L \mid C]}{2} N\lambda \frac{E[Y]}{M}\right\} \tag{30}$$

and a better approximation to the average time that an item remains locked is

$$E[L \mid C]_{new} = E[L \mid C]_{old} + \frac{\left(E[L \mid C]_{old}\right)^2 N\lambda E[Y]}{2M} \tag{31}$$

This new value can now be used to compute the delays caused by a conflict.

A second improvement on the conflict analysis involves the probability that an update waits. In chapter 4, we stated that the probability that an arriving update had to wait for locks at the central node was

$$Pr(W_1) = Pr(C)\overline{J}$$

where $\overline{J} = N\lambda E[L]$. Recall that $\overline{J}$ is the average number of updates that are holding locks and that are not waiting themselves. This equation does not take

into account all the locks that are being held by waiting updates. Some updates may be waiting for one lock while holding other locks. On the average, these updates have locked

$$\frac{\mathrm{E}[Y \mid C]-1}{2} \tag{32}$$

items. (The details of this derivation can be found in [GARC78].) The probability that an arriving update conflicts with one of these waiting updates is

$$\Pr(C_2)=\frac{\mathrm{E}[Y](\mathrm{E}[Y \mid C]-1)/2}{M} \tag{33}$$

(The derivation is similar to the one for equation (23). Assume that $\mathrm{E}[Y](\mathrm{E}[Y \mid C]-1)/(2M)$ is also less than 0.05.) The probability that an arriving update has to wait due to other waiting updates is

$$\Pr(W_2)=\Pr(C_2)\overline{J'} \tag{34}$$

where $\overline{J'}$ is the average number of updates that are waiting at the central node. Using Little's formula, $\overline{J'}$ is the product of the arrival rate of updates that wait times the average time these updates wait. The arrival rate is approximately $N\lambda\Pr(W_1)$. The average wait time is the average time that the item we are waiting on remains locked plus the time needed to lock it and the remaining items, i.e.,

$$\frac{\mathrm{E}[L \mid C]_{new}}{2}+\overline{W_c}+I_s(\mathrm{E}[Y \mid C]-1) \tag{35}$$

seconds, where $\mathrm{E}[L \mid C]_{new}$ is defined in equation (31). Therefore,

$$\overline{J'}=\left\{\frac{\mathrm{E}[L \mid C]_{new}}{2}+\overline{W_c}+I_s(\mathrm{E}[Y \mid C]-1)\right\}N\lambda\Pr(W_1). \tag{36}$$

This value can be substituted into equation (34) to obtain the probability that an arriving update has to wait at the central node due to a conflict with a waiting update. The total probability of an update waiting is then

$$\Pr(W)=\Pr(W_1)+\Pr(W_2).$$

The delay due to conflicts is greater now since with probability $\Pr(W_2)$ an update has to wait for two other updates to finish. The expected value of the

delays (considering both types of conflicts) is

$$\text{delay}=\Pr(W)\left\{\frac{\mathrm{E}[L \mid C]_{new}}{2}+\overline{W_c}+I_s(\mathrm{E}[Y \mid C]-1)\right\}+\Pr(W_2)\,\mathrm{E}[L \mid C] \tag{37}$$

(Compare to equation (32) in chapter 4.) There is no extra IO service time involved in this additional delay.

The last refinement we consider is the case where an update A waits for one item and then, after getting that lock, conflicts again with a third update B'. The average and the second moment of the number of items remaining to be locked after a conflict are

$$\begin{aligned}\mathrm{E}[\mathrm{REM} \mid C]&=\frac{\mathrm{E}[Y \mid C]-1}{2}\\ \mathrm{E}[\mathrm{REM}^2 \mid C]&=\frac{\mathrm{E}[Y^2 \mid C]}{3}-\frac{\mathrm{E}[Y \mid C]}{2}+\frac{1}{6}\end{aligned} \tag{38}$$

(See [GARC78].) Following the derivation of $\Pr(W_1)$, we estimate the probability of a second wait at the central node given that a first update has taken place by

$$\Pr(\text{2nd wait})=\frac{(\mathrm{E}[Y \mid C]-1)\mathrm{E}[Y]}{2M}N\lambda\,\mathrm{E}[L]. \tag{39}$$

The expected increase in delay due to this second conflict is

$$\Pr(W)\Pr(\text{2nd wait})\left\{\frac{\mathrm{E}[L \mid C]_{new}}{2}+\overline{W_c}+I_s\left(\frac{\mathrm{E}[\mathrm{REM} \mid C]-1}{2}\right)\right\}$$

The first term is the probability of the first conflict. The average number of items that remain to be locked after the second wait is $(\mathrm{E}[\mathrm{REM} \mid C]-1)/2$. The expression above should be added to equation (37) to obtain the total expected delay.

Since after the second wait, $(\mathrm{E}[\mathrm{REM} \mid C]-1)/2$ items will have to be locked, the mean and the second moment of the extra IO service time are

$$2I_s\left(\frac{\mathrm{E}[\mathrm{REM} \mid C]-1}{2}\right)$$

and

$$4I_s^2\left(\frac{\mathrm{E}[\mathrm{REM}^2 \mid C]}{3}-\frac{\mathrm{E}[\mathrm{REM} \mid C]}{2}+\frac{1}{6}\right).$$

(See [GARC78].) These extra loads must be taken into account when computing the average wait time at the central node.

3. COMPARISON OF NEW RESULTS FOR CENTRALIZED ALGORITHM.

The program to compute the average response time of an update in the MCLA algorithm (appendix 3) is now modified to include the results obtained in section 2. The complete listing of the modified program is given in appendix 5. The results of this program are compared to the simulation results in table 5.5. As can be seen by comparing this table to tables 5.2 and 5.3, the difference between the simulation and the analytic results has been reduced somewhat by the refined analysis of the previous sections. (The reduction ranges from 0.5 to 8 percent of the original difference.) The improvement is noticeable but not really significant.

The program of appendix 5 assumes that $(E[Y])^2/M$ is less than 0.05. However, notice that this is not true for the cases of table 5.5 where M is less than 500. As was mentioned earlier, when $(E[Y])^2/M$ is greater than 0.05, the values for the probability of conflict $Pr(C)$ (equation (3)) and the expected size of the base set given a conflict $E[Y \mid C]$ (equation (21)) are actually larger than the true values. Therefore, the values of $\overline{R_a}$ in table 5.5 when $(E[Y])^2/M > 0.05$ are larger than the values that would be produced by a more accurate analysis. This means that the differences between the analytic and the simulation results shown in table 5.5 for these cases are actually smaller than the difference we would obtain by using the correct values of $Pr(C)$ and $E[Y \mid C]$. In other words, the results of table 5.5 are deceiving because they make the analysis results look close to the simulation results even in the case where $(E[Y])^2/M$ is greater than 0.05.

The improvement (i.e., the reduction in the difference between the analytic and the simulation results) we have obtained by considering some of the "second order" effects has been small. It seems that we will be unable to obtain any significant improvements by considering any other similar "second order" effects. Thus, the analytic results of table 5.5 are as accurate as is possible to obtain from a simple analysis. Considering the confidence intervals shown in table 5.2, the simulation results seem to be very stable. Both results agree if $(E[Y])^2/M$ is less than 0.05 and if the maximum utilization is less than about 0.60. But they do not agree in the other cases. What results are the ones that are incorrect for $(E[Y])^2/M > 0.05$ or for the maximum utilization > 0.60? We address this question in the following section.

Table 5.5

TABLE 5.5

COMPARISON OF SIMULATION RESULTS TO RESULTS OF MODIFIED ANALYSIS.
--- CENTRALIZED LOCKING ALGORITHM ---

Bs = 5, M = 1000, T = .1, Is = Id = 0.025.

N	Ar	$\overline{Ra}$	$\overline{Rs}$	Utilization †	% Diff. *
6	15	0.775	0.800	0.285	+3.13
6	10	0.839	0.855	0.308	+1.87
6	7	0.959	1.010	0.443	+5.05
6	6	1.057	1.138	0.522	+7.12
6	5	1.266	1.415	0.627	+10.53
9	10	0.961	1.005	0.457	+4.38
9	7	1.327	1.553	0.667	+14.55

N = 6, Bs = 5, Ar = 10, T = .1, Is = Id = 0.025.

M	$\frac{(E[Y])^2}{M}$	$\overline{Ra}$	$\overline{Rs}$	Utilization †	% Diff. *
1000	0.03	0.839	0.855	0.308	+1.87
400	0.07	0.857	0.893	0.313	+4.03
200	0.15	0.892	0.946	0.318	+5.71
100	0.30	0.980	1.044	0.328	+6.13

† Utilization at central node (from simulation).

* % difference computed as $100*(\overline{Rs} - \overline{Ra})/\overline{Rs}$.

4. SIMULATION RESULTS VS ANALYTIC RESULTS.

There is a possibility that either the analytic or the simulation results are off when $(E[Y])^2/M$ is greater than 0.05 or when the maximum utilization is greater than 0.60.

The analytic results could be wrong simply because the analysis we have performed is just too simple to take into account the cases of high IO load or high lock activity. For example, when the IO load is high, our assumption that the arrival of all IO requests to a node is a Poisson process is not true and this may affect the results. (See chapter 4.) This assumption is a critical one since it permits the decomposition of the nodes into independent queueing systems. Similarly, when the mean base set of updates is large compared to the total number of items, the conflict analysis might not be accurate because it is based on average values instead of on the dynamic values of system variables. In particular, as $(E[Y])^2/M$ grows, the probability that at a given time a few updates will monopolize most of the locks increases. In these congestion periods, the queues of waiting updates will grow and the response time of updates will be considerably greater. The simplified conflict analysis we used did not take into account these types of congestions.

On the other hand, it is also possible that the simulation results are off even though the confidence intervals are small. The cause of this could be an initial transient that dies out very slowly and produces a bias in the results. However, the fact that the simulation results are fairly stable with respect to run time (after a clear initial transient) indicates that any such bias must be small. Furthermore, all the simulation runs show consistent differences with the analytic results. That is, in all cases, as $(E[Y])^2/M$ increases past 0.05 or the utilization increases past 0.60, the simulation average response time is increasingly greater than the analytic average response time. This rules out statistical variations as the probable cause of the difference.

In view of the above discussion, we decide that it is the analytic results that deviate from the true results when $(E[Y])^2/M > 0.05$ or when the maximum utilization is greater than 0.60. However, in most cases of interest, we expect $(E[Y])^2/M$ to be considerably less than 0.05 because most updates only reference a minimal fraction of the database. The fact that the analytic results are not as accurate when the utilization is greater than 0.60 is more serious. But on the other hand, obtaining simulation results when the utilization is greater than 0.60 is expensive in terms of computer time, so that in most heavy load cases we will have to be satisfied with the analytic results.

5. COMPARISON OF NEW RESULTS FOR THE DVA ALGORITHM.

The results obtained for the probability of conflict, $Pr(C)$, and for the expected size of the base set given that a conflict has occurred can also be applied to the distributed voting algorithm. Since the improvements obtained for the MCLA algorithm were not very impressive, we do not consider any additional "second order" effects for the distributed algorithm.

The program to compute the average response time in the distributed voting algorithm (appendix 4) has been modified to take into account the larger base sets of updates that have conflicted. Since we assume that $(E[Y])^2/M < 0.05$, we still use equation (3) as an estimate of $Pr(C)$. The modified program is shown in appendix 6 and its results are compared to the simulation results in table 5.6. Just as expected, the reduction of the difference between the analytic and the simulation results is small. As in the case of the MCLA centralized algorithm, the difference for the cases where $(E[Y])^2/M > 0.05$ is actually greater than shown in table 5.6.

6. ADVANTAGES OF EACH TECHNIQUE.

To summarize the findings of this chapter, we list some of the advantages and disadvantages of the simulation and analytic techniques. No technique is the better of the two. They actually complement each other and it is best to have both techniques available for studying the update algorithms.

The Analytic Technique – Advantages.

- Little computation time is required to obtain results. This is true even for the cases of high IO utilization or high lock activity. With more results available, results are easier to plot.
- Provides a good understanding of the operation of the system.

The Analytic Technique – Disadvantages.

- Results are not very accurate if $(E[Y])^2/M$ is greater than 0.05 or if the maximum utilization is greater than 0.60.
- Analysis becomes very complex if we wish to change some assumptions (e.g., change distribution of number of items in base set).

The Simulation Technique – Advantages.

- Produces fairly accurate results for all cases.
- Helpful in understanding the operation of the algorithms.
- Simulations are flexible. Once a simulation is written, it is easy to vary

Table 5.6

TABLE 5.6

COMPARISON OF SIMULATION RESULTS TO RESULTS OF MODIFIED ANALYSIS.
--- DISTRIBUTED VOTING ALGORITHM ---

Bs = 5, M = 1000, T = .1, Rt = 1, Is = Id = 0.025.

N	Ar	$\overline{Ra}$	$\overline{Rs}$	Utilization †	% Diff. *
6	15	1.556	1.537	0.117	-1.24
6	10	1.659	1.675	0.177	+0.96
6	7	1.819	1.871	0.256	+2.78
6	5	2.105	2.229	0.365	+5.56
9	15	1.907	1.903	0.158	-0.21
9	7	2.423	2.529	0.351	+4.19

N = 6, Bs = 5, Ar = 10, T = .1, Rt = 1, Is = Id = 0.025.

M	$\frac{(E[Y])^2}{M}$	$\overline{Ra}$	$\overline{Rs}$	Utilization †	% Diff.*
1000	0.03	1.659	1.675	0.180	+0.96
400	0.08	1.734	1.839	0.185	+5.71
300	0.10	1.776	1.898	0.187	+6.43
200	0.15	1.861	2.043	0.193	+8.91

† Average IO utilization at all nodes (from simulation).

* % difference computed as $100*(\overline{Rs} - \overline{Ra})/\overline{Rs}$.

some assumptions (e.g., try a new update arrival distribution).

The Simulation Technique – Disadvantages.

- It is expensive to get results for high IO utilization cases.
- It is hard to find "bugs" in the simulators.

This concludes the comparison of the analysis and the simulation techniques. We have only looked at two algorithms in this thesis, but the analysis for the other algorithms is very similar. (See [GARC78].)

CHAPTER 6

THE PERFORMANCE RESULTS

In this chapter we present some of the performance results for the update algorithms. In section 1 we describe the way we use both the analysis and the simulation results to plot the curves for this chapter. In section 2 we then compare the results of the centralized locking algorithm with hole lists (MCLA) with the results of the distributed voting algorithm (DVA). The performance results for the Ellis type algorithms are given in section 3. The results for the complete centralization algorithm (CCA) and the centralized locking algorithm with wait-for lists (WCLA) are given in section 4. In that section we also compare these algorithms to some of the other algorithms. In section 5 we study the centralized locking algorithm with limited hole list copies (MCLA-h), while in section 6 we compare two of the strategies for handling limited hole list copies. Finally, in section 7, we briefly look at the centralized locking algorithm with total-wait-for lists (TWCLA).

1. HOW THE RESULTS ARE PLOTTED.

When plotting the performance results for the update algorithms we take advantage of the fact that we have two independent techniques for studying the algorithms. We use the simulation results where we know or suspect that the analysis does not provide accurate results. (See chapter 5.) Since the analysis results are inexpensive to obtain (in terms of computing resources), we use them in all other situations. This "hybrid" method is illustrated in figure 6.1 where we show how the graph for figure 6.14 is produced. (We chose to illustrate the hybrid method with figure 6.14 because in this figure the differences between the analysis and the simulation results are some of the largest encountered. In the other figures of this chapter, the differences are usually smaller.)

Figure 6.1

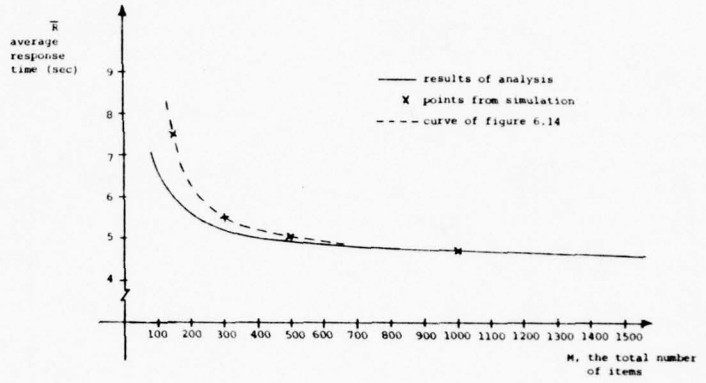


Figure 6.1. The Hybrid Method for Constructing Figures. N=6, Bs=5, Ar=10sec., Is=Id=0.025sec., T=0.01sec., the MEAS algorithm.

2. PERFORMANCE RESULTS FOR THE MCLA AND DVA ALGORITHMS.

In order to study the performance of the algorithms, we selected a set of "typical" parameter values which we consider reasonable for a distributed database implemented with current technology. Each one of these parameter values was then varied in order to discover the effect of the parameter on the system performance. The typical values used were 10 for the interarrival time, A_r; 5 for the base set parameter, B_s; 1000 for the number of items in the database, M; 6 for the number of nodes N; 0.1 seconds for the transmission time, T; 0.01 milli-second for the CPU time slice, C_s; 1 milli-second for the CPU compute time C_u; 0.025 seconds for the IO time slice, I_s; 0.025 seconds for the IO item update time, I_d; and 1 second for the retry time, R_t.

Figures 6.2 through 6.11 present some selected results for the MCLA and the DVA algorithms. The parameter values used to obtain each figure are given in the figures. If the value of a parameter is not given in the figure, then its typical value (defined in the previous paragraph) was used.

Figure 6.2 shows the relationship between the interarrival time A_r, the number of nodes N and the mean response time $\overline{R}$ for the initial set of parameters. In most cases, the MCLA centralized algorithm performs considerably better than the distributed voting algorithm. For small number of nodes, the difference is not as dramatic, but as the number of nodes increases the difference in performance increases. The so called "bottleneck" effect does appear in the centralized algorithm: there is a relatively sharp knee in the curve when the requests for locks swamp the central node (e.g., $N = 6$, $A_r = 5$.) Since there is no bottleneck in the distributed algorithm, one might expect this algorithm to do better under heavy loads. Indeed the distributed algorithm does not have the sharp knee, but it turns out that the distributed algorithm does better only in a limited range and this range covers cases where both algorithms perform poorly. The explanation for this is that in the distributed voting algorithm all nodes get swamped when the update arrival rate increases too much. This can be seen in figure 6.3, which shows the IO utilization as a function of the load (for 6 nodes). The CPU utilization shows similar behavior, but the utilization is much less than the IO one, i.e., the IO server is the critical resource here. Notice that the central node is the one that gets the heavy load in the centralized algorithm, but the total amount of IO time used in the distributed algorithm (sum of all nodes) is considerably greater than the total IO time needed in the MCLA algorithm.

Figure 6.4 shows the number of messages transmitted per update transaction for the case of 6 nodes. In the central algorithm, this number is independent of

Figure 6.2

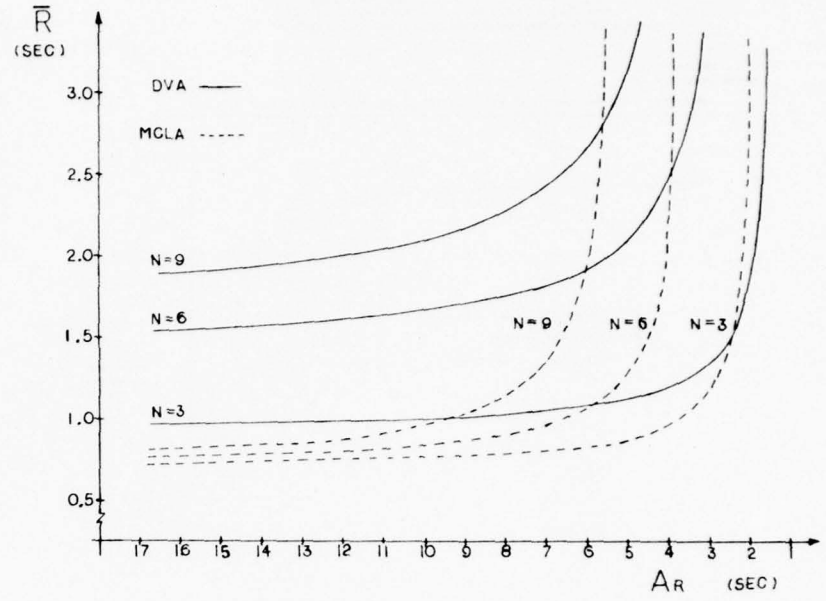


Figure 6.2. The MCLA and DVA algorithms: Effect of Ar and N on the average response time. M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec., Rt=1 sec.

Figures 6.3 and 6.4

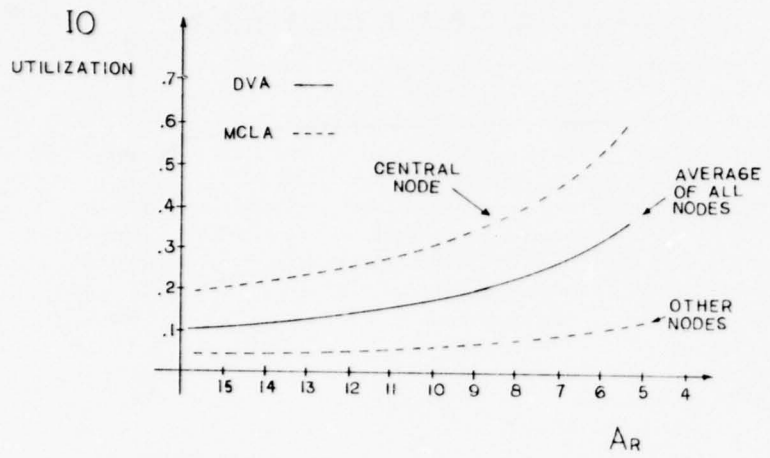


Figure 6.3. The MCLA and DVA algorithms: Effect of Ar on the IO utilization. N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec., Rt=1 sec.

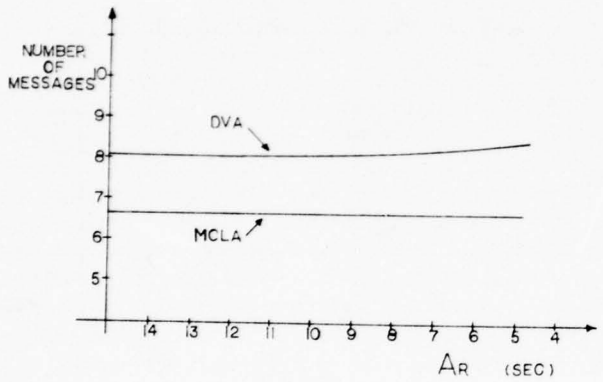


Figure 6.4. The MCLA and DVA algorithms: Effect of Ar on the number of messages transmitted per update. N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec., Rt=1 sec.

the load. Updates that originate at the central node only need 5 messages, while updates originating at other nodes require $5 + 2$ messages. The average over all updates is 6.66 messages per update. In the distributed algorithm, the number of messages increases slightly as some updates are rejected and retried. With no rejections, an update requires 3 messages to obtain a majority of votes, plus 5 messages for performing the update.

For 6 nodes, the difference in the number of messages transmitted is small. The difference accounts for about 0.23 seconds of the difference of response times (since $T = .1$ and since 5 of the messages in the centralized algorithm and 4 in the distributed algorithm do not add to the response time). This means that the big difference in response times comes primarily from the higher total IO utilization of the distributed voting algorithm. For a larger number of nodes, the difference in the number of messages transmitted is substantial and the difference in response times is even greater.

The fact that the MCLA algorithm performs better than the DVA algorithm as the number of nodes increases is illustrated in figure 6.5. The jumps in the curve for the distributed algorithm reflect the number of nodes needed for a majority consensus.

Figure 6.6 shows the effect of M (the number of items) on the system performance. As long as M is large (as compared to B_s), the response time is independent of M. This is convenient since it is possible to extrapolate our results to very large databases. As the size of the database is reduced (with B_s held constant), the number of conflicts among transactions must increase, and therefore the response times grow as M decreases. Notice how the centralized algorithm handles the increased number of conflicts much better than the distributed algorithm. Centralized control is a more efficient way of arbitrating many conflicts.

Figure 6.7 presents the effect of B_s (the mean base set) on the average response time $\bar{R}$ for constant M (number of items). For a small number of conflicts (in this case, $B_s < 5$), the curves are linear since the response time is proportional to the number of items in an update. As the number of conflicts increases ($B_s > 4$), the response time displays a nonlinear component due to the extra delays. In any case, the MCLA algorithm is less sensitive to changes in B_s and even with a mean base set of 1, the central algorithm performs better than the distributed algorithm.

Figure 6.8 shows the effect of varying the IO time slice I_s. An IO time slice of 0 represents a system where all the locks or timestamps are kept in main memory. Once again, the distributed voting algorithm outperforms the MCLA algorithm only in extreme circumstances.

Figures 6.5 and 6.6

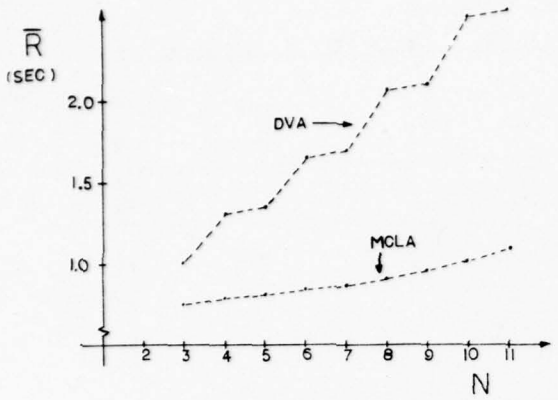


Figure 6.5. The MCLA and DVA algorithms: Effect of N on the average response time. Ar=10 sec., M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec., Rt=1 sec.

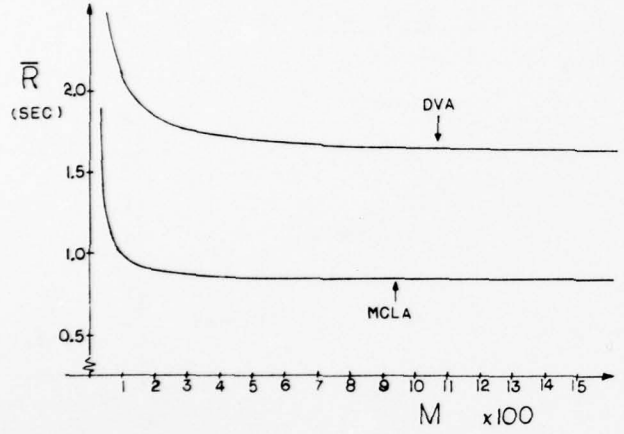


Figure 6.6. The MCLA and DVA algorithms: Effect of M on the average response time. N=6, Ar=10 sec., Bs=5, Is=Id=0.025 sec., T=0.1 sec., Rt=1 sec.

Figures 6.7 and 6.8

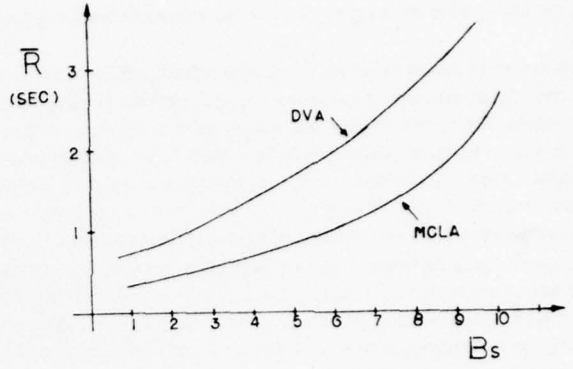


Figure 6.7. The MCLA and DVA algorithms: Effect of Bs on the average response time. N=6, Ar=10 sec., M=1000, Is=Id=0.025 sec., T=0.1 sec., Rt=1 sec.

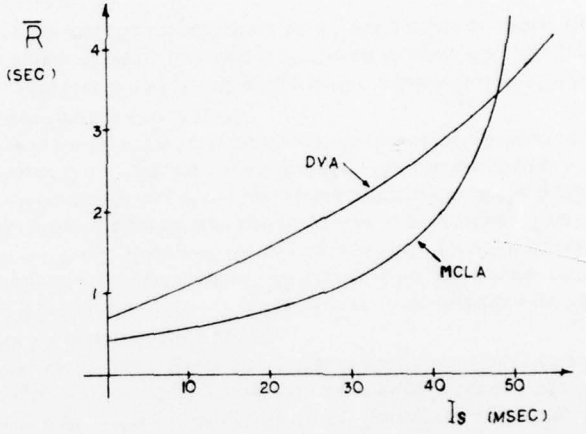


Figure 6.8. The MCLA and DVA algorithms: Effect of Is on the average response time. N=6, Ar=7, M=1000, Bs=5, Id=0.025 sec., T=0.1 sec., Rt=1 sec.

The IO time slice is a very critical parameter because response times grow rapidly as this parameter is increased. Therefore, a good distributed database implementation should try to reduce this value as much as possible. One obvious way to do this is by keeping the most commonly used locks or timestamps in main memory. But it turns out that this is simpler to do for locks than for timestamps. If the number of locked items is usually small, it is easy to keep a complete list of their names in memory. The absence of an item name in the list indicates that the item is free while the appearance of an item's name in the list means that the item is locked. Thus, the IO time slice can be reduced practically to 0 (as long as the number of locked items is small enough so that their names fit in main memory). On the other hand, it is harder to apply the same idea to timestamps. Since it is usually not feasible to keep all timestamps in memory, only the most recently used timestamps are kept. If a timestamp is not found in memory, then it must be paged in from the IO device. If a timestamp is modified, the new value must be written out to the IO device. The savings in IO time will depend on the reference pattern of the updates. The average IO time per timestamp read can be reduced but it will probably be greater than 0. Thus, for a fair comparison of the algorithms, the parameter IO time slice for the distributed voting algorithm should have a greater than 0 value, while the same parameter should be 0 for the MCLA algorithm. It is clear from figure 6.8 that this consideration only makes the distributed algorithm look even worse.

The effect of the network transmission time T on the average response time is shown in figure 6.9. One end of the range represents the slower networks like the ARPANET [KLEI75], while at the other end are the much faster ones like the ETHERNET [METC76]. Here again the MCLA algorithm performs better and is less sensitive to the parameter. For the centralized algorithm, the curve is a straight line with a slope of 1.67 sec/sec. This is the expected value since the average number of messages sent before a node finishes a transaction is 1.67. (For $N = 6$, the central node does not send any, the other nodes send 2 messages.) Since with a larger transmission time, updates take longer to complete and items remain locked a longer time, one would also expect a small nonlinear increase in response time due to the higher probability of conflicts. However, this effect is not noticeable for the centralized algorithm.

With the distributed algorithm, the curve is close to a straight line with a slope slightly greater than 4. (For $N = 6$, the number of messages needed before an update is completed is 3 for voting plus 1 for informing the originating node plus any messages due to rejections.) In this case the secondary effect due to increased probability of conflicts (as T increases) is barely noticeable.

Figures 6.9 and 6.10

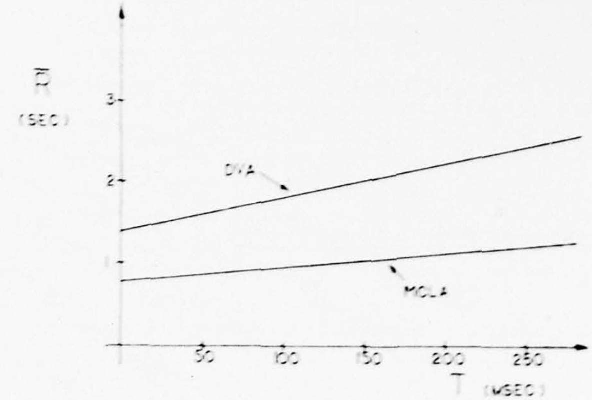


Figure 6.9. The MCLA and DVA algorithms: Effect of T on the average response time. [illegible]

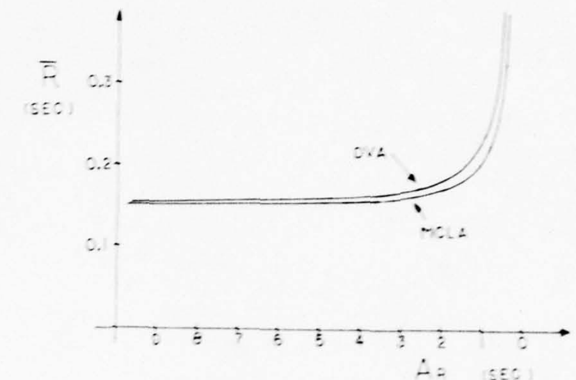


Figure 6.10. A case where the DVA algorithm and the MCLA algorithm have similar performance. [illegible]

Figure 6.10 shows a case where the distributed voting algorithm performs almost as well as the MCLA algorithm. The parameters were chosen so as to favor the distributed algorithm, but as we can see the MCLA algorithm still performs slightly better. The IO utilization is the same for both algorithms (since I_s is 0). The distributed algorithm does slightly better with respect to the number of messages transmitted. This can only happen in the special case of a 3 node network. (The distributed algorithm needs only 1 message to gain a majority of votes plus 2 more to broadcast the update giving 3 messages per update. In the centralized algorithm, the central node only needs 2 messages while the other two nodes need 4 messages, giving an average of 3.33 messages per update.) Therefore we can say that the performance of the distributed algorithm is similar to the performance of the centralized one only under special circumstances.

Figure 6.11 shows the effect of the parameter retry time (R_t) on the mean response time of the distributed voting algorithm. If the retry time is made too small, the rejected request might be retried before the request it conflicted with had a chance to finish. On the other hand, making R_t too large eliminates the possibility of a repeated conflict, but adds unnecessary wait time to the transaction. As is seen in the graph, there is an optimal value of R_t. Unfortunately, the savings obtained by choosing the optimum value do not seem to be very significant. And notice that this graph represents a high rate of conflicts case ($M = 100, B_s = 5$); for a more reasonable case ($M = 1000, B_s = 5$) the savings are hardly noticeable. (This graph not shown.) This is expected since R_t only affects the small number of rejected requests.

2.1 Some Conclusions.

Based on our comparison of the MCLA and DVA algorithms, we can conclude the following:

1) The MCLA algorithm performs considerably better than the distributed voting algorithm except in the cases of extreme IO utilization. In these high load cases, the average response time for updates in the distributed voting algorithm is smaller than the one for the centralized algorithm, although both responses are poor. In most cases, the redundant update problem seems to be solved more efficiently and naturally using centralized control strategies.

Of course, other factors must be considered in choosing an algorithm, but even if the distributed algorithm is chosen for other considerations, it is important to realize that it is going to be a more expensive algorithm, both in terms of

Figure 6.11

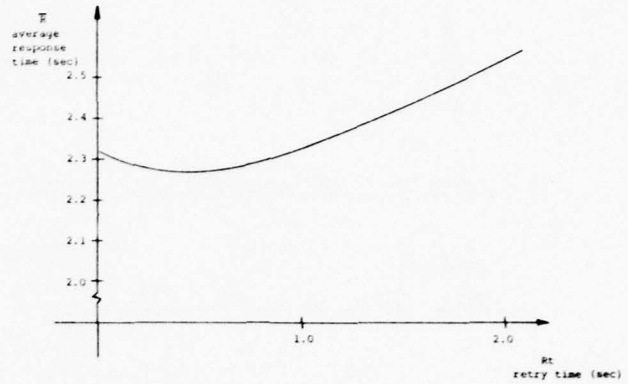


Figure 6.11. The DVA algorithm: Effect of Rt on the average response time. N=6, Bs=5, Ar=10 sec., M=100, Is=Id=0.025 sec., T=0.1 sec.

resources and response time.

2) The distributed voting algorithm is more complex than the MCLA algorithm. The amount of code needed to simulate each algorithm should be a good indication of the complexity of the algorithm. This is true because the length of the simulator code is not proportional to the length of the code in an implementation of the algorithm, but it is rather proportional to the number of states and special cases the implementation will have.

Not considering code common to both simulators (e.g. random number generator, event queue handler, etc.), the MCLA simulator was written in 110 lines of Algol W, while the distributed voting simulator was written in 230 lines, more than twice the number. This is a good indication that the distributed algorithm will be much harder to implement and it will probably be more prone to software bugs.

3) The critical parameters in our system, i.e. those that the system is most sensitive to, are N, the number of nodes, A_r, the interarrival time, B_s, the mean base set size, I_s, the IO time slice and T, the network transmission time. Under normal circumstances, these 5 parameters define the response time of an algorithm. The number of messages is mainly sensitive to the number of nodes N.

4) The simulator can also be useful in tuning an algorithm. The simulator aided us in choosing a good value of R_t. Similarly, it can be used to optimize other aspects of each algorithm: lock granule size, IO or CPU scheduling algorithms, the strategy for eliminating deadlocks, etc.

3. PERFORMANCE RESULTS FOR THE ELLIS TYPE ALGORITHMS.

In this section we present some of the performance results obtained for the three Ellis type algorithms of chapter 3: the original Ellis ring algorithm (OEA), the modified Ellis ring algorithm with sequential updates (MEAS) and the modified Ellis ring algorithm with parallel updates (MEAP). The results were obtained using the analytic techniques of chapter 4, as well as by simulating the algorithms [GARC78].

Figure 6.12 shows the average response time of updates in the MEAS algorithm as a function of the interarrival time A_r for several values of N (the number of nodes) and for the "typical values" of the other parameters. (See section 2.) Recall that small interarrival times imply high arrival rates and high loads. The shape of these curves is very similar to the curves for the distributed voting

Figures 6.12 and 6.13

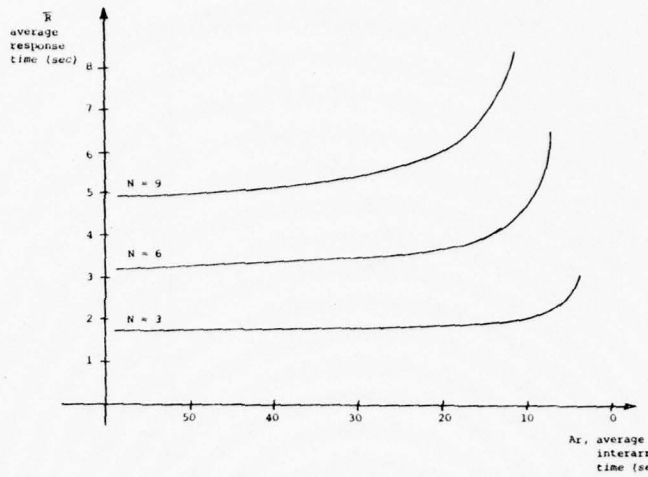


Figure 6.12. The MEAS algorithm: Effect of N and Ar on the average response time. Bs=5, M=1000, Is=Id=0.025 sec., T=0.1 sec.

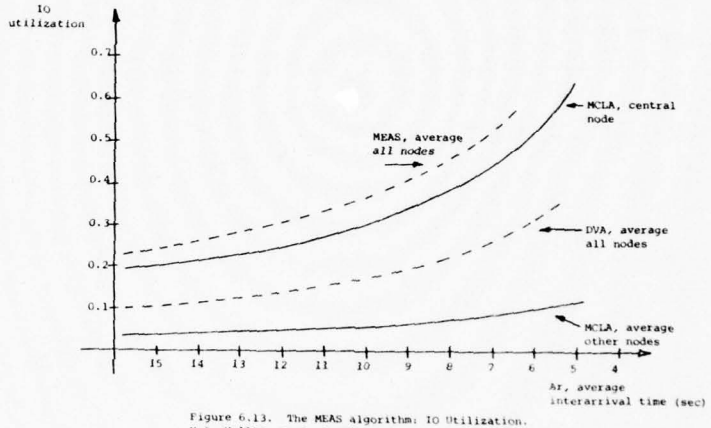


Figure 6.13. The MEAS algorithm: IO Utilization. N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec., Rt=1 sec.

algorithm. (See figure 6.2.) However, the response times are considerably larger than the ones for the distributed voting algorithm under the same circumstances. For example, if the interarrival time A_r is 10 seconds and there are 6 nodes, updates are on the average completed in 1.65 seconds with the distributed voting algorithm, but they take 4.66 seconds on the average with the MEAS algorithm. This is about a 200 percent difference. Furthermore, the MEAS algorithm becomes unstable at lighter loads. For example, the MEAS algorithm is unstable for $A_r = 6$ seconds while the distributed voting algorithm can still operate at that load with an average response time of 1.9 seconds. The difference in performance with the MCLA algorithm is also very dramatic. (See figure 6.2.)

One of the reasons for the poor performance of the MEAS algorithm is the extra transmissions required. However, this only accounts for a part of the difference. For the case of 6 nodes, an update in the MEAS algorithm is delayed by 12 transmission times T before finishing (i.e., two trips around the ring). For the same number of nodes, updates are delayed by 2 and by 4 transmission times in the MCLA and the DVA algorithms respectively. Thus the increase in response time due to transmission delays is only 1.0 and 0.8 seconds over the previous algorithms (because T is 0.1 seconds).

The rest of the difference in response times between the MEAS algorithm and the previous algorithms is due to the increased IO utilization at all nodes. Figure 6.13 shows the IO utilization as a function of the interarrival time A_r for the case of a 6 node network. The IO utilization is greater at all nodes in the MEAS algorithm than it is even in the central node in the MCLA algorithm. This is because every node in the MEAS algorithm must perform locking for all updates. Furthermore, when locks are released with an update, they must be read before they are written. (See procedure Perform-update in Appendix 2.) This is not the case in the MCLA algorithm and this is why the central node has less IO utilization.

For light loads, the number of messages transmitted per update is two times the number of nodes (i.e., two trips around the ring). As the load increases, the average number of messages transmitted increases slightly because the number of conflicts increases and some updates must release their forward locks. (Releasing one forward lock involves, on the average, a message to a fourth of the nodes.) For example, for a six node network and an interarrival time of 7 seconds, the number of messages transmitted per update is 12.018. (The rest of the parameters are the same as in figure 6.12.) This is only slightly above the 12 messages required when no conflicts occur.

Figure 6.14 presents the effect of M, the number of items, on the average

Figures 6.14 and 6.15

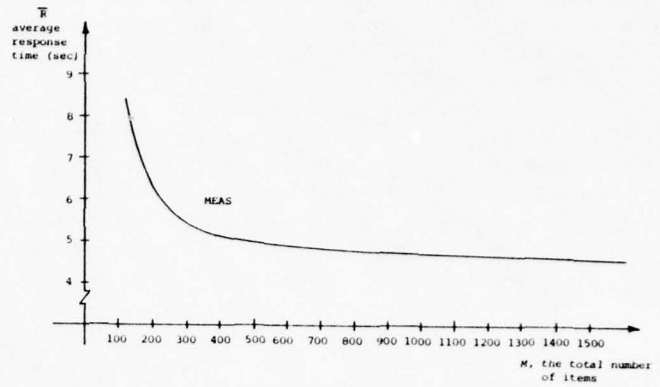


Figure 6.14. The MEAS algorithm: Effect of M on the average response time. N=6, Bs=5, Ar=10 sec., Is=Id=0.025 sec., T=0.1 sec.

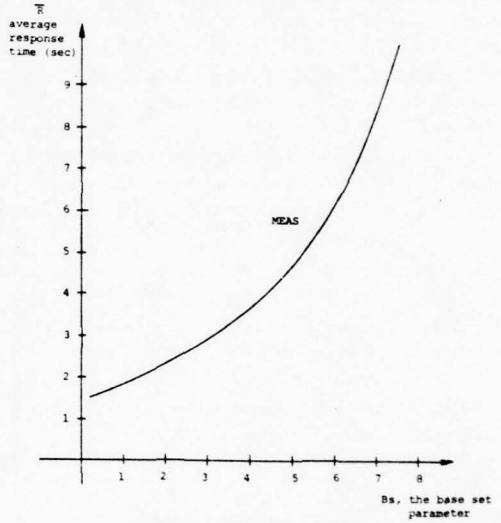


Figure 6.15. The MEAS algorithm: Effect of Bs on the average response time. N=6, M=1000, Ar=10 sec., Is=Id=0.025 sec., T=0.1 sec.

response time of updates in the MEAS algorithm. This curve is also very similar to the one for the distributed voting algorithm. (As we will see, all curves for the MEAS algorithm are similar in shape to the curves for the distributed voting algorithm. Unfortunately, the different scales makes it hard to plot the curves for both algorithms together.) Again, the response times are larger and as M decreases, their value increases more rapidly than in the distributed voting algorithm. As is to be expected, for large M, the average response time is almost independent of M.

Figures 6.15 through 6.18 show the effect of other parameters on the average response time of updates in the MEAS algorithm. (Figures 6.16, 6.17 and 6.18 also contain results for the MEAP algorithm which will be discussed later.) Notice how the MEAS is especially sensitive to the base set parameter, B_s, to the IO time slice, I_s, to the transmission time, T, and to the number of nodes, N. The curve for the average response time versus the transmission time, T, has an interesting but slight non-linearity as T approaches 0. This is due to the fact that, as T approaches 0, the processing of an update at its originating node after the first loop around the ring delays the processing of the same update after its second loop. As T increases beyond a certain threshold, the processing after the first loop can be completed before the update arrives after its second loop.

The performance of the modified Ellis ring algorithm with parallel updates (MEAP) is similar to the performance of the MEAS algorithm except for a reduction in response time proportional to T (the transmission time) and to N (the number of nodes). This is because under normal circumstances no messages have to be delayed before processing and because we are not taking into account the increased overhead due to the longer messages and the handling of the "status" information. (See chapter 3.) If the last two sources of overhead were taken into account, the savings in response time might be less.

When few conflicts occur, the savings in time are roughly NT seconds because updates do not wait for the second trip around the ring before completing. As the load increases, the savings should increase because locks are held for shorter periods of time and thus less conflicts occur. However, this effect is very small and is not observed in the results. For example, figure 6.19 shows the average response time of the MEAP and the MEAS algorithms versus the interarrival time A_r for $N = 6$. In this figure, the difference between the algorithms is roughly the same at all points and equal to 0.6 seconds (since $T = 0.1$ seconds and $NT = 0.6$ seconds).

Figures 6.17 and 6.18 show the effect of T and N on the average response time for the MEAP and the MEAS algorithms. These figures confirm our hypothesis

Figures 6.16 and 6.17

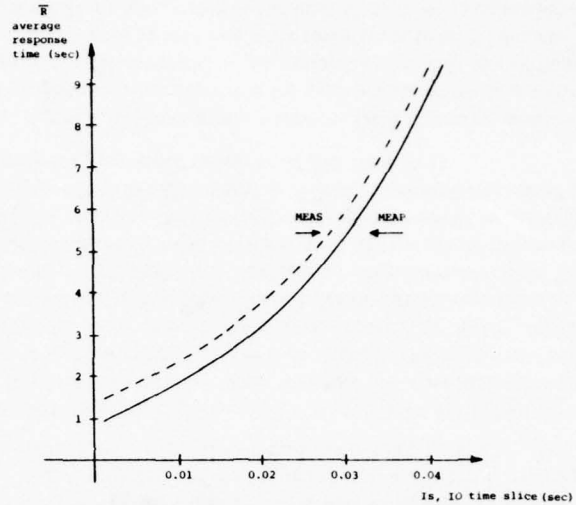


Figure 6.16. The MEAS and MEAP algorithms: Effect of Is on the average response time. N=6, Ar=10 sec., M=1000, Bs=5, Id=0.025 sec., T=0.1 sec.

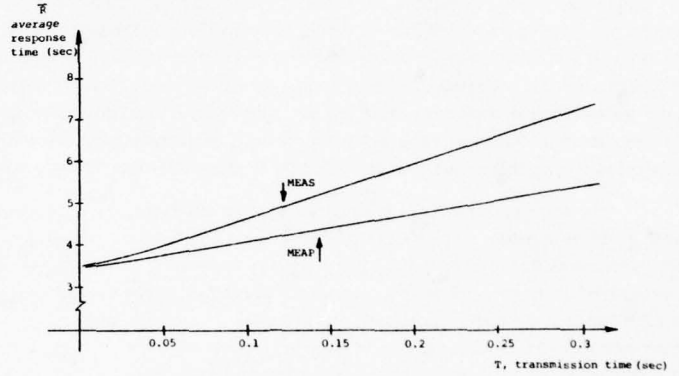


Figure 6.17. The MEAS and MEAP algorithms: Effect of T on the average response time. N=6, Ar=10 sec., M=1000, Bs=5, Is=Id=0.025 sec.

Figures 6.18 and 6.19

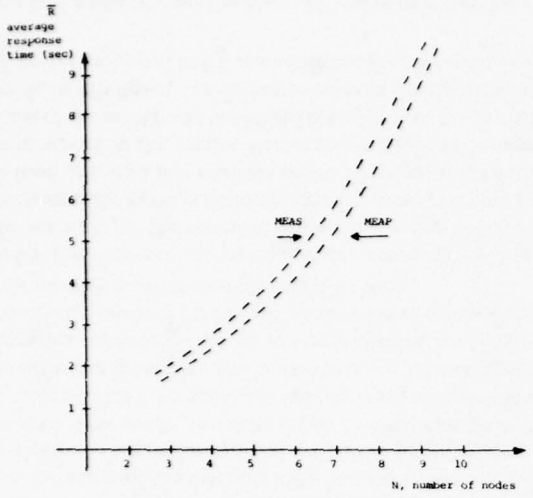


Figure 6.18. The MEAS and MEAP algorithms: Effect of N on the average response time. M=1000, Bs=5, Ar=10 sec., Is=Id=0.025 sec., T=0.1 sec.

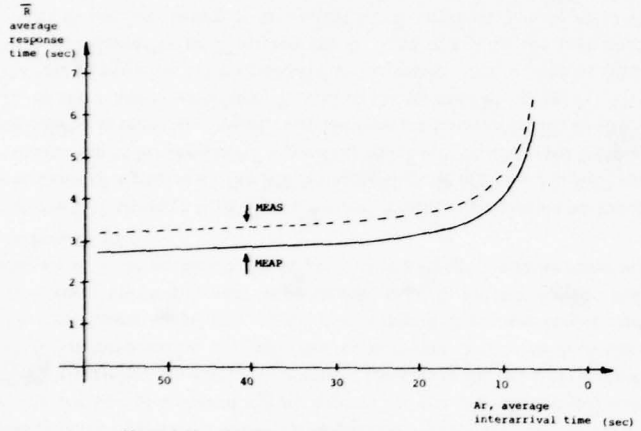


Figure 6.19. The MEAS and MEAP algorithms: Effect of Ar on the average response time. N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

that the savings are NT seconds. Figure 6.16 graphs the average response time versus the IO time slice I_s. As can be seen in this figure, the savings are independent of I_s. The effect of B_s (the base set parameter) and M (the number of items) is not shown because the savings are also independent of these parameters. The IO utilization at each node for the MEAP algorithm is the same as in the MEAS algorithm because the IO requests of each update are the same. Figure 6.13 shows this IO utilization as a function of the interarrival time A_r.

Figure 6.20 shows the average response time of the original Ellis algorithm (OEA) versus the interarrival time A_r for a six node network. Notice that for the OEA algorithm we assume that the IO time slice, I_s, is 0 because all state information can be kept in each node's main memory. When we compare the OEA algorithm with the MEAS algorithm where locks are kept in an IO device, we find that the OEA algorithm performs better as long as the load is not too heavy. But when a large enough number of updates per second has to be processed, the IO time invested by the MEAS algorithm to lock individual items pays off and it performs better than the OEA algorithm. As can be seen in figure 6.21, for the case where $I_s = 0.025$ seconds, the MEAS algorithm performs better in a very small interval. However, as I_s is reduced for the MEAS algorithm (I_s is always 0 for the OEA algorithm), the interval becomes larger and when $I_s = 0$, the MEAS algorithm performs better for all values of A_r. (See figure 6.20.)

Notice that response times in the OEA algorithm are independent of the number of items in each database (M). The performance of the OEA algorithm can be viewed as the limiting performance of the MEAS algorithm with $I_s = 0$ as the number of items in each database approaches 1. As M decreases, the curve of the average response time versus A_r for the MEAS algorithm approaches the curve for the OEA algorithm. For example, the curve for $M = 300$ is also shown in figure 6.20. The response times for that case are in between the ones for the the OEA algorithm and the MEAS algorithm with $M = 1000$. However, the performance of the MEAS algorithm is close to the performance of the OEA algorithm only for very small values of M (i.e. close to 1).

Finally, figure 6.22 shows a case where the OEA algorithm performs better than the distributed voting algorithm for some values of the interarrival time, A_r. Here again, the OEA algorithm can perform better than the DVA algorithm because it does not need to read and write timestamps to an IO device. The case shown in figure 6.22 is a special case and in most other cases, the distributed voting algorithm performs better. (Notice that in figure 6.22, the base set parameter B_s is 7 and not 5 as usual.)

Figures 6.20 and 6.21

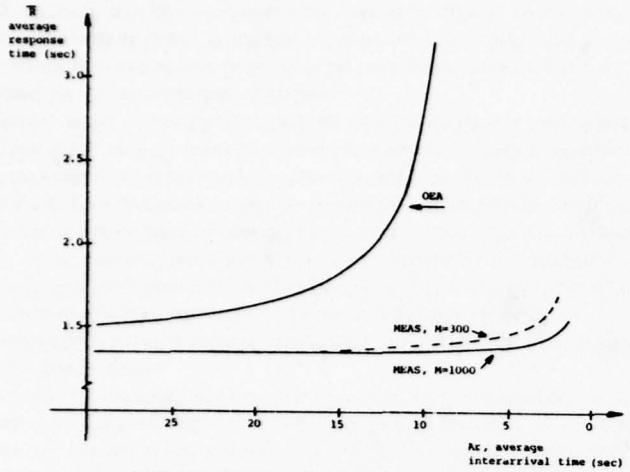


Figure 6.20. The OEA and MEAS algorithms: Effect of Ar on the average response time. N=6, Bs=5, Id=0.025 sec., T=0.1 sec., Is = 0 (both algorithms).

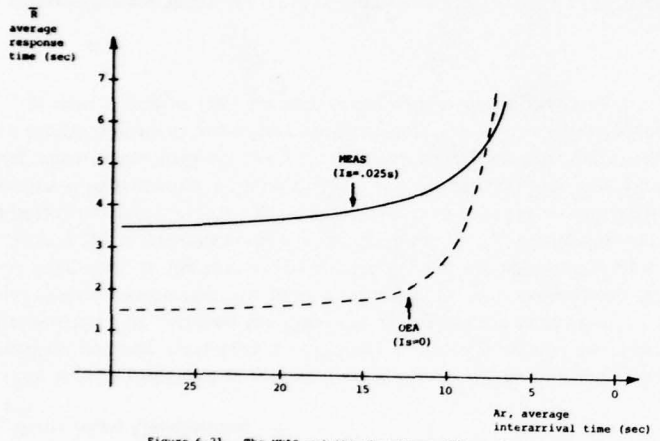


Figure 6.21. The MEAS and OEA algorithms: Effect of Ar on the average response time, for different values of Is. N=6, Bs=5, M=1000, Id=0.025 sec., T=0.1 sec., Is in OEA = 0, Is in MEAS = 0.025 sec.

Figure 6.22

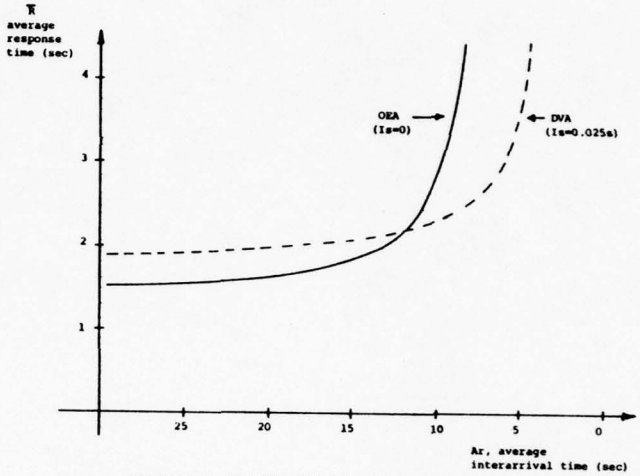


Figure 6.22. The OEA and DVA algorithms. N=6, M=1000, Bs=7, Id=0.025 sec., T=0.1 sec., Is in OEA = 0, Is in DVA = 0.025 sec., Rt=1 sec.

3.1 Some More Conclusions.

The results presented in section 3 show that the Ellis ring algorithm and its variants perform worse than the DVA and the MCLA algorithms under most circumstances. Furthermore, the Ellis type algorithms are more sensitive to the critical system parameters: the number of nodes, N, the transmission time, T, the IO time slice, I_s, the base set parameter, B_s, and the interarrival time A_r.

Among the algorithms studied in section 3, the modified Ellis ring algorithm with parallel updates (MEAP) performed better than both the modified Ellis ring algorithm with sequential updates (MEAS) and the original Ellis ring algorithm (OEA) when tested with the same set of parameters. However, the complexity of the MEAP algorithm is greater than the complexity of the MEAS algorithm, which in turn is greater than the complexity of the OEA algorithm.

4. PERFORMANCE RESULTS FOR THE CCA AND WCLA ALGORITHMS.

Curve "CCA" of figure 6.23 shows the average response time of update transactions with the CCA algorithm, as a function of the transaction interarrival time A_r, for the set of representative parameter values. Notice that as A_r decreases, the arrival rate of transactions and the load increases. In this curve we observe a sharp knee which occurs when the central node is swamped by requests to process transactions.

In order to provide a point of comparison, in figure 6.23 we also show the performance of the DVA algorithm. The average response time of update transactions with this algorithm is given by curve "DVA" in figure 6.23. This algorithm does not have a central node which acts as a bottleneck, but surprisingly, its performance is not as good as that of the CCA algorithm. The main reasons for this relatively poor performance of the distributed voting algorithm are that (a) transactions must visit a majority of nodes (instead of one) before being executed, and (b) the CPU and IO loads produced by a voting operation at a node are considerable, while in the CCA algorithm there is no IO and very little CPU load caused by the serialization of updates.

Although it is not shown in figure 6.23, both algorithms saturate at about the same interarrival time. When the loads become very high, the analysis is not very accurate and the simulations are very expensive to run. Fortunately, we are not very interested in this region because both algorithms perform so poorly there. For all cases which are not close to the saturation point, the CCA

Figure 6.23

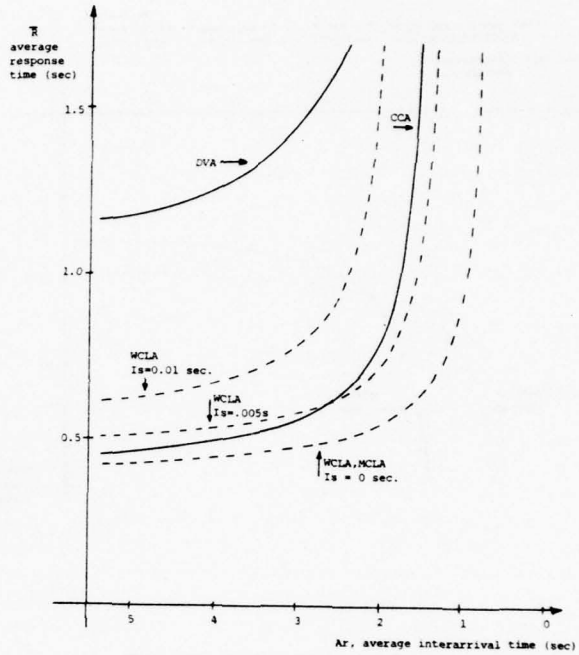


Figure 6.23. The WCLA and MCLA algorithms.
N=6, M=1000, Bs=5, Id=0.025 sec., T=0.1 sec.,
Is in DVA = 0.01 sec., Rt=1 sec.

algorithm performs better than the distributed voting algorithm.

The results of figure 6.23 are for the particular set of parameter values shown in the figure. Extensive tests have been run to study the effect of the parameters on the average response time. We have found that the CCA algorithm, just like the MCLA algorithm, performs better than the DVA algorithm in most cases of interest. The actual difference in average response time between the two algorithms can be reduced or increased by varying some parameters, but the basic relationship remains unchanged. For a two or three node system and for a small value of the I_s parameter (i.e., the IO time to read or write a timestamp), the performance of the CCA and DVA algorithms is very similar. As the number of nodes N, the transmission time T, or I_s increases, the difference in average response time increases and the CCA algorithm becomes more attractive. Notice that the results of figure 6.23 are for an IO bound situation. However, the results are similar for a CPU bound case.

Under our constant transmission time assumption, the performance of the WCLA algorithm is the same as the performance of the MCLA algorithm because both algorithms eliminate all unnecessary delays. In figure 6.23 we have also shown the average response time of these two centralized locking algorithms, for three values of the I_s parameter. Recall that the I_s parameter is the IO time needed to set or check a lock. In the WCLA algorithm the value of this parameter should include the IO time needed to maintain the table of last transactions that modified the items (called LAST(i) in chapter 3). Since this table will usually be in an IO device, the value of I_s will probably be greater than zero for the WCLA algorithm. On the other hand, the I_s parameter will usually be very close to zero in the MCLA algorithm.

Hence, the lower curve ($I_s = 0$) should only be considered as a lower bound for the WCLA algorithm, while this same curve is the most likely average response time for updates in the MCLA algorithm. As can be seen in figure 6.23, it is possible for the WCLA algorithm to perform worse than the simple CCA algorithm. This occurs when the locking overhead becomes larger than the data reading load which was moved out of the central node by the WCLA algorithm. By using caches, the value of I_s for the WCLA algorithm may be reduced, thus making this algorithm more attractive.

(In a system where communication delays have a large variability, the performance of the WCLA algorithm might be better than the performance of the MCLA algorithm. (See chapter 3.) However, in such cases, the response time of transactions in all algorithms will be affected, and which algorithm performs better will depend on the type of the communication delays.)

5. PERFORMANCE RESULTS FOR THE MCLA-h ALGORITHM.

5.1 Size of the Hole List.

Before we study the performance of the MCLA-h algorithm, it is a good idea to estimate the size of the hole list at the central node. We assume that there are no unnecessary delays for non-conflicting updates. (That is, assume that $h = \infty$.) Let $\overline{S}$ be the average time that a hole remains on the hole list at the central node. Using Little's equation, we can obtain the average size of the hole list, $\overline{H}$, as

$$\overline{H} = \overline{S}(N\lambda)$$

where $N\lambda$ is the rate of update lock grants at the central node. (N is the number of nodes in the system, λ is the arrival rate of updates at each node, and we assume that the system is stable. $N\lambda$ can also be interpreted as the arrival rate of holes to the hole list.) Since we are only interested in an estimate, we use $\overline{R}$, the average response time of updates, instead of $\overline{S}$. ($\overline{R}$ is computed assuming that there are no unnecessary delays.) Since $\overline{R}$ will always be larger than $\overline{S}$,

$$\overline{H} \leq \overline{R}(N\lambda) \qquad (1)$$

Using the values for $\overline{R}$ obtained in chapter 5, we can compute estimates for $\overline{H}$ for some typical cases. These cases are shown in table 6.1. As can be seen in table 6.1, the average size of the hole list for the typical cases is quite small. This suggests that a MCLA-h algorithm with a small value of h might perform just as well as the MCLA-infinity algorithm which has no unnecessary delays. For example, in a 6 node network with an update arrival rate of 0.2 updates per second per node ($\overline{H} = 1.69$), an h of 2 or less would not give us good results, but an h of 5 should be enough to allow most updates to proceed without unnecessary delays.

As the number of nodes, N, increases, we expect the average hole list size to increase. (See equation (1).) However, as N increases, the maximum possible value of λ decreases because the system can handle less updates per node. This is why the increase in $\overline{H}$ in table 6.1 as N goes from 6 to 9 is small. We therefore expect that a relatively small value of h will still be adequate in larger networks.

If the total number of update arrivals $N\lambda$ is held constant, then the value of $\overline{H}$ does not increase as N is increased. This is true because in the MCLA algo-

Table 6.1

TABLE 6.1

ESTIMATES FOR THE AVERAGE HOLE LIST SIZE ($\overline{H}$).

For Bs = 5, M = 1000, Is = Id = 0.025 sec., T = 0.1 sec.
(Assuming that no unnecessary delays occur.)

N	λ	$\overline{R}$	$\overline{H}$
6	1/15	0.800	0.32
6	1/7	1.010	0.87
6	1/5	1.415	1.69
9	1/10	1.005	0.90
9	1/7	1.553	2.00

N is the number of nodes.

λ is the arrival rate of updates at each node (= 1/Ar).

$\overline{R}$ is the average response time of updates (from chapter 5).

$\overline{H}$ is the estimated upper bound for the size of the hole list.

rithm, the average response time of updates does not increase if $N\lambda$ is constant. (See equation (1).)

5.2 The Simulator for the MCLA-h Algorithm.

The MCLA-h algorithm was not analyzed with the technique of chapter 4 because the analysis seemed to be too complex. Instead, a simulator, similar to the ones used for the other algorithms, was written. As in the previous simulators, we assume that the database is completely duplicated at every node, we assume that there are no read-only transactions and we assume that there are no failures. We also assume that the message transmission delays are constant and equal to T seconds. This automatically implies that there are no excessive transmission delays in the system. Therefore, when a "grant" message arrives at the update's originating node, we are guaranteed that all necessary previous updates have been seen at that node. Thus, with the MCLA-infinity algorithm there are no delays caused by sequence numbers. In a MCLA-h algorithm with finite h, there might be delays in the central node when a hole list size exceeds the limit h, but once the grant message is sent, the update will not be delayed further due to sequence numbers. These observations simplify the design of the simulator because with constant transmission time, some sections of code in the MCLA-h algorithm are not needed. (In particular, Procedure Perform-update(A,n) in appendix 1 can be simplified to

Procedure Perform-update(update A; node n);
    if n = c then central-update(A,c)
    else update local database as indicated by update-values(A);)

Finally, we assume that there is no IO or CPU overhead due to the handling of the hole lists. Our predictions indicate that the hole list will be small and will therefore fit in main memory. The additional CPU time needed to handle these small lists should be very small. (If this last statement is not true, the value of C_s, the CPU time slice, can be increased to compensate. However, with our parameter values, any small increase in C_s will not alter the results significantly.)

5.3 The Results.

We now present some selected results obtained from the simulator. Figure 6.24 is a graph of the average response time of updates, $\overline{R}$, as a function of the

update interarrival time at each node, A_r ($A_r = 1/\lambda$), for several values of the hole list copy size limit h, in a six node network. The other simulation parameters are set to their "typical" values. (See section 2.) High update loads are to the right of the figure while light loads (low update arrival rates) are to the left.

The curve for $h = \infty$ is the same as the one given in figure 6.2 for the MCLA algorithm. Notice that for an h larger than 3, the curves for the average response time are almost indistinguishable from the curve for $h = \infty$. This means that in this case, a value of h of 4 is sufficient to obtain the best performance possible. A hole list limit of 4 is reasonable and should not produce much overhead when appended to "grant" and "perform update" messages. (The value of 4 for h confirms our guess for a good and small h. See section 5.1.)

Figure 6.25 shows these same results in a different way. In this figure, we plot the average response time as a function of h for different values of A_r, the update interarrival time. Notice that for some high load cases (e.g., small A_r), the value of $\overline{R}$ is not shown when h is small. In these cases the system is saturated and the value of $\overline{R}$ is not defined. Figure 6.25 shows that the value of h becomes a critical system parameter as the system becomes heavily loaded. However, even in these cases, an h of 4 or 5 is sufficient to bring the response time down close to its minimum value.

Figure 6.26 gives the average size of the hole list, $\overline{H}$, (obtained from the simulations) as a function of the hole list size limit h for several values of A_r, the update interarrival time. (Again, in some cases where h and A_r are small, the system is saturated and $\overline{H}$ is undefined.) The values of $\overline{H}$ for the case of no unnecessary delays (e.g., large h) agree well with the bounds predicted above. (See table 6.1.) For example, for $A_r = 5$ seconds, we predicted that $\overline{H}$ would be less than 1.69. From figure 6.26, the true value (for large h) is about 1.25. As the size of h is decreased, the number of unnecessary delays increases. This increases the average response time and thus the average size of the hole list grows proportionately. (See equation (1).) (Notice that $\overline{H}$ can be larger than h because $\overline{H}$ is the average hole list size at the central node. The size of this list is not bounded; only the size of a copy of this list placed in a message is restricted to size h.)

The fraction of the updates that are delayed at the central node due to large hole lists is shown in figure 6.27 as a function of h for different values of A_r. (When the value is not shown, the system is saturated.) Notice that for h larger than 5, the number of delayed updates is negligible for all values of A_r shown.

Figure 6.24

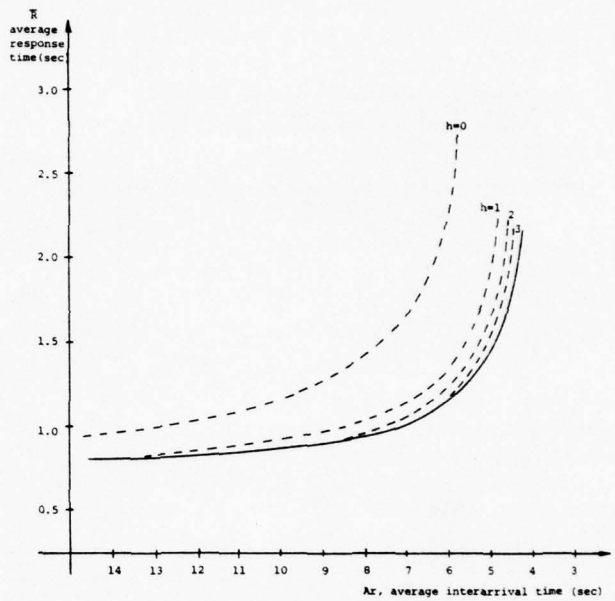


Figure 6.24. The MCLA-h algorithm: Effect of Ar and h on the average response time. The delay at central node strategy is used. These are simulation results. N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

Figures 6.25 and 6.26

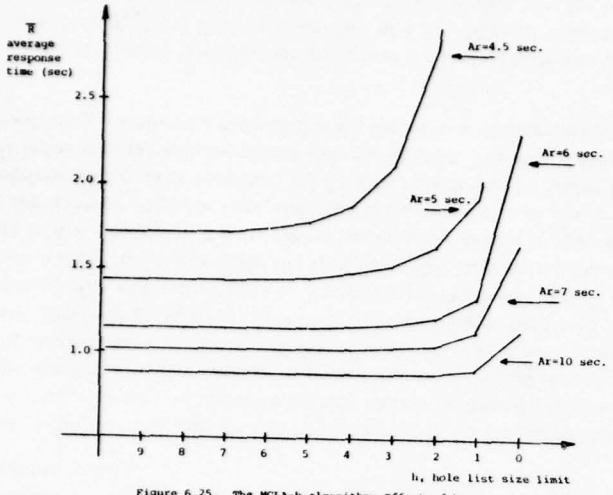


Figure 6.25. The MCLA-h algorithm: Effect of h.
N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

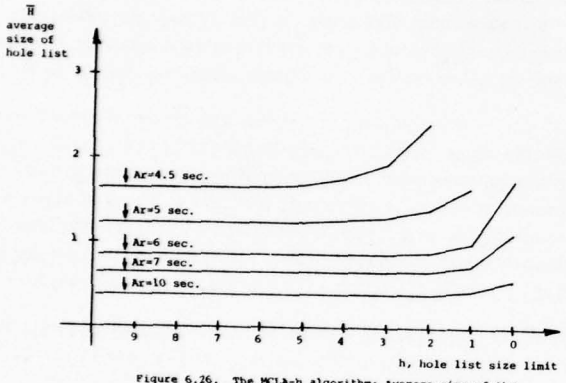


Figure 6.26. The MCLA-h algorithm: Average size of the hole list. N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

Figure 6.27

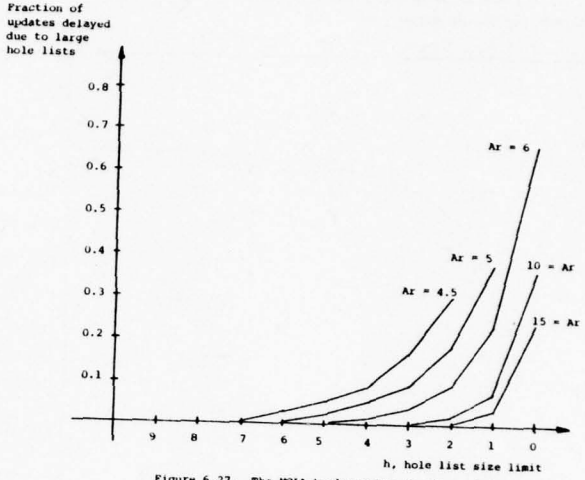


Figure 6.27. The MCLA-h algorithm: Fraction of delayed updates. N=6, M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec. (The values of Ar given in the figure are in seconds.)

5.4 Results for a Hyperexponential Base Set Distribution.

The results of the previous section were obtained with a discrete exponential distribution for the number of items referenced by an update. (See chapter 4). With this distribution, updates with large base sets (e.g., updates that reference many items) do not occur very often.

The MCLA-h algorithm with a small value of h is very sensitive to updates with large base sets because each such update produces a hole that remains on the hole list at the central node for a long time. Thus, a small number of updates with large base sets can easily cause the hole list to exceed its limit h and can cause delays to occur with the rest of the shorter updates. To study the effect of updates with large base sets, we consider a different base set distribution: a discrete hyperexponential distribution.

With a discrete hyperexponential distribution, a certain fraction, p, of the base sets are generated from a discrete exponential distribution with mean X_1 while the rest of the base sets $(1-p)$ are generated from a discrete exponential distribution with mean X_2. ($0 \leq p \leq 1$.) For our study, we make a small fraction of the updates (p) have a large base set (with mean X_1), while the majority of the updates have smaller base sets (with mean X_2). In order to present the same average load to the system, we will require that the average size of the base sets of all updates (i.e., $pX_1 + (1-p)X_2$) be the same as the average size of the base sets with the original discrete exponential distribution. As was shown in chapter 4, this mean was

$$\text{original mean} = [1 - \exp(-1/B_s)]^{-1} \tag{2}$$

(where B_s is the base set parameter in the performance model), so we need

$$pX_1 + (1-p)X_2 = [1 - \exp(-1/B_s)]^{-1} \tag{3}$$

To find the required value of X_1 we need a second equation. We therefore define the parameter q to be the factor by which X_1 is larger than X_2. That is,

$$X_1 = qX_2. \tag{4}$$

This gives the value of X_1:

$$X_1 = \frac{[1 - \exp(-1/B_s)]^{-1}}{p + (1-p)/q} \tag{5}$$

Having computed X_1, we can find X_2 from equation (4).

The values of X_1 and X_2 we have found are the means of the two discrete exponential distributions. The mean values for the continuos exponential distributions that are used for generating the discrete distributions, B_{s1} and B_{s2}, are different and can be computed by inverting equation (2):

$$B_{s1} = -\left[\ln\left(1 - X_1^{-1}\right)\right]^{-1},$$
$$B_{s2} = -\left[\ln\left(1 - X_1^{-1}\right)\right]^{-1}.$$

The MCLA-h algorithms were simulated using the new hyperexponential distribution for the base sets. There were four cases studied:

1) $p = 0.1$, $q = 10$. That is, 10 percent of the updates have larger base sets. The mean size of the base sets of these updates was 10 times larger than the mean size of the base sets of the rest of the updates. For the case of $B_s = 5$, $p = 0.1$ and $q = 10$ implies that $X_1 = 29.03$ and $X_2 = 2.903$.

2) $p = 0.05$, $q = 10$. In this case, only 5 percent of the updates have the larger base set. The average size of the large base sets is $X_1 = 38.04$ while the average size of the smaller base sets is $X_2 = 3.804$.

3) $p = 0.1$, $q = 5$. Ten percent of the updates have larger base sets, but the mean size of these larger base sets is only 5 times larger than the mean size of the smaller base sets. In this case, $X_1 = 19.70$ and $X_2 = 3.940$.

4) $p = 0.0$. In this last case, there are no updates with larger base sets. In other words, all base sets are generated from a single discrete exponential distribution with mean $X_1 = 5.516$. This is the original distribution that was used to obtain all previous results.

Notice that in all cases above, $pX_1 + (1-p)X_2 = 5.516$, the mean of the original discrete exponential distribution.

The four cases were simulated for an interarrival time A_r of 7 seconds. The rest of the simulation parameters were not changed. Figures 6.28 through 6.30 show the results.

In figure 6.28, the average response time of updates, $\overline{R}$, is shown as a function of h, the hole list size limit for the four cases. (This figure should be compared with figure 6.25.) Even though in all four cases the mean size of the base sets is constant, the response times are considerably different. This difference is partially due to the fact that the variances of the different base set distributions are not the same. As was expected, the MCLA-h algorithm is more sensitive

Figures 6.28 and 6.29

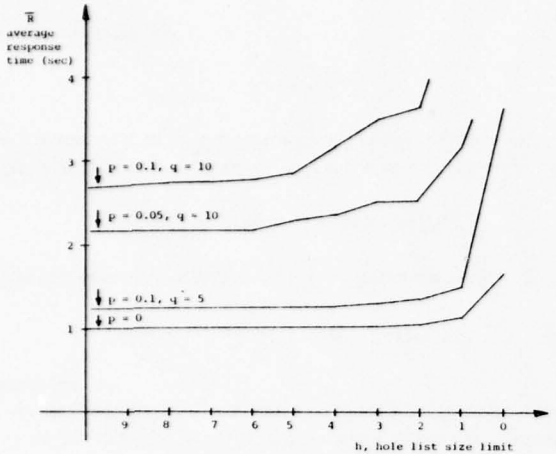


Figure 6.28. The MCLA-h algorithm with hyperexponential base sets: Effect of h. N=6, Ar=7 sec., M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

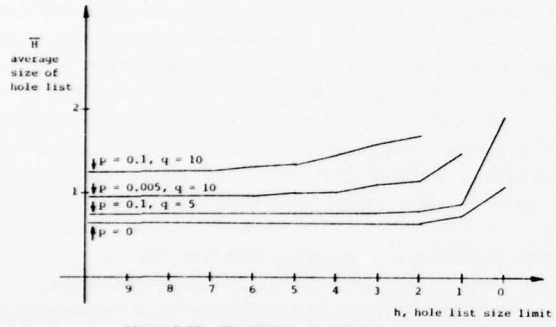


Figure 6.29. The MCLA-h algorithm with hyperexponential base sets: Average size of the hole list. N=6, Ar=7 sec., M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

to h when we use hyperexponential distributions. However, even in these cases, a relatively small value of h (e.g., 6) produces average response times close to the ones obtained with $h = \infty$. As was also expected, as p (the fraction of the updates with large base sets) or q (the factor by which the larger base sets are bigger) are decreased, the average response times and the sensitivity to h both decrease.

The average size of the hole list, $\overline{H}$, is shown in figure 6.29 as a function of the hole list size limit h. As was observed in figure 6.26, as the average response times grow, the value of $\overline{H}$ grows proportionately. Figure 6.30 shows the fraction of the updates that are delayed at the central node because their hole lists exceeded the limit h. As was expected, as p or q grow, the number of delayed updates increases.

5.5 Some More Conclusions.

If the distribution of the number of items in the base set of updates is discrete hyperexponential, the average response times of updates increase. This increase is mostly due to the increased variance of the hyperexponential distribution.

For a given value of the hole list size limit, h, as the fraction of updates with larger base sets (p) increases or as the mean base set size of these updates (q) increases, the response time increases. This increase is due to the fact that the updates with larger base sets produce holes that remain on the hole list for longer periods of time. This causes the hole list to overflow (i.e., its size exceeds h) and causes updates to be delayed. However, a relatively small value of h (e.g., 6) makes this increase in response time negligible.

6. COMPARISON OF STRATEGIES FOR LIMITED HOLE LIST COPIES.

In section 1.9 of chapter 3 we described several strategies for handling limited hole list copies. One of these strategies was to delay sending the "grant" message for a transaction until the hole list copy of the transaction had shrunk to a size smaller than the limit h. We called this the "delay at central node" strategy. Another solution was to simply truncate the hole list copy to the right size. This was the "truncating" strategy.

In appendix 7 we analyze the delays involved in the "delay at central node"

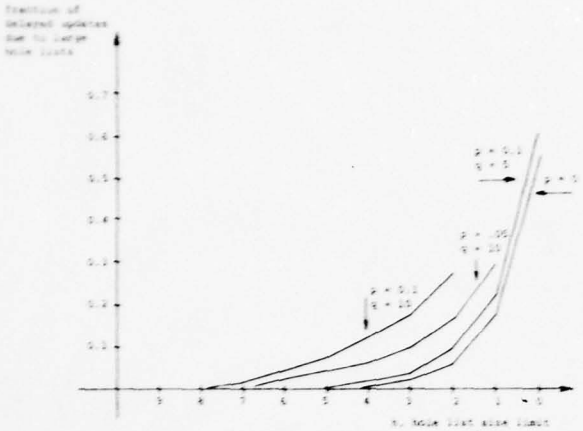


Figure 6.3b. The MCLA algorithm with hyperexponential hole sets. Fraction of delayed updates. $\bar{R}=0.1$ sec., $R^2/\bar{R}^2=5$, $\bar{R}=0.05$ sec., $R^2/\bar{R}^2=10.25$ sec., $\bar{R}=0.1$ sec.

and the "truncating" strategies and we compare these delays. In the appendix we show that if the central node does know what holes will disappear first from the hole list, then the "truncating" alternative is superior. We also show that if the central node cannot predict what holes will disappear first, then the "delay at central node" alternative is superior in most cases.

7. THE SIZE OF THE "TOTAL-WAIT-FOR" LIST.

In section 1.10 of chapter 3, we described a centralized locking algorithm (TWCLA) which used total-wait-for lists. The TWCLA algorithm is a centralized locking algorithm like the MCLA algorithm. Both of these algorithms eliminate all unnecessary delays (at least with our performance model), so we expect the performance of both algorithms to be similar. Hence, we will not study the performance of the TWCLA algorithm in this thesis.

In this section we will only estimate the average size of the total-wait-for list at the central node in the TWCLA algorithm. It turns out that the average size of the total-wait-for list is roughly the same as the average size of the hole list. This means that the performance of the TWCLA algorithm with limited total-wait-for lists should also be similar to the performance of the MCLA algorithm with limited hole lists.

The average size of the total-wait-for list, $\bar{F}$, can be estimated as follows: Assume that the system is stable and let $\bar{S}$ be the average time that an element remains on the total-wait-for list. The sequence number of update A is added to the total-wait-for list when A releases its locks. Assuming that the process of granting all locks to updates at the central node is a Poisson process with arrival rate $N\lambda$ (see chapter 4), the time to the next grant is $1/(N\lambda)$ (because of the memoryless property). Say this grant is for update B. Then A's sequence number will remain on the total-wait-for list at most until B releases its locks. (Notice that a third update C could obtain its locks after B did but could finish before B did.) The average time for B to release its locks will be less than $\bar{R}$, the total average response time of updates. Thus,

$$\bar{S} \leq \frac{1}{N\lambda} + \bar{R}$$

and by Little's formula,

$$\bar{F} \leq (N\lambda)\bar{S} = 1 + \bar{R}(N\lambda).$$

Therefore, we expect the average size of the total-wait-for list to be about the same size as the average size of the hole list. (See equation (1).) However, it seems that the size of the total-wait-for list will not be as sensitive to updates with large base sets as the hole list was. As long as there are updates with small base sets finishing in "short" times, the updates with larger base sets cannot cause the total-wait-for list to grow. Some simulations are needed to confirm this hypothesis.

This concludes the presentation of the performance results for the update algorithms.

CHAPTER 7

CRASH RECOVERY

This chapter starts the second part of the thesis. In this second part we will eliminate some of the restrictions that were made for the performance analysis of the first half of the thesis.

We start by considering failures and their effect on the system performance. We will show that it is possible to design a resilient centralized locking algorithm which does not use backup central nodes and which involves little additional overhead during normal operation as compared to the original algorithm. When a failure occurs, the resilient centralized locking algorithm will temporarily operate inefficiently. However, after the system adapts to the new state, the algorithm will operate as efficiently as before. If we assume that failures are rare, the average performance of the resilient algorithm will be similar to the performance of the original centralized locking algorithm. Thus, the centralized locking strategy continues to be an attractive alternative even in the presence of failures.

In section 1 we comment on the state of the art in crash recovery for distributed databases. Next, in section 2, we describe the types of failures that can occur and we state which types we will not consider in this thesis. Before describing the resilient centralized locking algorithm, in section 3 we discuss some basic concepts that apply to any resilient update algorithm. In section 4 we outline the resilient centralized locking algorithm, and in section 5 we briefly compare its performance to the other algorithms we have studied, assuming that these algorithms are also modified to make them resilient.

1. STATE OF THE ART IN CRASH RECOVERY.

Many crash recovery techniques and algorithms for distributed databases have recently appeared in the literature [e.g., GRAY77, STON78, ROSE78, MENA78, BADA78, LAMP, THOM76, ALSB76, BERN78, ROTH77]. Before

describing any new algorithms, it is important to understand what has been achieved up to now in this area, what is still missing and what this chapter attempts to accomplish.

By examining the current literature, we can reach several general conclusions:

(1) There are many types of failures that can occur in a distributed database system. Any resilient algorithm should be able to deal in a proper way with at least a good number of these types of failures. We will discuss some of the types of failures in section 2.

(2) There are many available strategies for dealing with the different types of failures. For example, there is a "two phase commit protocol", there are "time-outs", there are "transaction logs", and there are "message pipes". (These techniques do not solve the same problem.) We refer the reader to the literature for descriptions of these strategies.

(3) If one considers one type of failure and one recovery strategy in a simple algorithm, then the incorporation of resiliency to the algorithm is not hard, it simply has to be done carefully.

(4) However, making a complex algorithm resilient to different types of failures which can occur simultaneously is much much harder. The problem is not conceptually hard; it is simply that close attention has to be paid to innumerable details. Some of the published papers are clear testimony to this fact.

(5) But the worst part of the problem is that none of the complex resilient algorithms has been formally proven correct. After reading pages and pages of detailed code, the reader is always left with the feeling that there might be a case that has not been considered. Several such cases or "bugs" have already been found in published algorithms. Once found, a bug is relatively easy to fix by including additional tests in the algorithm. But until the algorithms are formally proven correct (which might take some time considering the complexity of the algorithms), there will always be the possibility of discovering more bugs.

Considering all this, in this chapter we will not attempt to present still another detailed resilient distributed algorithm. Unless we could prove the correctness of such an algorithm, the algorithm itself would not be a significant contribution. The main thrust of this thesis is the performance of update algorithms for distributed databases. Thus our main interest is not how to design resilient algorithms, but how these resilient algorithms will perform. Fortunately for us, it is not necessary to give all the gory details of an algorithm to understand how it will perform. Therefore we will simply give a "top level" description of the resilient centralized locking algorithm we are interested in. We will also give arguments to convince the reader that it is possible to implement the algorithm

correctly.

Another reason for not being as interested in the low level implementation details of resilient update algorithms is the following one. The redundant update algorithm is only part of the complete distributed database system. Crash recovery techniques should not be designed for each part of the system independently, but should be integrated in a uniform fashion into the complete system. Therefore, there is little use in going through the implementation details of crash recovery when we are only studying the redundant update algorithms. In this chapter, the only reason for considering crash recovery techniques for an update algorithm independently of the rest of the system is to convince the reader that

1) it is possible to have a resilient centralized locking algorithm, and
2) this algorithm performs well.

If we can show that the above is true, then we can reasonably expect the centralized locking algorithm incorporated into a complete system with crash recovery to perform well.

2. TYPES OF FAILURES.

In this section we will classify the possible types of failures in a distributed database system. A good understanding of the possible failure modes will be helpful in the following sections. The types of failures that follow are not disjoint. That is, a single failure can be classified into several categories. (For another discussion of possible failures and their problems, see [ROTH77].)

(a) Node Failure. This occurs when a node in the system "crashes" and temporarily ceases operation. Node failures can be further grouped into "soft" and "hard" failures. In a soft crash, the node does not lose its vital data and it is possible for the node to get up-to-date by simply performing the updates it missed while it was down. In a hard crash, the node loses some data (e.g., values in the database) and backup techniques must be used to repair the damage.

(b) Communication Line Failures. This occurs when messages are not transmitted properly. If we assume that the distributed database system uses a communications sub-system, then this sub-system will take care of many of the failures. Some common failures that are "solved" by the communications subsystem are missing messages, out of order messages (both solved with sequence numbers), and physical line failures (solved by re-routing). In some cases, the communications sub-system will be unable to deliver or receive messages and one

or more nodes will be temporarily isolated from the rest of the network. In such cases, we say that a network partition has occurred.

(c) Detectable Failures. If all nodes in the system are able to discover a failure (node or communication) before the failure is corrected (the node brought up or the communication re-established), then the nodes can take appropriate action and the failure is called "detectable". Failures are usually detected by time-outs or through "alerting" messages from other nodes. A failure that is detected too late (i.e., an undetected failure) can cause the database consistency to be violated. (For example, see section 4.7.)

(d) Malevolent Failures. In order to be able to recover from failures, we must assume that all nodes cooperate and follow the system protocols. If a node does not do so, it can cause a malevolent failure. These failures are almost impossible to correct. For example, if a node starts broadcasting "perform update" (or "accept update") messages with garbage in them, then all databases will contain garbage. As another example, consider what would happen if the central node in the MCLA centralized locking algorithm started granting locks to all update requests without checking for conflicts.

(e) Multiple Failures. The failures mentioned above can occur one at a time or multiple failures can occur at once. Algorithms that recover from multiple failures are more complex because the number of cases that must be considered increases tremendously. Furthermore, the case of failures occurring during recovery from a different set of failures must also be considered.

In this chapter, we will not consider certain types of failures. We will not consider malevolent failures because crash recovery techniques are not adequate for this type of failures. Techniques from the area of protection and security are probably more useful. We will only consider communication failures that partition the network. (Note: A partition might just have one node.) We assume that the communication sub-system will effectively deal with the other communication failures. The resilient algorithms we discuss in this chapter will attempt to recover properly from all the other types of failures mentioned.

3. BASIC CONCEPTS.

In this section we will discuss some basic concepts common to all resilient update algorithms. For simplicity, we are still assuming a completely duplicated database at each node and we are also assuming that all transactions are updates.

However, many of the ideas and concepts we will present can be extended to the more general case of partially duplicated data and read-only and update transactions.

3.1 The Principal Idea.

The principal idea behind the resilient algorithms is to reduce the overhead to a minimum during normal operation. Of course, some additional steps must be performed during normal operation in order to allow recovery from a crash. But these additional operations should increase the number of messages transmitted and the IO and CPU overhead as little as possible. Only when a failure occurs will the algorithms temporarily operate less efficiently. But after a brief recovery and reconfiguration period after the crash, the algorithm should return to its normal operation, even if nodes are missing. If we make the reasonable assumption that failures do not occur frequently, then the overall performance of the algorithm will not be affected by these rare and brief periods of inefficient operation.

3.2 Logs.

Consider the crash of a single node in any update algorithm. Since this node will be out of operation for a certain period of time, it will miss a set of updates. This means that somehow the rest of the network will have to save these updates. There are many alternatives as to how and by whom these updates are saved.

One alternative is to assume that the communication sub-system will guarantee delivery of a message that has been given to it. (The mail system in the ARPA net is an example of such a communication system.) Thus, all missed update messages are saved by the communication sub-system and when the node comes up after the crash, it automatically starts receiving the messages as if nothing had happened. Although this can be a useful technique in some systems, we will not consider it here because we would like to solve the problem of missing updates explicitly. Therefore, we assume that the communication sub-system will attempt to send a message and will later inform the sender either "Yes, I sent it" or "No, I could not send it".

If the communication sub-system does not save updates, the nodes themselves must save them. Thus we see the need for "logs". A log is a collection of performed updates that is "safely" kept by a node. Each log entry contains

the database values that were modified by the update plus some sequencing information (e.g., a sequence number or a timestamp). Now the question is: What nodes save what updates?

One answer to this question is to have each node only save the updates that are "finalized" at that node. A node finalizes an update when it decides that the update should be performed. In the centralized locking algorithm, a node would only save the updates that originated there, while in the distributed voting algorithm, a node would only save updates that are accepted at that node. This strategy saves space in logs since each update only appears in one log. The obvious disadvantage of this scheme is that when a node crashes, its log is not available and the updates in that log cannot be accessed by another recovering node.

A second alternative is to have all nodes save all updates they ever perform, including updates that were finalized at other nodes. Although this is more wasteful in terms of space and time, a recovering node will always have access to all the updates it missed (except if all other nodes are down or unavailable).

A third alternative is to name a subset of reliable nodes to be the keepers of the logs. Only these nodes would save all updates; the rest of the nodes would not save any of the performed updates. Hopefully, at least one of the log keepers would be up at all times. Unfortunately, before completing an update, a node would have to wait for a confirmation from the log keepers that the update has been properly saved. (If the number of log keepers is larger than or equal to the number of nodes required to form a majority, then one of the logs will always be available and no confirmation is needed. The reason why this is true will become evident after reading section 3.4.)

In some systems it might be convenient to keep a record of all previous updates. For example, the log could be stored on magnetic tape which provides inexpensive, reliable and abundant storage. However, in other systems it might not be possible to keep a continuously growing log. Fortunately, updates can be removed from the log when we are positive that all nodes (active or down) have performed an update. There are several ways to do this trimming of the logs. One way would be to have the node in charge of each log periodically send out a message to all nodes requesting a list of performed updates. Then each update that has been performed at all nodes can be removed from the log. In the MCLA-h algorithm in particular (see chapter 3), there is an even simpler way of trimming the log. Suppose that update A, with sequence number a, is in the log at node x. Then suppose that a "perform update" message from node y for update B (with sequence number b) arrives at node x. Furthermore, assume that

$b > a$ and assume that A is not in B's hole list. In this case, update A must have been performed at node y and the "perform update" message for B acts as a confirmation that A has been performed at y. When node x receives similar messages from all nodes, update A can be removed from the log at node x.

Our comments on saving performed updates for crashed nodes apply to any update algorithm. Thus, we can expect any resilient update algorithm to have some sort of logging mechanism, and the overhead produced by the logs will be very similar in all algorithms.

3.3 Broadcast of Updates.

Another common problem to all update algorithms is the reliable broadcast of an update to all nodes once the update has been "finalized". Recall that an update is finalized when a node decides that the update should be performed. This happens when a node accepts an update with a majority of votes in the distributed voting algorithm. In a centralized locking algorithm (e.g., MCLA), an update is finalized when a node obtains locks for all the items referenced and computes the update values.

No matter what algorithm was used to decide if an update can be performed, once we do decide to perform it, we would like either that all nodes perform the update or that no nodes perform it. There are two basic alternatives for accomplishing this.

The first alternative is not to perform any update at any node until we can guarantee that that update will eventually be performed at all nodes in the system. There are many ways to achieve this, but most of them are variations of the "two phase commit protocol" [GRAY77]. Under this protocol, a node that has finalized an update and wishes to perform it, first sends the new update values to all nodes. The nodes save these values without updating the database. When the finalizing node receives acknowledgments from all nodes, it can be sure that all nodes received and accepted the values, and only then does it transmit a "commit" message to all nodes. Upon receipt of this second message, nodes actually perform the update on the database. By taking the proper precautions, this protocol can guarantee that either the update is eventually performed everywhere or the update is not performed at all. The disadvantage of this two phase commit protocol is that $3(N-1)$ messages are needed in order to broadcast an update instead of the minimum of $(N-1)$ messages needed without the protocol (where N is the number of nodes). (In some algorithms, the finalizing node only has to

wait for a majority of acknowledgments.) In section 4 we will discuss how this protocol can be used in the centralized locking algorithm.

The second alternative to the problem of reliable broadcast of an update is to allow "undoing" of updates. Undoing an update A at node x is not especially hard. In the log entry for A (stored at x or at some other node), we simply add the old values of the updated items. (The old values have already been read by whatever node computed the new values, so there is little overhead in adding these values to the log.) Update A can then be undone by writing the old values of the items into the database. Notice that other updates with larger sequence numbers (or larger timestamps) than A's and whose base sets have elements in common with A's base set, must also be undone.

If we can undo updates, a node can perform an update without knowing for sure if the update will be performed at all nodes. If later on the node discovers that an update it performed was not performed at the other nodes, it will undo the update. This means that a node that finalizes an update can simply send $(N - 1)$ "perform update" messages to all nodes and without waiting for acknowledgments can proceed to update the local database and to mark the update as finished. However, this also means that transactions at that node will see values in the database that will possibly be undone in the future. Because of this, undoing updates does not seem to be a satisfactory alternative for most cases.

(Notice that in some special cases undoing updates may not be necessary, even if we do not use a two phase commit protocol. For example, if all transactions are commutative (i.e., the order in which they are performed is unimportant), then a node that discovers an update transaction which was not performed at all nodes, does not have to undo the transaction. The node simply makes sure that all nodes do perform the update. The fact that the update is performed in a different order (with respect to other updates) is not important. See chapter 8.)

At this point some readers might think that the distributed voting algorithm can operate safely without update undoing or without a two phase commit protocol, thus making it superior to the centralized locking algorithm. As we will see in the next two sections, this is not true if we want the algorithm to operate efficiently even when nodes are inaccessible. In other words, the distributed voting algorithm (as well as the other algorithms) can operate without undoing updates and without a two phase commit protocol but the price that must be paid is inefficient operation when one or more nodes are down.

3.4 The Majority of Nodes Requirement.

Suppose that the communication sub-system fails and partitions the nodes of the network into independent groups. A node in one of these groups can only communicate with nodes in the same group, and from their point of view, it seems that the rest of the nodes crashed. We now address the following question: In what groups will we allow new updates to be performed?

If we do not allow updates to be undone, then we cannot permit a group smaller than a majority to perform any new updates because such a group cannot guarantee that their updates will be performed at the other nodes. (In an N node network, a majority consists of $(N/2)+1$ nodes if N is even or $(N+1)/2$ nodes if N is odd.) For example, consider a six node network that is partitioned into two groups with three nodes each. If we allow each group to continue performing new updates, the databases in the two groups will diverge (i.e., will contain different values) and there will be no way of re-uniting the network without undoing some updates.

On the other hand, if we allow updates to be undone, then every group of nodes in a partitioned network can perform new updates. However, when the network is re-united, many of the updates will have to be undone. Performing updates that are possibly going to be undone is a wasteful strategy. Furthermore, coordinating the undoings as the network is united is a very messy and hard problem. Therefore, even if updates can be undone, it is wise not to perform new updates in groups of nodes that are not a majority.

Thus, in general, we will not permit groups of nodes without a majority to perform any new updates. On the other hand, if there is a group with a majority of nodes, then we will require that that group perform new updates. This is in line with our philosophy that a system should operate as efficiently as possible, even after failures have isolated some nodes.

3.5 Cancelling Updates.

If a majority group of nodes is to operate efficiently, it is necessary for the group to be able to cancel or invalidate any updates that had been initiated but never completed by nodes outside the group.

For example, consider the case where a node x becomes isolated (due to a crash or to a communications failure) from the majority group. Also assume that the system is using a centralized locking algorithm (e.g. the MCLA algorithm)

and assume that x is not the central node. When node x was cut off, it was holding locks for some of its updates in progress. Since node x is isolated, its locks will be held until x can communicate again. And since we cannot allow other updates in the main group to be delayed indefinitely because of node x's locks, we need a mechanism for reclaiming the locks. This mechanism must also insure that the updates that lost their locks (i.e., were cancelled) are not performed at all, even when node x come up again. We will describe such a mechanism when we describe the resilient centralized locking algorithm.

The need to cancel updates also arises in other update algorithms. For example, consider the distributed voting algorithm with five nodes. Suppose that an update A arrives at node 1 and then proceeds to obtain OK votes at nodes 1, 2, and 3. At that point, node 3 accepts update A (e.g., finalizes it) but at the same instant node 3 is cut off from the rest of the system as in the example above. Since node 3 does not have a chance to transmit any messages informing the other nodes of the acceptance of A, node 2 will later time out and will send a vote request for A to say node 4. However, because of a second update B which conflicts with A and which has received OK votes at nodes 4 and 5, update A receives deadlock reject votes (DR) at nodes 4 and 5. Now update A has two OK and two DR votes and needs another vote to decide its fate. Update B is delayed at node 1 waiting for update A to complete, and similarly other updates can be waiting for A and B. So unless the four up nodes do something about update A, the system will slow down. The only solution (if we do not allow undoing) is for the four up nodes (1, 2, 4 and 5) to together decide to cancel or reject update A, thus permitting other waiting updates like B to complete. The four up nodes can cancel A because they constitute a majority of nodes.

Notice that this example illustrates the need for some sort of two phase commit protocol in the distributed voting algorithm (unless updates can be undone). If node 3 performs update A locally before making sure that the rest of the nodes have gotten its "accept" message, then node 3 will be performing an update that will later be cancelled by the other nodes. (Notice that it might be possible to modify the distributed voting algorithm in order to simplify the two phase commit protocol. When an update is accepted, a majority of nodes must know of the existence of the update. Thus, if done properly, the voting phase might serve as the first phase of the two phase commit protocol.)

Similar examples can also be constructed for all the other algorithms described in chapter 3. Hence, it seems that all update algorithms will need some form of two phase commit protocol and some type of update cancelling protocol (unless we allow update undoing).

4. THE RCLA-T ALGORITHM.

In this section we will present a resilient centralized locking algorithm that is efficient during normal system operation. We will call this the RCLA-T algorithm because this algorithm uses a variant of the two phase commit protocol and does not undo any updates. This algorithm will work with any of the logging schemes we have described, but for simplicity we assume that all nodes log all the updates they perform. Updates will be performed under this algorithm as long as a majority of nodes are up and able to communicate with each other.

The RCLA-T algorithm does not use any explicit "backup" central nodes. A backup central node would be a special node that is kept informed of all of the central node's activities, so that in case the central node fails, the backup node can immediately replace it. The idea of using backup central nodes is very intuitive but has a big disadvantage: keeping the backups up to date on the central node's activities introduces additional overhead and delays even during normal system operation. The RCLA-T algorithm eliminates this type of overhead by not using backup nodes. The price that must be paid for this is higher overhead when the central node fails and must be replaced. We prefer this alternative because we are assuming that failures do not occur often. Instead of using backups, the RCLA-T algorithm allows any node to become a new central node after the failure of the old central node. Before starting operation, the new central node must complete any pending work started by its predecessor. The details of this will be given in section 4.4.

As we will see, the RCLA-T algorithm can recover from hard crashes where important state information is lost. However, the recovery from these crashes is more involved. In order to simplify the presentation, for the time being we assume that no hard crashes occur. Then, in section 4.7 we consider the modifications needed to recover from these hard crashes.

4.1 The Two Phase Commit Protocol for Performing Updates.

Under normal operation, the RCLA-T algorithm is simply the MCLA-h algorithm with two modifications: logs of performed updates are kept and a two phase commit protocol is used to perform an update at all nodes after the update has been finalized. (Recall that the MCLA-h algorithm was described in chapter 3.)

Informally, the procedure for the two phase commit protocol is as follows:

When node x receives the locks for update A from the central node, it proceeds to compute the update values. (See appendix 1.) Then, before performing A locally, node x sends "intend to perform A" messages to all nodes. The contents of this message are similar to the contents of the "perform update" message in the original algorithm, except that now when the other nodes receive the "intend to perform A" message, they do not perform update A. Instead, they save A and its new update values in a safe place, they acknowledge receipt of the "intend to perform A" message to node x, and they await a "commit A" message. No node will perform A without having seen this second message. But all nodes will "remember" A until either they see the "commit A" message or they realize that A has been cancelled. The saved "intend to perform A" message is called a "pending" message at the node. Later on we will see other types of pending or saved messages.

After node x sends the $N-1$ "intend to perform A" messages (where N is the number of nodes), it waits until it receives a majority of acknowledgments from the other nodes. (Since node x has obviously seen the "intend to perform A" message, it only waits for $\lfloor N/2 \rfloor$ acknowledgments from the other nodes.) After receiving a majority of acknowledgments, node x can guarantee that A will be performed and can proceed to update the local database copy and to send the $N-1$ "commit A" messages. (Node x can ignore any acknowledgments for the "intend to perform A" message that arrive later.) After sending the "commit A" messages, node x is done with update A.

Node x can perform update A when it has only heard a majority of confirmations of its "intend to perform A" message because at that point node x can guarantee that update A will eventually be performed everywhere in the system. Update A will eventually be performed everywhere for two reasons: (1) Update A can no longer be cancelled, and (2) all nodes implicitly know about update A because of the sequence number mechanism. (See chapter 3.) As we will see in section 4.2, update A can only be cancelled in a majority of nodes have not heard anything about A. Since at least one member of any majority whatsoever has seen the "intend to perform A" message, then we know that update A can no longer be cancelled. Furthermore, update A was assigned a unique sequence number by the central node, so every node in the system is expecting to see "intend to perform A" and "commit A" messages. If these messages do not arrive at any node y, then that node will take the necessary actions to insure that A is performed at y. Again, since at least one node in any working majority of nodes has seen A, then node y will be able to get the necessary information to perform A from that node that has seen A. Therefore, update A will eventually

be performed at all nodes.

The two phase commit protocol described above and the logs are the only two crash recovery mechanisms that produce overhead during normal system operation. The rest of the mechanisms we will describe for the RCLA-T algorithm are invoked when a failure occurs and are only used for short periods of time while the system adapts itself to the new situation. The system uses time outs and invokes the failure mechanisms whenever it notices that something is not functioning properly.

4.2 Update Cancelling Protocol.

We now describe the procedure for cancelling updates. The procedure could be called a three phase commit protocol. The reason that three phases are needed is that when the procedure is initiated by the central node, it does not know whether the update can be cancelled. Thus, a first phase is required to find out if an update can be cancelled. After this phase, two more phases are required to actually perform the cancellation.

The cancelling procedure is invoked by the central node when it notices that one or more nodes are not responding. The central node can also be asked to start this procedure by other nodes that have waited too long for a certain update. In the following discussion, we assume for simplicity that one update only is being cancelled. This procedure can be generalized to allow the cancellation of several updates at a time.

Initially, the central node realizes that update A is missing and decides to try to cancel it in order to reclaim A's locks. The central node send a "propose to cancel A" message to all nodes in the system and waits for confirmations.

When a node x receives the "propose to cancel A" message, it checks to see if it has "seen" update A before. A node x has "seen" update A if node x has gotten a "intend to perform A" message or if node x has actually performed A. If node x has seen update A, it immediately sends a "have seen A" message to the central node informing it of this fact. (Node x also sends the central node the update values for A.) Upon receipt of this message, the central node aborts the cancel procedure. The procedure for aborting is described later on in this section.

If node x has not seen update A, then it sends a "have seen proposal to cancel A" message back to the central node. With that message, node x makes a commitment not to acknowledge any "intend to perform A" messages it might

receive later. Thus, node x must remember the pending "propose to cancel A" message until it hears from the central node again.

When the central node receives a majority of "have seen proposal to cancel A" messages, it knows that it is impossible for update A to be performed and hence A can be cancelled. However, since failures could occur before the protocol completes, the central node must still use an additional two phases to actually cancel update A. Therefore, the central node sends "intend to cancel A" messages to all nodes. When receipt of these messages is acknowledged by a majority of nodes, the central node sends out a "cancel A" message to all nodes.

A node that receives and acknowledges a "intend to cancel A" message knows that update A cannot be performed. When the central node receives a majority of acknowledgments for the "intend to cancel A" message, it knows that a majority of nodes know that update A cannot be performed. Only at that point can the central node guarantee that in any possible majority of nodes, at least one node will know that A cannot be performed. In other words, if the central node fails after this point, any other node that becomes the central node will be able to find out that A cannot be performed and will thus finish the cancelling procedure that was not terminated. (This aspect will be described in more detail in section 4.4.) So after receiving a majority of acknowledgments for the "intend to cancel A" message, the central node can send out the "cancel A" messages which actually cancel the update.

The effect of the "cancel A" message received at node x is very similar to the effect of a "commit A" message, except that no actual values are stored in the database. That is, update A is recorded as a null update, its sequence number is added to the list of performed updates and a log entry for A is made. The log entry for A indicates that update A is a null update. After having sent out the "cancel A" messages, the central node releases A's locks and makes them available to other updates.

As we stated previously, the central node may decide to abort the cancel procedure during the first phase if it discovers that update A has been seen by a node. (The cancel procedure will not be aborted after the second phase has started.) In addition to aborting the cancelling procedure, the central node should make sure that all nodes perform update A. Since the central node now has a copy of the update values needed for performing A (obtained from the node that had seen A), it uses the following two phase protocol to abort the cancelling procedure and to perform A at all nodes: First a "force performance of A" message is sent to all nodes. This message is similar to the "intend to perform A" message except that the "force performance of A" message overrides any "propose to cancel A"

pending messages at the nodes. In other words, any node that had a "propose to cancel A" pending message simply forgets about this message and remembers the new message. If both a "force performance of A" and a "intend to perform A" messages arrive at a node (in any order), the node only has to remember the "force performance of A" message because it implies the existence of the "intend to perform A" message. After a node has processed the "force performance of A" message, it sends an acknowledgment to the central node. When the central node receives a majority of acknowledgments, it sends out "commit A" messages which cause A to actually be performed at all nodes.

There are a few details we have omitted in our description of the cancelling protocol. It is possible that a node z that has seen update A, receives an "intend to cancel A" message from the central node. This can occur is the "have seen A" message sent by node z arrives at the central node after the central node has gone into the second phase of the cancelling protocol. When node z receives the "intend to cancel A" message, it can acknowledge the message and node z can pretend that it never saw A because at that point node z knows that update A cannot be performed.

It is also possible for a node to receive an "intend to perform A" message after it has received a "cancel A" message. This can occur if A's originating node is not aware that update A has been cancelled and is still trying to perform it. We solve this problem by requiring nodes not to acknowledge an "intend to perform A" message if A has already been performed (i.e., cancelled).

If the central node cannot get a majority of acknowledgments in the first phase, but gets no "have seen A" messages either, then the central node has lost its majority of nodes and should go into central node recovery procedure described in section 4.6. Similarly, if the central node fails to get a majority of acknowledgments for its "intend to cancel" or "force performance" messages, it has also lost its majority. After we describe how a new central node is elected, we will describe how an unfinished cancelling procedure is terminated by the new central node.

4.3 State Diagrams.

We summarize the two phase commit protocol and the cancelling protocol by giving the "state diagrams" for the nodes. Table 7.1 presents the diagram for the central node, while table 7.2 gives the state diagram for any non-central node. In these diagrams we list the possible states of a node (with respect to

TABLE 7.1
STATE DIAGRAM FOR THE CENTRAL NODE

(The top entry in each square is the next state; the bottom entry is the message that must be sent out to all nodes. "*" means that the event cannot occur in that state.)

EVENT ==> / STATE v	arrival of "please cancel"	Majority of acks for "propose to cancel"	Majority of acks for "intend to cancel"	arrival of "have seen"	Majority of acks for "force performance"	"failure"
idle	may cancel "propose to cancel"	*	*	*	*	see (1) below
may cancel	may cancel none	will cancel "intend to cancel"	*	will perform "force performance"	*	see (2) below
will cancel	will cancel none	*	can-celled "cancel"	will cancel none	*	will cancel "intend to cancel"
can-celled	can-celled none	*	*	can-celled none	*	can-celled "cancel"
will perform	will perform none	*	*	will perform none	per-formed "commit"	will perform "force per-formance"
per-formed	per-formed none	*	*	per-formed none	*	per-formed "commit"

(1) In this case, there are two possible next states. The successor central node can either remain "idle" (and not send out any messages) or it can go to state "will perform" (and send out "force performance" messages).

(2) In this case there are also two possible outcomes. The successor central node can either go to "will cancel" (sending "intend to cancel" messages) or it can go to "will perform" (sending out "force performance" messages).

TABLE 7.2
STATE DIAGRAM FOR THE NON-CENTRAL NODES

(The top entry in each square is the next state; the bottom entry is the message that must be sent out in response to the event. The response message is sent to the node that sent the message causing the event. "*" means that the event cannot occur in that state.)

EVENT ==> / STATE	arrival of "intend to perform"	arrival of "commit"	arrival of "force per-formance"	arrival of "propose to cancel"	arrival of "intend to cancel"	arrival of "cancel"
idle	seen ack	per-formed none	will perform ack	may cancel ack	will cancel ack	can-celled none
seen	seen ack	per-formed none	will perform ack	seen "have seen"	will cancel ack	can-celled none
per-formed	per-formed none	per-formed none	per-formed none	per-formed "have seen"	*	*
will perform	will perform none	per-formed none	will perform ack	*	*	*
may cancel	may cancel none	per-formed none	will perform ack	may cancel ack	will cancel ack	can-celled none
will cancel	will cancel none	*	*	will cancel none	will cancel ack	can-celled none
can-celled	can-celled none	*	*	can-celled none	can-celled none	can-celled none

one update) on the left. Across the top, we give the possible events that might occur at the node. Each square in the diagram represents the action that must be taken by the node when it is in the given state and the listed event occurs. For example, if a non-central node is in state "seen" (i.e., it has seen the "intend to perform" message for the update) and a "intend to cancel" message arrives for that update, then the node must send an acknowledgment to the central node and must move to state "will cancel". If a square is marked with an asterisk, this means that the listed event should not occur when the node is in the given state. (Since we have assumed that there is no loss of state information, if the actions in a square are interrupted by a failure, then these actions will be completed correctly when the node comes up again.)

In the state diagram for the central node, there is a special column marked with the event "failure". The meaning of this column will become clearer after having read section 4.4. At this point, simply notice that squares in this column have a slightly different meaning from the other squares. The next state indicated in these positions is the state that the successor central node must assume after a crash of the old central node. Also notice that in some cases there are two possible next states. Which next state is assumed by the successor node after the crash depends on the nature of the crash. Finally, observe in the central node state diagram that the central node always advances towards "performed" or "cancelled" states, and never goes back to previously visited states.

4.4 The Election Protocol.

We now describe the mechanism for electing a new central node when the old one fails or is isolated from the majority of the nodes. There are two basic steps in this procedure: (1) Elect a new central node that can communicate with a majority of nodes and (2) The new central node collects "state" information from all active nodes and completes all unfinished updates or cancellations. After these two steps, normal system operation can resume.

There are many alternatives for the election procedure. Some of these alternatives can be quite involved if they are to work while new failures appear and old failures disappear. Fortunately, the election procedure does not have to be particularly efficient, as long as it guarantees that one and only one new central node will be elected in the system.

The solution we present here is not the most efficient but is relatively simple and safe. If the system does not change while the election procedure is in progress,

then our election procedure will certainly produce a new unique central node. If the environment is changing, the procedure might fail to elect a new central node. In this case, the procedure is repeated until it succeeds. If the system remains unstable (e.g., new failures appear or old failures disappear constantly), then a new central node might never be elected during the unstable time.

The election procedure is based on the use of node identification numbers and central node version numbers. We assume that all nodes in the system have a predefined identification number. These integers are unique and permanent and will be used as priority numbers in the election procedure. (Actually, we could do away with node identification numbers by modifying the algorithm slightly. However, we believe that these number make the algorithm more intuitive.)

The central node version number is an integer that identifies an instantiation of the central controller and is used to distinguish between different instantiations. The version number is appended to all update sequence numbers generated by the central node. It is therefore possible to distinguish updates whose locks were granted by different central nodes by simply comparing the version numbers in each update. During normal operation, all active nodes have a copy of the central node's version number and will reject any messages regarding updates with another version number. When a new central node is elected, it will choose a version number that is larger than any previously used version number and it will distribute this new number to all active nodes. Notice that sequence numbers for a given version number can start at zero.

If we assume that failures of the central node are not very common, then a small number of bits should be enough to represent version numbers. Hence the increased overhead in transmitting these numbers should not be significant. Notice that if several sequence numbers are sent in a single message (e.g., a hole list), they must all have the same version number and only one copy of this number is needed in the message. It is possible to design an algorithm that reuses version numbers. A version number can be reused if we are positive that all nodes, active or not, have performed all updates authorized during the existence of that version number. For simplicity, we do not consider this algorithm and we assume that version numbers are used in increasing order without recycling.

4.4.1 The Election Protocol – First Part.

The first part of the election protocol elects a new central node. The basic idea here is to have "candidate" nodes attempt to become the new central node. A node x is a candidate if it can communicate with a majority of other nodes

that have also lost contact with the central node and node x has the largest node identification number of the set. Under certain circumstances, several nodes might attempt to become central nodes simultaneously. Therefore, attempts to become a new central node can fail and must be repeated if necessary.

In the following discussion, let us assume that due to failures in the communications system, it is possible to have one way communications only. That is, a node x might be able to communicate with node y, but node y may be unable to communicate with node x. The three other cases are that both nodes x and y can communicate with each other, that neither one of the nodes can communicate with each other, and that y can communicate with x but not viceversa.

The election of a new central node works as follows. When a node x discovers that it cannot communicate with the current central node, it immediately suspends all normal activities and goes into failure mode. (We may allow some restricted read-only transaction to run in failure mode.) Then, every t_1 seconds, node x will send out a "What is going on?" message to all nodes. When another node y receives such a message, it will acknowledge the message and will inform x if it (y) can communicate with a central node. (Node y could also be sending out its own "What is going on?" messages simultaneously.) Thus, every t_1 seconds, node x constructs an "active table" which indicates what it can communicate with. If this active table indicates that node x can now communicate with a node that calls itself the central node, then node x initiates the crash recovery procedure described in section 4.5. If on the other hand, node x discovers that it can communicate with a majority of other failed nodes and that node x has the highest node identification number, then node x becomes a candidate and attempts to become the new central node. If node x is not a candidate and cannot communicate with a central node, it waits t_1 seconds and constructs a new active table. (To see how several nodes might believe that they are candidates, consider a three node system that is electing a new central node. Node 3 (highest priority) can communicate with all other nodes and thinks that it is a candidate. Node 2 can only communicate with node 1 and therefore also believes that it is a candidate.)

A candidate node attempts to become the new central node by sending out the message "Node x proposes to become new central node" to all nodes in x's active table. When a node y receives a "Node x proposes to become new central node" message, it will send out a confirmation (which we call a a vote) to node x if node y thinks that node x can become the new central node. The vote will only be sent if y has not "recently" sent a vote to some other node. (What we mean by recently will be explained shortly.) In other words, node x is attempting to

"lock" all the nodes and only if it succeeds in locking all of them, will x become the new central node. Every node y sending a vote will also include the latest version number seen by it. This way, if node x becomes a new central node, it can pick a version number larger than any seen by the nodes.

If node x does not get votes from all nodes in its active table, it must release its "locks" by sending a "I did not make it" message to all nodes. Unfortunately, because of some new failure, these messages might not reach their destinations. We therefore need a default mechanism for freeing nodes after they have confirmed or voted for a "Node x proposes to become new central node" message. We will say that a node y that sent such a vote will honor it for only t_2 seconds. During these t_2 seconds, node y will await news of the election from node x and will ignore all other messages. When t_2 seconds go by without hearing from node x, node y will be free to send another vote to some other node that requests it.

If node x can send out the "Node x proposes to become new central node" messages, receive all the votes, inform all the nodes that it succeeded, and receive a second acknowledgment from the nodes, all in less than t_2 seconds, then node x can be sure that it became the new central node. If node x cannot do all this in less than t_2 seconds, then it must assume that it failed and node x should send out "I did not make it" messages. Notice that as long as node x does not take any actions regarding the updates, it is free to quit. If node x quits, nodes that sent a vote to x will time out after t_2 seconds; nodes that sent a vote to x and though that x had become the new central node, will simply go into failure mode again when they realize that node x is not the central node.

If the clocks or timers at the nodes do not advance at the same rate, then the limit for node x to become the central node will be less than t_2. Let d be the maximum number of seconds that any two clocks can diverge in t_2 real seconds. Then, if node x completes the election in less than $t_2 - d$ seconds, it can be sure that no node voted for another candidate.

After node x obtained votes from all nodes, it selects a version number. The version number it selects is one plus the largest version number reported by the voting nodes. Then node x informs all active nodes of its success in the election and gives them a copy of the new version number. After getting a new acknowledgment from all nodes, node x can guarantee that it is the only central node and that all active nodes have the new version number. Furthermore, any future central node will use a larger version number because a majority of nodes have seen the current one.

If node x fails before distributing the new version number to a majority of the nodes, then it is possible that the same version number will be chosen in a

future election. This is no problem because in that case, the version number had not been used for anything yet. However, after x distributes the version number to all active nodes, it is sure that no other central node will use that number and therefore node x can start the second part of the election procedure.

4.4.2 The Election Protocol – Second Part.

Before node x, a newly elected central node, can authorize any updates, it must make sure that all old updates are either completed or cancelled. This is the second step of the election procedure.

Suppose that the current version number is i. Then node x requests all "state" information for version $i-1$ from all active nodes. The state information for node y includes (a) the list of all version $i-1$ updates performed by y and (b) the list of all pending "intend to perform A", "propose to cancel A", "intend to cancel A", and "force performance of A" messages saved by y for any version $i-1$ update A. If both lists are empty for all nodes, then the central node for version $i-1$ failed before any updates were committed, so node x then requests the state information for versions $i-2, i-3, \ldots$ until it finds non empty lists. Say that the version with non empty lists is version j. Since the central node of version j authorized some updates, it means that it made sure that all previous versions were completed correctly and hence node x only has to check the correct completion of unfinished updates of version j. Even though all updates from versions $j-1$ and earlier have been completed correctly (e.g., a majority of nodes have performed them), some of the active nodes might not have seen them. Therefore, the central node x must first make sure that all active nodes are brought up to date up to version $j-1$. This is done using the protocol that is described in section 4.5.

In summary, this is the situation before the second part of the election procedure begins. All active nodes are in election mode and have halted all normal operations. Also, all active nodes in the current majority have completed all updates in versions 0 through $j-1$. The new central node, x, must make sure that all updates in version j are either performed or cancelled by a majority of nodes. To accomplish this, node x has all the state information of the active nodes. However, node x realizes that other previous central nodes with version numbers $j+1, j+2, \ldots i-1$ have also attempted to complete all updates in version j. These other central nodes did not finish their job (else they would have authorized some updates for their version). This means that the state information collected by node x from the other nodes not only contains pending messages that

originated at central node j, but may also contain pending messages regarding a version j update that originated at central nodes $j+1, j+2, \ldots i-1$ as they were trying to complete version j updates.

In order to make node x's job simpler, we would like to be able to distinguish what central node generated each pending message found. This can be done if we require that each central node sets a special flag in each message it sends during the election procedure. Each special message will include the current central node's version number as well as the version number of the update that it is trying to complete. Therefore, when node x collects all the state information, it can sort the messages by the version number of the central node that sent it. (All messages should deal with updates of version j only.)

The next step of the second part of the election procedure is to decide what to do with each update of version j. The new central node, x, considers each update at a time, independently from the others. The decision for update A is based only on the state information that involves update A. We now describe the procedure that node x follows in order to decide what to do with update A.

STEP 1. (Check if A has been performed everywhere.) The new central node, x, has collected all the state information for update A. If all the active nodes have performed A, then node x does not do anything else with update A.

STEP 2. (Check if A has been performed somewhere.) If update A has been performed at one or more active nodes (but not all), then the new central node must make sure that A is correctly performed at all nodes. This is done using the update values for A found in any log and with the two phase commit protocol that forces the performance of an update. (See section 4.2). Node x sends out "force performance of A" messages, and when a majority of these are acknowledged, it sends "commit A" messages and is then finished with A. Any pending messages involving A at the nodes can be ignored because we know that A must be performed. Notice that the update values for A found in an "intend to perform A" pending message could be different from the log values. The values in the pending message should be ignored. (Exercise for the reader: How can this situation occur?)

STEP 3. (Decide what to do with A. Initial step.) Otherwise, update A has not been performed at any active node, and the new central node, x, must decide what to do by examining the pending messages involving A. It starts by looking at those messages the were sent by central node version $i-1$, and will then examine the messages from versions $i-2, i-3, \ldots$ in turn, until node x can decide what to do. (Recall that i is the current version number.) Let k be the version number of the central node that sent the messages we are currently

examining. That is, initially set k to $i-1$.

STEP 4. (Analyze messages from central node version k.) Unless k is equal to j, there are only two types of pending messages that the new central node can observe. (Recall that j is the version number of the updates we are trying to complete.) This is true because the second part of the election procedure only "intend to cancel A" or "force performance of A" messages are sent. Notice that "commit A" and "cancel A" messages are not pending messages. If any of these messages exist at a node, the node reports to node x that it has performed A. (The case where $k = j$ is considered in step 5.) Furthermore, it is impossible to have both "intend to perform A" and "force performance of A" messages originating from central node version k because no central node will ever decide to both cancel and perform update A. Therefore, we only have three cases to consider. These are described in steps 4 A, 4 B, and 4 C:

STEP 4 A. (No messages.) Central node x does not observe any "intend to cancel A" or "force performance of A" messages from central node version k. Since no update is ever cancelled or performed without a majority of nodes knowing that this is going to happen, node x is sure that central node k did not perform or cancel update A. In this case, node x cannot decide anything yet and therefore sets k to $k-1$ and repeats step 4 in order to discover if any previous central node did anything.

STEP 4 B. (An "intend to cancel A" message observed.) Central node x observes at least one "intend to cancel A" message that was sent by central node version k. This means that central node version k had decided to cancel A and was therefore positive that A had not been previously performed at any node. Even though central node version k did not complete the cancelling procedure, we trust that it was doing the right thing before it crashed. Node x also knows that no later central node with version numbers $k+1, k+2, \ldots i-1$ performed A either (otherwise, this procedure that node x is executing would have stopped before reaching version k). Therefore, node x is positive that update A was never performed, so node x will complete the cancelling procedure. However, there is no need for the first phase of this procedure because node x already knows that A has not been performed and it knows that A cannot be performed in the future because all nodes have halted normal operation. (That is, no "intend to perform A" messages will be acknowledged.) Therefore, the new central node x sends "intend to cancel A" messages to all nodes. When node x gets a majority of acknowledgments, it can guarantee that any successor central node will see one of the "intend to cancel A" messages and hence the successor central node will complete the cancellation of A. So node x can then send out the "cancel A"

messages to complete the procedure for update A.

STEP 4 C. (A "force performance of A" message observed.) Central node x observes at least one "force performance of A" message that was sent by central node version k. This case is similar to the case of step 4 B, except that now node x is positive that update A was never cancelled. Therefore, node x will make sure that A is performed at all nodes by following the two phase commit protocol to force the performance of A. After getting acknowledgments for a majority of new "force performance of A" messages, node x can guarantee that A will be performed by any successor central node, so it sends out the "commit A" messages to all nodes and is finished with update A.

STEP 5. (Analyze messages from central node $k = j$.) Step 4 is repeated for values of $k = i-1, i-2$, and so on until a pending message sent by central node version k is found. At that point, the new central node x can decide what action to take with A and finishes the procedure. However, if no messages are found, node x can reach the last value of $k = j$. In this case, we can still decide what action to take.

When $k = j$, there are other messages that can be observed for this version number in addition to the "intend to cancel A" and the "force performance of A" messages. These are the "propose to cancel A" and the "intend to perform A" messages. However, if node x observes at least one "intend to cancel A" message, it can be sure that A has not been performed under version $k = j$. Since A was not performed under any other version, node x can proceed as in step 4 B above and cancel A. Similarly, if at least one "force performance of A" message is found, update A is performed as in step 4 C. If neither one of these messages is seen for version $k = j$, then node x knows that no previous central node took any specific action on update A. If any "intend to perform A" messages are found (sent during version j), then A might have been performed and the node x forces the performance of update A as in step 4 C above. If no "intend to perform A" messages are seen, then A was not performed, so node x cancels A as in step 4 B. This concludes processing of update A.

The process just described ensures that all updates whose existence is known by the central node are either cancelled or performed at all nodes. However, there might be some updates that were authorized by the central node version j but that the new central node x does not know existed. Let s be the largest sequence number of version j that was observed by node x in the collected state information. Then, node x knows that updates with sequence numbers 0 through s existed and the above procedure will deal with them. But it is also possible that updates $s+1, s+2, \ldots$ were authorized by central node version j and were

AD-A075 268 STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE F/G 12/1
PERFORMANCE OF UPDATE ALGORITHMS FOR REPLICATED DATA--ETC(U)
JUN 79 H GARCIA-MOLINA MDA903-77-C-0322
UNCLASSIFIED STAN-CS-79-744 N/L

2 OF 3
AD A075268

END
DATE FILMED
4-80
DTIC

in progress at some of the nodes that are isolated from the current majority of *nodes*.

Since the new central node x knows that none of the updates with sequence numbers $s+1, s+2, \ldots$ were ever performed (else node x would see some record of them), node x can cancel them all. Node x cannot cancel them individually because it does not know how many of these updates there are. But by executing a first dummy update in version i, all messages of previous versions will be automatically made void by the version number mechanism. (Recall that under *normal operation, nodes reject any messages* from older versions.) The dummy first update will also serve to (1) make sure that any node that recovers later will perform updates with sequence numbers 1 through s from version j, and (2) make sure that no other future central node should worry about correctly completing the updates of version j (or older).

Therefore, the last step in the second part of the election procedure is for the new central node x to perform a dummy "update" with sequence number 0 under version i. This update will not really be an update to the database, but will cause the largest sequence number s and the old version number j to be written in the logs. The log entry will later be used by recovering nodes to find out what updates from version j they missed. And by performing the first update in version i, we make a permanent record that all updates in versions $i-1$ and earlier have been performed correctly. Since updates $s+1, s+2, \ldots$ were never completed, this is a defacto cancellation of these updates.

The update with sequence number 0 and version number i is performed with the majority two phase commit protocol. (See section 4.1.) After update 0 is performed, a node can go back to normal operation. The node can then throw away any pending messages from versions older than i. When the central node performs update 0, it sets all item locks free and it can then start granting locks to new updates.

4.4.3 Some Comments on the Second Step of the Election Protocol.

When we discussed the mechanism for completing unfinished updates, we said that steps 1 through 5 were performed for each update individually. However, it is also possible for the new central node, x, to first decide what to do with every update, and then to send out a message which includes all the "intend to cancel" and "force performance" messages for the updates. An acknowledgment of this compound message by a node is equivalent to all the individual acknowledgments. Then all the "commit" and "cancel" messages can be sent out together.

Now let us briefly consider what happens if new failures occur during the *second step of the election procedure*. If a non central node fails or is isolated, it will simply be eliminated from the central node's list of active nodes. If this causes the central node to lose a majority of nodes, then the central node will go into failure mode. During the election, no new nodes will be allowed to join the majority. Since any messages from those nodes have old version numbers, it easy to ignore the messages until after the election. If the new central node fails before initiating update 0 of the new version i, the fixing up of version j will be continued by any successor central node. And since two phase commit protocols are being used by the new central node, there is no problem with leaving partially complete updates or cancellations. If node x, the new central node, crashes before a majority of nodes see update 0 version i, then the successor central node might or might not see this update. If the successor node does not see update 0, it will have to go back and check version j. There is no problem with this since no real updates in version i had been authorized yet. If the successor central node does see an update 0 version i that has not been performed at all nodes, then it will not check version j (which was not needed anyway) and will make sure that update 0 is performed at all nodes. Finally, if update 0 was completed by node x, then any successor central node will see it and will know that recovery from versions $i-1$ and earlier was completed correctly.

4.5 Non-Central Node Recovery Protocol.

A non-central node can have two types of failures: detected or undetected. A *node detects a failure if it is unable to communicate* with its current central node (i.e. with same version number), or if it realizes that a larger version number is in use by other nodes, or if the node is "told" by the operator that it has failed.

4.5.1 Detected Non-Central Node Failure.

Let us first study detected failures. After a detected failure, a node goes into failure mode where all normal activities are halted. The failed node will start constructing its active table until it either joins in an election or it is able to communicate with a central node. The first case has already been discussed. In the second case, the node will perform the following procedure in order to be brought up to date.

Let y be the node that is recovering. Let us assume that y remembers what updates it has already performed, and in particular, y knows that it has correctly performed all updates from versions j or less. Hence y is sure that it has not lost any updates from these versions. (Notice that j could be zero.) Let k be the version number at the central node that y has located.

To recover, y sends a "I would like to recover" message to the central node. If the central node is in the middle of the election procedure, it will tell node y to defer its recovery until the election is over and normal operation has been restored. When the central node receives the message, it also checks that it still is the central node by making sure that it can communicate with a majority of nodes. Next, the central node informs all the other active nodes that node y is now up and that they should start sending messages to y. (For a time, node y might not acknowledge these messages because it is busy catching up, but this causes no problem.) Then the central node informs node y that the current version number is k.

Next, node y requests update 0 version k from one of the logs. Suppose that update says: "Last sequence number for version m was s". Then y knows that it must perform all updates $1, 2, 3, \ldots s$ from version m, and that there are no updates from versions $m+1, m+2, \ldots k-1$. Next, node y requests update 0 version m from the logs and repeats the procedure until version j is reached. In this way, node y can discover the sequence and version numbers of all the updates it could have missed. By comparing this list to the list of updates it did perform, it can find out all the updates it missed. The update values for these updates can be requested from the logs and the updates can be performed.

Node y might have some updates from older versions in progress (i.e., the updates had originated at y, node y had obtained their locks, but node y had not reached the point where it had a majority of acknowledgments to the "intend to perform" messages). Since these updates originated at an old version, then they have certainly been cancelled, and node y should either throw them away and inform the users or node y should restart the updates from scratch. Any pending messages (e.g., "propose to cancel") from older versions can also be thrown away because the fate of these updates has already been decided.

Node y is now caught up to all previous versions (up to $k-1$), but still has to catch up to the current version k. However, node y does not have to worry about this because the rest of the protocols will automatically force y to perform any updates of version k that it missed. Therefore, node y can set its current version number to k and can become a normal member of the system. It can start acknowledging "intend to perform", "propose to cancel", "force performance" and

"intend to cancel" messages. When a "commit" or "cancel" message arrives, node y will request and perform the missing updates before committing or cancelling (because of the sequence number rule).

We still have to discuss what happens to any pending messages from version k that node y might have. These messages can exist if there was no central node version number change while node y was down. These messages definitely cannot be thrown away because node y is now part of the active majority and it must honor the commitments it made to remember these messages. Some of the messages might refer to updates that have already been performed or cancelled. Only in these cases can node y throw away the messages. This can be done by checking for pending messages as each missed update is performed.

If node y has any version k updates in progress (i.e., the updates had originated at y under version k, y had obtained their locks, but y had not obtained a majority of acknowledgments for the "intend to perform" messages), it can continue processing them. If the updates have been cancelled, node y will never be able to obtain a majority of acknowledgments for the "intend to perform" messages, and those updates will never be performed. In those cases, node y must either inform the users or must restart the updates from scratch.

4.5.2 Undetected Non-Central Node Failure.

We now discuss the case of an undetected non central node failure. This can occur if a non central node y temporarily halts but then resumes operation at exactly the place where it left off, without realizing that it was delayed. Fortunately, this case is no different than the case where the failure was detected.

If the central node was replaced while node y was down, then node y will immediately detect the failure because it has an old version number. And if the same central node is still active, then the normal protocols will bring node y up to date just as we described above.

4.6 Central Node Recovery Protocol.

A central node discovers that it has failed when it sees a higher version number in use by any node or when it finds that it cannot communicate with a majority of active (i.e., not failed) nodes. In that case, the central node stops acting like a central node and goes into failure mode like any other node would.

Recovery then is performed as described in the previous section.

In some cases where the network is partitioned, a central node and a minority of other nodes might continue to operate for some time before realizing that they have all failed. The central node can grant locks to updates, but these updates will never be performed because there is no majority to acknowledge the "intend to perform" messages. Therefore, whatever the central node and its minority try to do, it will be of no consequence to the rest of the system. The successor central node will cancel all the updates that were authorized after the failure. (See section 4.4.2.)

4.7 Recovery From Loss of State Information.

Up to this point in this chapter, we have assumed that no node loses its vital state information. This information includes the database, the list of updates that have been performed at the node, and any pending messages (e.g., "intend to perform A") at that node. In this section, we will discuss how a node can recover from the loss of its state information.

First we must assume that any loss of information is detected by the node that loses it. A node that loses state information and does not realize it, can easily cause the RCLA-T algorithm (and almost any other algorithm) to fail. (Note: If we change our definition of majority of nodes to mean the true majority plus m nodes more, then the RCLA-T algorithm can tolerate the simultaneous undetected loss of pending messages at m nodes. However, this modification does not prevent errors if the database or if the list of performed updates is lost or if erroneous pending messages can appear. This modification will not be discussed further in this thesis.)

It is possible to recover from a detected loss of state information, but the mechanism is more involved than the simple recovery without losses. First we will discuss recovery when only the pending messages have been lost. Later we will discuss the case where the list of performed updates is lost and finally we consider the case where data in the database is lost.

4.7.1 Loss of Pending Messages.

Suppose that node y is recovering from a crash where the pending messages have been lost. Also assume that the current version number is k. Since y's

pending messages are not needed in order to come up to date to version $k-1$, this part of the recovery protocol is unchanged. (See section 4.5.) When y is ready to start processing version k updates, it informs the central node that it is doing so without pending messages. The central node replies with the current sequence number that is being used. Suppose that this sequence number is s.

The rest of the recovery proceeds as before, except that node y does not acknowledge any messages involving updates with sequence numbers smaller than or equal to s. In other words, node y acts as if it were down for any messages involving these updates. Nevertheless, "commit" and "cancel" messages are processed by node y correctly for all updates.

Since node y is not responding in reference to updates with sequence numbers less than or equal to s, the central node must find a majority that does not include y before it takes any action on one of these updates (e.g., cancel it or force its performance). Therefore, all updates with sequence number smaller than or equal to s will be processed correctly.

If the central node cannot communicate with a majority of active nodes that have not lost their pending messages, then the system might operate inefficiently. For example, if node y is required by the central node in order to obtain a majority of nodes, then the central node will be unable to cancel updates with sequence numbers less than or equal to s. There does not seem to be a way around this problem. In such a case, the central node can decide to continue operation without cancelling these updates or it can decide to quit until it can find more nodes that have not lost their state information. (If a majority of nodes loose their state information regarding an update A, then the central node will never be able to perform or cancel update A.)

If the central node crashes before all updates with sequence number less than or equal to s are performed or canceled, node y must inform the successor central node of its loss of pending messages for updates with sequence number less than or equal to s. Thus, the new central node will not take any action on update A unless it finds a majority of nodes that have not lost pending messages regarding A.

4.7.2 Loss of the List of Performed Updates.

It is also possible to recover from the loss of the list of performed updates at a node y. (The list of performed updates consists of the sequence numbers of the updates that have been performed at that node.) If node y is keeping a log, this list can be reconstructed from the log. But let us suppose that either

there is no log at y or that the log has been destroyed. The loss of the list of performed updates can occur without the loss of the pending messages, but here we will consider the more general problem of losing both the pending messages and the list of performed updates.

Let us assume that every node periodically saves in a safe place a copy of the list of performed updates and the version number corresponding to that list. This is called checkpointing. The recovery of node y can then start from this checkpointed state. (If there is no checkpoint, we simply start with an empty list at version 0.)

To recover, node y simply assumes that the list of performed updates at the checkpoint is the current one and then proceeds as was described in section 4.7.1 above. Clearly, some updates will be performed twice at node y, but this causes absolutely no problems, as long as updates are performed in order of increasing version and sequence number. (Actually, the updates can be slightly out of order if their hole lists permit it. See chapter 3.)

To see why this last statement is true, consider a sequence of updates u_1, u_2, $u_3, \ldots u_n$ ordered by increasing version and sequence number. Database D_1 is obtained by applying the updates in order to an initial database D_0. Database D_2 is obtained by first applying a subset of the updates to D_0 and then applying all the updates in order. To show that D_2 has the same values as D_1, take any item i in the database. If none of the updates modify item i, then this item will have the original value of D_0 in both D_1 and D_2. Otherwise, let u_j be the last update in the sequence that modified item i. Let v be the value stored into item i by update u_j. Then item i in D_1 will have a value v and item i in D2 will also have a value of v. Therefore, both databases are identical.

4.7.3 Loss of Data in the Database.

A node y can also lose part or all of its database. Let us also assume that node y has also lost all pending messages and the list of performed updates (which is not very useful after the database has been ruined).

Node y has three basic alternatives. Each alternative could be useful in certain cases. The first alternative is to go back to a checkpoint database and its list of performed updates and then to recover from there as outlined in the previous sections.

The second alternative is to start from an empty database and then to perform all old updates to recreate the database. The last alternative is to copy the database from another node. It is interesting to note that new updates could be

performed at the source node while the copy operation was in progress. After the copy operation ends, node y would perform all the updates that occurred during the copy and the database would then be up to date and consistent. (To see why, consider the argument at the end of section 4.7.2.)

4.8 Summary of the RCLA-T Algorithm.

We have outlined the design of a resilient centralized locking algorithm that uses a two phase commit protocol to perform updates. We discussed the mechanisms for cancelling updates, for electing a new central node and for recovering from crashes. Although we did not prove the correctness of the RCLA-T algorithm, we hope that the reader is convinced that the algorithm could be implemented and that such an algorithm would operate correctly in the presence of failures (except malevolent failures which we did not consider).

5. PERFORMANCE OF THE RCLA-T ALGORITHM.

As was stated at the beginning of this chapter, we have tried to keep to a minimum the overhead of the RCLA-T algorithm during normal (e.g. without failures) operation. The two main sources of additional overhead during normal operation as compared to the original MCLA centralized locking algorithm, are the logging of the performed updates and the two phase commit protocol for performing updates. Also recall that in section 3.5 we showed that any resilient update algorithm would have at least this overhead during normal operation (unless we can undo updates). Therefore, when we compare the performance of the RCLA-T algorithm to other algorithms, we assume that these algorithms also have similar overhead.

5.1 Logging of Updates.

Let us consider how this overhead can be taken into account by our system model and parameters. (See chapter 4.) The logging of updates can either occur at the same IO device that is used for the database or it could occur at a separate device (e.g., a tape unit).

If a different device is used, we would have to add that device to our model. Notice that in this case, the delay incurred by updates due to logging would be independent of the algorithm being used. The delay would only depend on the number of updates being logged (N/A_r per second) which is the same for all algorithms. Thus for the case of independent logging devices, the logging delay does not have to be considered when we are only comparing the performance of the different algorithms.

If the same IO device is being used for logging, we can assume that a log entry is written just before the new database values are written out. We can model this by assuming a single IO operation for logging and for performing the update and by increasing the value of the parameter I_d. Recall that I_d was the IO service time needed to write one item to the database. Therefore, the performance results of chapter 6 can still be used to compare the performance of the different algorithms. Using the results of chapter 6, we observe that unless the system has a very high IO load and is close to saturation, the centralized locking algorithm still performs better than the other algorithms as the parameter I_d is increased (starting at the typical value used in chapter 6).

5.2 The Two Phase Commit Protocol.

Next, we consider the additional overhead of the two phase commit protocol. This overhead involves sending "intend to perform" messages and waiting for a majority of acknowledgments. The second phase, i.e., sending the "commit" messages is not considered additional overhead because it is equivalent to sending the "perform" or "accept" messages of the original algorithms.

The two phase commit protocol can increase the response time of updates in two ways: First, the protocol increases the CPU load at each node because the additional messages have to be processed. This additional load will cause an increase in the CPU queue wait times for all CPU operations of the algorithm. Secondly, the response time of updates can also be increased by the additional transmissions and by the wait for a majority of acknowledgments.

Let us consider the first source of overhead. The CPU time needed to process an "intend to perform A" message is very small. (There is no IO involved in processing this message.) To process such a message, a node has to check if it has seen update A before and the message has to be stored in memory. Checking for A involves looking at the list of performed updates and at the list of pending messages. (If there are many messages, the check can be performed with the aid

of a hash table.) Thus, the time to process an "intend to perform A" message is comparable to the CPU time needed to set a lock, i.e., C_s. (See chapter 4.)

Processing the acknowledgments of the "intend to perform A" message will take even less time. The node that receives these acknowledgments must simply count the number of acknowledgments. When a majority of acknowledgments is reached, the processing of the update will be completed.

Since the additional CPU load is in both cases minimal, the increase in CPU wait time will be negligible for the parameter values we have considered. Therefore, the increase in response time of the other algorithm steps will also be negligible. (Notice that this last statement is not true if the CPU utilization is very high. In such cases, even a small load increase can cause large increases in the wait times. However, in all the cases we studied in chapter 6, this was not the case. As a matter of fact, in all the cases the CPU utilization was quite low.)

Now let us consider the increase in response time due to the additional waits of the two phase commit protocol. Computing the increase in update response time is not simple. Nevertheless, in the cases that were studied in chapter 6, the increase in update response time can be approximated by $2T$ because the CPU times involved are negligible. (Recall that T is the network transmission time.) This means that the differences in response times between algorithms found in chapter 6 are valid even if a two phase commit protocol is used in the algorithms.

If the CPU time needed for processing the messages of the two phase protocol is not negligible, then the delay will be larger than $2T$. However, in many cases, this increased delay will be smaller in the MCLA centralized locking algorithm than in other algorithms like the distributed voting algorithm. To see why this is true, recall that the CPU load at all nodes except the central node in the MCLA algorithm is lower than the CPU load at all nodes in the other algorithms. This happens because in the other algorithms, all nodes are locking or voting, while in the MCLA algorithm only the central node is locking. Therefore, in the MCLA algorithm, a node will wait a smaller amount of time for a majority of acknowledgments. The acknowledgment from the central node will take longer, but this really does not matter because only a majority of acknowledgments are needed. Of course, in some special cases where the acknowledgment from the central node is needed to get a majority, the delay in the MCLA algorithm could be greater.

5.3 Summary.

In section 5 we studied the performance of the *RCLA-T algorithm* and compared it to the performance of other resilient update algorithms. We observed that a centralized locking algorithm with logging and two phase commit protocol still performs better than the other algorithms with similar modifications unless the IO or the CPU servers are heavily loaded.

This concludes chapter 7. To simplify the presentation, in the next chapters we are again going to assume that no failures occur in the system. Then we will briefly return to the issues of crash recovery in chapter 11 when we consider crash recovery in a partitioned database with multiple controllers.

CHAPTER 8

RESTRICTED TRANSACTIONS

Another one of the assumptions that was made for our performance analysis was that the update algorithms must be able to process arbitrary transactions as defined in chapter 2. In this chapter we will study the implications of this assumption, and we will discuss why we will not eliminate this restriction.

In section 1 we show that the update algorithms can be modified to take advantage of advance knowledge of the transaction types that will run in the system. In section 2 we justify our decision for not studying these specialized algorithms. Finally, in section 3 we make some brief comments on the SDD-1 system [BERN78]. The SDD-1 is a distributed database system which attempts to take advantage of particular transaction types in an automatic way.

1. THE ARBITRARY UPDATE RESTRICTION.

Throughout our studies we have assumed that an update algorithm should be able to handle any transaction which reads an arbitrary set of items in the database and then, based on what it read, the transaction writes into a subset of those items. However, if we restrict the types of acceptable transactions, then we can simplify the algorithms considerably. We present two examples to illustrate this fact.

1.1 Examples.

Assume that all update transactions do not need to read the database in order to compute their update values. That is, every update is simply a set of new item values that must be stored in the database regardless of the old contents. This type of updates arise in any database which only acts as a passive recorder

of outside information. For example, stock brokers have databases which simply contain the current status of the stock market. An update in such a system simply informs the database that a certain stock has a new price or that a certain volume of stock has been traded in a day. These updates, which come in directly from the stock market, are independent of the old contents of the database.

There are many simplified algorithms that can be suggested for a database with the described update restriction [GRAP76]. Using the terminology of Gray et al [GRAY76], in this case we only need degree 1 consistency because there are no read actions in the transactions. All we require is some sequencing mechanism that guarantees that all updates are performed in the same order at all nodes. Simple timestamps [JOHN75] or a central site that issues sequence numbers provide solutions to the problem. Clearly, these simple algorithms perform better than the general algorithms would when used for these restricted updates.

As a second example, consider transactions which are of the form "Add a constant k to item i". In this case all transactions simply read the value of one item, add k to it, and store the new value back in the same item. Such transactions could arise in a bank database where customer accounts are debited or credited as money is withdrawn or deposited. The same situation shows up in an inventory control database where parts enter and leave the warehouse.

For this situation, several simplified algorithms can be designed that take advantage of the update type restriction. Notice that updates are commutative. This means that two updates can be performed in any order at a site and the end result will be the same. Therefore, an algorithm only needs to guarantee that all updates are performed (in any order) at all nodes in the distributed database. Within each site we need a concurrency control that makes each update atomic at a single database. However, our distributed algorithms have always assumed that this exists. No timestamps or locks are needed; an update algorithm simply delivers all updates (in the from "add constant k to item i") and then each node performs the read and write operations. Due to the type of updates, the databases will always be consistent.

1.2 More Than One Transaction Type.

In the above examples we have of course assumed that there is only one type of update. If we have, say, two types of updates, then the simple algorithms might not be enough. For example, we may also have updates of the form "Make item i equal to constant k" in the second example of section 1.1. Since the new

updates are not commutative with the old ones, the simple protocol breaks down and we can easily get inconsistencies.

We will illustrate how these inconsistencies can arise. Suppose that we have two databases and the value of an item j is 10 in both databases. Then suppose that an update to increase item j by 5 arrives at node 1. Let us call this update transaction A. Suppose that update A is only performed at node 1 and, for some reason, the "perform update A" message to node 2 is delayed. Then the value of item j will be 15 at node 1 and 10 at node 2. Next, an update B of the new type arrives and sets item j to 100 at both nodes. When the delayed message for update A arrives at node 2, it increases the value of item j by 5, leaving item j with a value of 105. At this point, the databases have a permanent inconsistency because the value of item j at node 1 is 100.

One solution to the multiple transaction types is to use a general algorithm that works for arbitrary transactions. The algorithms we have been studying (e.g., MCLA, CCA, distributed voting) are of this class. Another solution is to use several specialized algorithms, one for each transaction type possible. The specialized algorithms can then take advantage of the particular transaction type it is assigned to.

1.3 Another Example.

We will now illustrate this idea by considering the following example: Assume that we have a system where a majority of the updates are of the type "Make item i equal to constant k". We will call these the type T1 updates. The rest of the updates are of type T2 and can be arbitrary updates. Furthermore, we assume that it is easy to decide the type of an update simply by examining it. For our system we choose to have two different algorithms or protocols, one for each update type. There are many alternatives available for the two algorithms.

One alternative is based on centralized locking. We choose A2, the algorithm for the type T2 updates, to be the MCLA-infinity algorithm (see chapter 3) with one slight modification: The table of locked items (i.e., locked(i) in Appendix 1) also contains the sequence number of the update that has locked the particular item. That is, given a locked item number, the central node (where locks are granted for all T2 updates) can tell what update holds that lock. The effort needed to keep this extra information should be minimal because the extra information can be stored in the same (hash) table used to keep track of what items are locked.

Algorithm A1, the algorithm for the type T1 updates, is simpler than A2 because type T1 updates do not need to hold locks while they are being performed. Only the correct sequencing information is needed before a T1 update can be performed and such information is available at the central node. We now present an outline of the A1 algorithm:

Algorithm A1.

STEP 1. Update B arrives at node x: "Make item i equal to constant k". Node x identifies this update as type T1 and marks it as such. Node x forwards the update to the central node.

STEP 2. Upon receipt of the "Make item i equal to constant k" update, the central node checks if item i is locked. If it is, go to step 4, else go to step 3.

STEP 3. Item i is free. Therefore update B need not wait for any of the currently executing updates. So B is assigned the next available sequence number, the "hole list" (see below) is appended and the "perform update" messages are sent to all nodes. Go to step 5. (Notice that the "perform update" messages can be sent directly by the central node without having to first send a "grant" message to the update originating node. This can be done this way because there is no base set to read and no computations to perform before the new value for item i (i.e., k) is obtained.) (The hole list contains the sequence numbers of the currently executing updates. This list represents the updates that B does not have to wait for before being performed at a node. See chapter 3 for details.)

STEP 4. Item i is locked by update C. Say C's sequence number is r. Therefore, B only has to wait for the update with sequence number r. So B is assigned the next sequence number, and a hole list with all sequence numbers except r is appended to B. (This means that B can be performed at any node that has performed update C.) The "perform update" messages are sent out to all nodes.

STEP 5. Upon receipt of the "perform update" messages, all nodes (including the central node) perform the update in the usual way (i.e., checking the hole list and the sequence number of B). (End of algorithm A1.)

Notice that a T1 update does not obtain any locks and is never deferred at the central node. The central node only attaches the necessary sequencing information to each T1 update so that the nodes know when the update can be performed. Also notice that the sequence number of a T1 update is not placed in the hole list (because updates in the hole list are the ones that currently hold locks and T1 updates never hold locks). This means that any other updates that follow B, a T1 update, will have to wait for B before being performed. This represents no problem at all since B cannot be delayed reading a base set and

computing as the other general updates can.

The A1 algorithm we have described is much more efficient than the general A2 (i.e., MCLA-infinity) algorithm because several of the steps of the general algorithm have been eliminated. Furthermore, because the T1 updates are able to execute without holding locks, the rest of the updates will also benefit from the A1 algorithm. Of course, the magnitude of the savings will strongly depend on the mix of T1 and T2 updates in the system.

2. WHY WE ONLY STUDY ARBITRARY TRANSACTION ALGORITHMS.

Another alternative for the system with T1 and T2 updates is to design two algorithms based on the use of timestamps and the distributed voting algorithm. In this case the A2 algorithm for the T2 updates would be the general distributed voting algorithm. (See chapter 3 and [THOM76].) In the A1 algorithm for T1 updates, we can eliminate the voting phase of the A2 algorithm because T1 updates do not read any data. As before, all we need is the correct sequencing information, which in this case is provided by the timestamp mechanism. Hence, in the A1 algorithm, all we need to do is to obtain the current timestamp at a node and attach it to the "perform update" messages which are immediately sent out to all nodes.

The A1 algorithm that uses timestamps seems to be more efficient and simpler than its counterpart that uses centralized control. (After we study read only transaction in chapter 9, we may disagree with the above statement. For the time being we are only considering updates, so the above statement is valid.) On the other hand, as we have seen in chapter 6, the A2 algorithm is more efficient when centralized control is used instead of timestamps. We therefore reach the conclusion that the particular mix of T1 and T2 updates will determine whether the central control or the timestamp strategy is superior.

Recall that the above discussion refers only to a particular example where T1 updates are of the form "Make item i equal to constant k" and T2 updates are any other updates. For a different set of transaction types we may find that a particular strategy is always superior, or we may again reach the conclusion that the transaction mix dictates the best strategy.

For a given set of transaction types and mixes, we are able to compare the performance of several alternatives by using the simulation and analysis methods outlined in this thesis. But it is very hard to reach any general conclusions as

to which is the best strategy or algorithm for all transaction types and mixes. We will therefore limit ourselves to study only the general algorithms that can handle arbitrary transactions. The designers of a general purpose distributed database system will probably have to use a general update algorithm and will hence be able to use the results we have obtained so far. The designers of a specialized database system with a known set of transaction types will have to study their particular system. We hope that they can use some of the tools that we have used in this thesis to reach their own conclusions.

3. THE SDD-1 SYSTEM.

Before we conclude this chapter, we mention a distributed database system currently being designed which attempts to take advantage of particular transaction types in an automatic way. In the SDD-1 system [BERN78], the database administrator selects a set of predefined transaction types which he hopes will cover most of the updates that will be submitted by the users. The system has 4 update algorithms or protocols. The algorithms have different degrees of complexity and of generality. At system creation time, the chosen transaction types are analyzed in an automatic way and a protocol is chosen for each type. The protocol chosen for a transaction type is the most efficient one of the four that can correctly handle the transaction. When the system is in operation, updates are analyzed to decide what their type is and the corresponding protocol is used to execute the update. If the update does not fall within one of the predefined types, the least efficient but most general protocol must be used.

An interesting research project would be to study the most general SDD-1 protocol and compare it to the other arbitrary update algorithms we have presented in chapter 3. We have not done this in this thesis. However, at first sight, the SDD-1 most general protocol seems to be more complex and less efficient than the other algorithms. Furthermore, the advantage of the SDD-1 system lies in its ability to handle 4 different protocols automatically, and only by studying the system in a particular application with a gives set of transaction types and mixes, would we be able to make a fair comparison to some other system.

In summary, in a specific application where many of the transaction types can be analyzed at system creation time, the SDD-1 system could operate efficiently. However, by adopting any of the other update algorithms to the same application,

we could probably get performance improvements too. What strategy performs best depends on the particular application and transaction mix.

CHAPTER 9

READ-ONLY TRANSACTIONS

In this chapter we will study the elimination of the update-only restriction that was made in the performance analysis. In section 1 we define read-only transactions (queries) and we classify them into three groups: free, consistent and current queries. In the following three sections we present consistent and current query algorithms for the cases where the MCLA-h, the DVA, and the Ellis type algorithms are used for the update transactions. In these sections we also study the consistency provided by these query algorithms by applying the notions of consistency developed by Eswaran et al [ESWA76]. In passing, we also prove the consistency of the MCLA-h algorithm (in section 2.3) and the end (or convergence) consistency of the DVA algorithm (in section 3.2). In section 5 we discuss the performance of the query algorithms, while in section 6 we state some conclusions for this chapter.

1. READ-ONLY TRANSACTIONS.

A read-only transaction or query reads a set of items from the database and presents the values obtained to the user. The transaction in no way modifies the database. This means that a user cannot make an update to the database based on the data obtained from a query. If the user wishes to submit such an update, the update must first read the data again to check if the data has not changed.

A read-only transaction or query can be considered as an update transaction where the write set is empty. Therefore, the update algorithms can also be used to read data. However, many simplifications are possible when handling queries. That is, since queries are simply a restricted type of update, we can take advantage of this to improve efficiency.

We are now making an exception to the statement of section 2 of chapter 8 that we will only study general algorithms for handling arbitrary transactions.

The reason why we will study these restricted updates called queries is that queries are very common in almost any conceivable database system, regardless of the types of updates that are performed. As we may suspect from the discussion of chapter 8, we will be unable to reach sweeping conclusions as to which algorithm performs best because the efficiency will depend on the types and percentages of queries submitted. However, we will be able to reach some limited but interesting performance conclusions which will be very useful in any system that must process queries.

1.1 Types of Queries.

There are two requirements that we can make on queries. The first is to require that the query should give the user a consistent view of the data [ESWA76]. In other words, all consistency constraints or assertions that can be fully evaluated with the data read should be true. (Constraints that cannot be fully evaluated with the data read by the query are irrelevant here.) A second independent requirement that can be made on a query is that the data read from the database is the latest or most current [GRAY79]. In other words, we can require that a query submitted to the system at time t should reflect any updates that were performed anywhere in the system before and up to time t. We than say that the data produced by the query is current as of time t. (In our discussion of currency we use time in an intuitive fashion. The ideas could be formalized using concepts in [LAMP78].)

We can define four types of queries according to the consistency and currency requirements they make. A "free" query makes no requirements at all. A "consistent" query requires consistent data, while a "current" query requires current data. Finally, a "current and consistent" query makes both requirements.

1.2 An Example.

We will illustrate some of these concepts through a simple example. Suppose that we have a subset of the database, d, and a set of consistency constraints, c, on this subset. Assume that at time t_0 all copies of d have the same value, there are no pending updates that involve d, and d is consistent (i.e., $c(d)$ is true).

Next, three conflicting updates u_1, u_2, and u_3 that involve d are performed. Update u_1 is first performed at a node at time t_1, update u_2 is first perfor[illegible]

at a node at time t_2, and update u_3 is first performed at time t_3, where $t_0 < t_1 < t_2 < t_3$. Since the updates conflict, we assume that the node that performs u_2 must have seen u_1 first. In other words, u_2 was computed based on $u_1(d)$, where $u_1(d)$ is the resulting subset of the database after u_1 has been performed. Similarly, u_3 was computed based on $u_2(u_1(d))$ to preserve consistency.

A consistent query should return either d, $u_1(d)$, $u_2(u_1(d))$ or $u_3(u_2(u_1(d)))$ because only these values are consistent. In some update algorithms (e.g., the distributed voting algorithm), a node can perform the updates u_1, u_2, u_3 in a different order. In such a case, that node cannot answer a consistent query until all updates are performed. Hence, if the local database is $u_3(u_1(d))$, the query must wait until u_2 is performed. Then $u_2(u_3(u_1(d)))$, which is here equal to $u_3(u_2(u_1(d)))$, can be returned as a consistent answer to the query. Notice that in other update algorithms (e.g., the centralized locking algorithm) $u_2(u_3(u_1(d)))$ is not equal to $u_3(u_2(u_1(d)))$ and the updates can only be performed in the proper sequence. In such algorithms, any database will either contain d, $u_1(d)$, $u_2(u_1(d))$ or $u_3(u_2(u_1(d)))$, and consistent queries can simply read any database.

The values in d are current up to time t_1. After time t_1, d is still consistent but is out of date. Thus, a current query Q submitted at time t, $t_1 < t < t_2$, should reflect update u_1. Current query Q could read $u_1(d)$. This data also happens to be consistent. However, current query Q could also read $u_3(u_1(d))$ which is current as of time t but is inconsistent. A consistent and current query submitted at time t, $t_1 < t < t_2$, can only read $u_1(d)$, $u_2(u_1(d))$ or $u_3(u_2(u_1(d)))$ and not d.

A free query can read data that is inconsistent and out of date. For example, a free query submitted at time t, $t > t_3$, could read d, $u_1(d)$, $u_3(u_2(u_1(d)))$, $u_2(d)$, $u_2(u_3(u_1(d)))$, or any such combination.

1.3 Why We Need Different Query Types.

It might seem that free queries are not very useful because they can produce data that is inconsistent and out of date. On the other hand, free queries are extremely simple and efficient to process since all they must do is read the data at a node without bothering with anything else (not even with local concurrency control). In many applications, users may be willing to sacrifice consistency and currency for efficiency. Furthermore, in well designed systems, free queries should produce results based on data that is not too old. For example, a warehouse manager might want a rough idea of where the inventory stands. The manager

does not really care if the data obtained is 15 minutes old. The manager might not mind that the total number of parts reported does not exactly match the sum of the itemized entries in the report obtained. As another example, consider a query that computes an average salary for a large set of employees. The result might not be accurate if some of the salaries are being updated during the long period that the averaging query is running. But the user might decide that such occasional conflicts will not alter the average significantly. Furthermore, not running the averaging query as a free query will produce long delays in other transactions that access the salary data.

Another case where free queries are valuable is in one item queries. A query that only reads one item will always give a consistent view of the data. To see why this is true, consider a given item j and its value v at node x. The value v must have been written by some update U. If there are any consistency constraints that deal exclusively with item j, then update U must have produced a consistent value v because a single update never violates the consistency constraints. Therefore, the free query mechanism can be used for one item queries and the result will always be consistent. In many systems, one item queries are very common and considerable effort can be saved if we use an efficient method like the free query mechanism for performing these queries.

Clearly not all queries in a system can be free queries. In some queries, a consistent view of the data is required. The checking account monthly statement that is sent to a bank customer must be consistent. (For example, the sum of cashed checks should equal the total debits entry.) Also, a read-only transaction to look up the location of a device in a distributed system directory given the name of the device should not encounter duplicated names in the directory. These are only two of the many cases where consistent queries are needed.

In many systems, the currency of the data is not a critical factor. For example if a query to produce a monthly statement for an (interest free) checking account misses some of the latest withdrawals or deposits, they will simply be reported in the next statement. However, in other situations current queries are a must. Consider a general who has to decide whether to fire or not a missile at an incoming warship. In this situation, the general needs the latest information on the ship's position and speed in order to make the best decision.

1.4 The Query Algorithms.

We will now describe how queries can be processed in a distributed database

system. The algorithm used for queries depends on the update algorithm being utilized, so we will divide the following discussion into three sections, one section for each of the main update algorithms we have studied. We still assume that databases are completely duplicated at each node, and that transactions (queries and updates) fully specify at their inception the items that they reference. We are again going to assume that no failures occur in the system. (In chapter 11 we make some comments as to how failures affect queries.) We will not discuss free queries in the next sections because the algorithm for them is independent of the update algorithm. To process a free query, a node simply reads the referenced values from the local database. Not even local concurrency control is needed for free queries.

2. QUERIES IN THE CENTRALIZED LOCKING ENVIRONMENT.

2.1 Consistent Queries.

When the centralized locking algorithm (MCLA-h) is used for updates, all updates are assigned a sequence number after they obtain their locks. At each node, conflicting updates are always performed in ascending sequence number order. In other words, the base set data read by an update will reflect all updates with lower sequence numbers; and the data that is written by an update will be seen by all updates that need this data and that have higher sequence number. This means that if all updates are performed at nodes as atomic operations, then the database will always be consistent in between these operations. Therefore, if a query reads its data between the update operations, it will get a consistent view of the database. This is equivalent to saying that a consistent query only needs local concurrency control in order to be executed. Hence, when the MCLA-h algorithm is used for updates, queries can be executed very efficiently without the need for communicating with other nodes.

The above discussion has been very informal. We will now show that the above statements are true in a more formal way. In passing, we will also show that the MCLA-h algorithm for updates provides all updates with a consistent view of the database. The following formalization is simply an application of the *consistency notions* defined in [ESWA76] with a few minor modifications. The

reader is strongly urged to read [ESWA76] since it provides the basis for the following material.

2.2 The Notions of Consistency in a Distributed Database.

The notions of consistency in a distributed database system are identical to the ones in a centralized system. In both cases we have a set of items, some consistency constraints, and some processes that are executing actions on the items. The fact that in the distributed system the items are stored in different nodes and the processes may run on different computers makes no difference. Therefore, we will simply view the distributed databases as a single large database where each item $d[i, x]$ in this database corresponds to item i at node x. (See chapter 2.) The *consistency constraints* for the distributed database are simply written in this notation. For example, assertion $a = b + c$ at node 3 is now written as $d[a, 3] = d[b, 3] + d[c, 3]$. Since the databases are completely duplicated, then the assertion $a = b + c$ will become a set of assertions $d[a, y] = d[b, y] + d[c, y]$ for $1 \leq y \leq N$ (N is the number of nodes). In addition to the constraints at each node, we would like a duplicated item i to have the same value at all nodes. These are simply the *implicit consistency constraints* $d[i, x] = d[i, y]$ for all nodes x, y. (See chapter 2.)

The concepts of *actions* and *schedules* are not changed for distributed databases. An action is represented by (T, a, e), where "T" is the name of the transaction, "a" is either read (r) or write (w), and "e" is an item. We assume that all actions are atomic and that they can be written in a linear sequence which is called the schedule of the actions. (For a formalization of the ideas of actions and schedules in a distributed database see [BERN78].)

Our model of an update transaction is a series of read actions at a single node followed by a series of identical write actions at all nodes. (See chapter 2.) That is, let T be an update transaction that originates at node x. Transaction T first reads items $d[B_1, x], d[B_2, x], \ldots d[B_b, x]$ at originating node x, where $B_1, B_2, \ldots B_b$ are the indices of the items in the base set of T. After reading and performing computations, T writes the items $d[B_1, y], d[B_2, y], \ldots d[B_c, y]$ for all nodes y such that $1 \leq y \leq N$. Notice that all written items are in the base set, that is, $c \leq b$.

Thus, we represent update T, which originated at node x, by the sequence

$$T = ((T, a_1, e_1), (T, a_2, e_2), \ldots (T, a_{b+Nc}, e_{b+Nc}))$$

where (1) action a_i = read for $1 \leq i \leq b$; (2) item $e_i = d[B_i, x]$ for $1 \leq i \leq b$;

(3) action a_i = write for $b < i \leq b + Nc$; and (4) items e_i for $b < i \leq b + Nc$ are the items $d[B_1, y], \ldots d[B_c, y]$ for $1 \leq y \leq N$ in some order. Notice that the particular order of the write actions is not important as long as they all follow the read actions. We assume that if T is run on a consistent database and without interference from other updates, it will produce another consistent database. For example, the actions $(T, w, d[k, 1]), (T, w, d[k, 2]), \ldots (T, w, d[k, N])$ (fixed k) must write the same value in order not to violate the implicit consistency constraint $d[k, 1] = d[k, 2] = \ldots d[k, N]$.

A schedule is said to be *serial* if the transactions in it are executed one at a time. A schedule is said to be *consistent* if it is equivalent to a serial schedule. The requirement that an algorithm produce a consistent schedule is stronger than the requirements we had defined for an algorithm in chapter 2. That is, if an algorithm produces a consistent schedule then (1) all transactions will get a consistent view of the data, and (2) the implicit consistency constraints will not be violated. As we will see shortly, the inverse of this statement is not necessarily true.

To see that all transactions in a consistent schedule read consistent data, simply note that the consistent schedule is equivalent to a serial schedule and that all transactions in a serial schedule see consistent data because the transactions are executed one at a time. (See chapter 2.) Similarly, the implicit consistency constraints are not violated in a consistent schedule because it is equivalent to a serial schedule where the implicit consistency constraints are not violated.

2.3 Consistency of the MCLA-h Algorithm for Updates.

The concepts presented up to this point apply to any of the update algorithms we have studied. Now we will concentrate on the MCLA-h algorithm. Consider the schedule S produced by running a set of update transactions (as defined in section 2.2) under the control of the MCLA-h algorithm. In this case, we can prove the stronger condition that S is consistent. To show that S is consistent (i.e., equivalent to a serial schedule) we must show that the binary relation "$\prec$" on the set of transactions, which is defined below, is acyclic [ESWA76].

The relation "$\prec$" produced by schedule S is defined as follows. $T_p \prec T_q$ (p different from q) if and only if for some $i < j$,

$$S = (\ldots(T_p, a_i, e), \ldots (T_q, a_j, e), \ldots)$$

where (1) either a_i or a_j are write actions and (2) there is no k such that $i < k < j$

and $e_k = e$ and a_k = write.

To show that "$\prec$" is acyclic, we order the transactions using the sequence numbers that were assigned to each update at the central node (in the MCLA-h algorithm). Let T_p be the update transaction with sequence number p. We will now show that $T_p \prec T_q$ implies that p is less than q and therefore that "$\prec$" is acyclic and S consistent.

THEOREM 1. Let T_p and T_q be two different update transactions, with sequence numbers p and q respectively, that were executed using the MCLA-h algorithm. Then $T_p \prec T_q$ implies that $p < q$.

PROOF OF THEOREM 1. Let S be the schedule produced by the MCLA-h algorithm. $T_p \prec T_q$ implies that for some $i < j$,

$$S = (\ldots(T_p, a_i, e), \ldots (T_q, a_j, e), \ldots)$$

where (1) either a_i or a_j is a write action and (2) there is no k such that $i < k < j$ and $e_k = e$ and a_k = write. Let e be item $d[m, x]$ at node x. Since both T_p and T_q reference item $d[m, x]$, these updates conflict.

We show that $p < q$ by contradiction. Assume that $p > q$. This means that T_p obtained its locks at the central node after T_q did. Since T_p and T_q conflict, when T_p obtained its locks and its sequence number p, T_q must have released its locks at the central node (else T_p could not get locks for item $d[m, x]$). Therefore, T_q is not in T_p's hole list. This in turn implies that node x cannot perform any T_p action involving $d[m, x]$ until T_q's write action involving $d[m, x]$ has completed at x. (This is guaranteed by the local concurrency control at node x.) That is, any action $(T_p, a_i, d[m, x])$ must follow an action $(T_q, w, d[m, x])$ at node x. Since all of T_q's reads precede its writes in S, then $(T_p, a_i, d[m, x])$ must also follow any action $(T_q, r, d[m, x])$. Therefore, the above schedule S is impossible for any actions a_i, a_j. This is a contradiction, so p must be less than q. (End of proof.)

THEOREM 2. Any schedule S for update transactions produced by the MCLA-h algorithm is consistent.

PROOF OF THEOREM 2. Suppose that the relation "$\prec$" defined by S has a cycle $T_p \prec T_q \prec \ldots \prec T_r \prec T_p$. By theorem 1, this implies that $p < p$, which is impossible. Therefore, "$\prec$" cannot have any cycles. This implies that S is equivalent to a serial schedule [ESWA76] and hence consistent. (And thus, the MCLA-h provides transactions with a consistent view of the database and does not violate the implicit consistency constraints.) (End of proof.)

2.4 Consistency of Queries.

Up to now, we have only dealt with update transactions in the MCLA-h algorithm. We will now show that any query that is performed at a node x with local concurrency control will also get a consistent view of the database. A query or read-only transaction is of the form

$$Q = ((Q, a_1, e_1), (Q, a_2, c_2), \ldots (Q, a_b, e_b))$$

where (1) all a_i for $1 \leq i \leq b$ are reads, and (2) $e_i = d[B_i, x]$ for $1 \leq i \leq b$.

The concurrency control at a node x should process queries as follows. Suppose that a query Q arrives at node x at time t. At that instant, a set P of update transactions have initiated their writes at node x. That is, if $T_i \in P$, then some action $(T_i, w, d[m, x])$ occurs before time t (for some valid item index m). If $T_j \notin P$, then no action $(T_j, w, d[m, x])$ occurs before time t at node x. Query Q should be processed at node x in such a way that it sees the effects of all updates in P and sees no effect of any update not in P.

The following observation may seem surprising at first, but we will shortly show that it is no cause for concern.

OBSERVATION. Not all schedules S for query and update transactions produced by the MCLA-h algorithm and the local concurrency control described above are consistent.

PROOF OF OBSERVATION. We show that this observation is true by looking at a particular schedule S, produced by the MCLA-h algorithm and the local concurrency control, which is not equivalent to any serial schedule. Consider two update transactions T_p and T_q in a system with two nodes x and y. Update T_p simply updates item 1 without reading any data, while update T_q similarly updates item 2. Thus we can write

$$T_p = ((T_p, w, d[1, x]), (T_p, w, d[1, y]))$$

and

$$T_q = ((T_q, w, d[2, y]), (T_q, w, d[2, x])).$$

Notice that these two updates do not conflict because they reference different items. Therefore, the actions of T_p and T_q can occur in any order in schedule S.

Next, consider two queries which read items 1 and 2. One query, Q_1, is performed at node x, while query Q_2 is executed at node y. We can represent

Q_1 and Q_2 as

$$Q_1 = ((Q_1, r, d[1, x]), (Q_1, r, d[2, x]))$$

and

$$Q_2 = ((Q_2, r, d[1, y]), (Q_2, r, d[2, y])).$$

The following schedule S can be produced by our algorithm:

$$\begin{aligned} S = (\ & (T_p, w, d[1, x]), \quad (T_q, w, d[2, y]), \quad (Q_1, r, d[1, x]), \\ & (Q_1, r, d[2, x]), \quad (Q_2, r, d[1, y]), \quad (Q_2, r, d[2, y]), \\ & (T_p, w, d[1, y]), \quad (T_q, w, d[2, x])). \end{aligned}$$

Schedule S is legal because the MCLA-h algorithm permits T_p and T_q actions in S to come in any order and because both queries either see the complete effect of updates or they do not see any effects. According to the definition of the relation "$\prec$", we see that $T_p \prec Q_1$ in S because transaction T_p "hands" an item (i.e., $d[1, x]$) to transaction Q_1. Similarly, $Q_1 \prec T_q$, $T_q \prec Q_2$, and $Q_2 \prec T_p$. This implies that the relation "$\prec$" is cyclic and hence schedule S is not equivalent to any serial schedule. (*End of proof.*)

We can interpret the above observation as follows. In S, Q_1 sees T_p but does not see T_q, while Q_2 sees T_q but not T_p. There is no way that these four transactions can be executed one at a time (i.e., serially) and produce this same effect. In any serial schedule, if Q_1 sees T_p, and Q_2 follows Q_1 in the schedule, then Q_2 must also see T_p. This is not the case in S.

Although this result may seem surprising at first, it does make sense. Notice that in S we are executing some transactions (e.g., Q_1 and Q_2) without any global concurrency control. If we wanted S to be strictly consistent, then all transactions, including Q_1 and Q_2, should be performed following the MCLA-h algorithm.

Fortunately for us, the fact that S is not consistent does not mean that Q_1 and Q_2 do not see a consistent view of the data. If we eliminate Q_2 from S, we observe that the resulting schedule is consistent and equivalent to the serial schedule $\{T_p, Q_1, T_q\}$ (i.e., T_p executed first, then Q_1, and then T_q). Thus, Q_1 sees a consistent database at node x. Similarly, if we delete Q_1 actions from S, we find that Q_2 sees the consistent data at node y produced by serial schedule $\{T_q, Q_2, T_p\}$. In other words, both queries see a database produced by some serial execution of the updates, but these serial executions of updates may be different for each query. Since queries in no way modify the database, then the fact that the queries perceive different serial executions of the updates is of no consequence. (Notice that if some user were able to look at the results produced

by Q_1 and Q_2 at the same time, the user may be confused. Here we assume that users only look at the results of their own queries.)

To conclude this section, we will prove for the general case that queries see a consistent view of the database, even if the overall global schedule is not consistent.

THEOREM 3. Any schedule S, produced by running a single query Q at node x with local concurrency control together with a set of update transactions under the control of the MCLA-h algorithm, is consistent.

PROOF OF THEOREM 3. We show this by contradiction. Assume that there is a schedule S that violates the statement of theorem 3. Then there must be a cycle in the "$\prec$" relation defined by S. This cycle must contain Q because (as was shown in theorem 2) any schedule of updates is cycle free. Say that this cycle is $Q \prec T_1 \prec T_2 \prec \ldots \prec T_n \prec Q$ (where $T_1, T_2, \ldots T_n$ are update transactions). Since $Q \prec T_1$, there is a write action of T_1 that follows a read of Q. And because of the concurrency control at node x, no write action of T_1 can precede the first read of Q in S at node x. In particular, the first write action of T_1 must follow the first read of Q. By a similar argument, we can show that $T_n \prec Q$ implies that the first write action of T_n at node x must preceed the first read of Q at x. Combining these two observations, we see that the first write action of T_n at x must occur before the first write action of T_1 at node x. This can only occur if T_n has a lower sequence number than T_1. However, theorem 1 and the fact that $T_1 \prec T_2 \prec \ldots \prec T_n$, implies that T_1 must have a lower sequence number than T_n. This is a contradiction. (End of proof.)

2.5 Current Queries in the Centralized Locking Environment.

A current query Q submitted to the system at time t must reflect any updates that were performed anywhere in the system before and up to time t. When the centralized locking algorithm is used for updates, the currency restriction can be expressed in terms of the sequence numbers (assigned by the central node) as follows: Let s be the last sequence number assigned by the central node the instant that Q is submitted. Then Q should reflect all updates with sequence numbers up to s.

There are many alternatives for dealing with current queries. Here we will simply outline a few of them. One straightforward way to process a current query Q that originates at node x is to use the update algorithm for it. Since Q

will receive a sequence number s_1 greater than s (the current sequence number at the central node when Q was submitted), then all conflicting updates with lower sequence number than s will have been performed at x when Q is ready to be "performed".

This algorithm can be made more efficient by not having Q hold locks at the central node and by not having Q obtain a sequence number. Furthermore, Q does not even need to wait at the central node if it finds a locked item. Instead, Q can note the current sequence number, s, and obtain a copy of the current hole list as soon as it arrives at the central node. Then, as Q checks if the items it references are locked, it simply deletes from its copy of the hole list the sequence number of any update that Q discovers holding a lock for an item Q needs. After this processing at the central node, Q returns to its originating node x (or to any other node) where it waits until all updates with sequence number less than s and not in its hole list copy are performed at the node. After this wait, Q can be executed. By following this strategy, Q will not wait for updates with lower sequence numbers that do not modify items referenced by Q.

A further improvement in the response time of query Q may be obtained by having the central node send copies of Q to various nodes after Q has obtained its sequencing information. This way, the first node to have all the necessary data for Q can execute Q and send the results to the user (who may be at another node). However, the duplication of effort produced by this strategy may slow down other transactions.

Finally, if response time is not critical, a very simple algorithm can be devised. In this algorithm, a current query Q, originating at node x, simply requests the latest sequence number issued by the central node. Then Q waits until all updates with sequence number up to the latest one have been performed at node x.

2.6 Current and Consistent Queries in the Centralized Locking Environment.

A current query which also needs consistent data simply follows both protocols for current reads and for consistent reads. In other words, when a query Q is finally executed according to any of the current algorithms mentioned above, it should be subjected to the local concurrency control in order to obtain consistent data as well.

2.7 Summary.

In this section we have shown how current and consistent queries can be executed when the MCLA-h algorithm is used to coordinate updates. We observed that the sequence numbers issued by the central node are a very powerful concept which permit us to perform consistent queries with simple local control and to perform current queries in an efficient and intuitive fashion.

3. QUERIES IN THE DISTRIBUTED VOTING ENVIRONMENT.

3.1 Consistent Queries.

In this section we will study how consistent queries can be executed when the distributed voting algorithm is used to handle update transactions. We will find that local concurrency control is not enough for consistent queries. The fact that a more complex protocol is required for consistent queries in this environment is a serious drawback of the distributed voting algorithm.

To show that local concurrency control is not adequate for consistent queries, we consider the following example. The system consists of three nodes: x, y, and z; and the database at each of these nodes contains three items: a, b, and c. (We can think of these items as being the "deposits", "withdrawals" and "balance" items used in the examples of chapter 2. We use the shorter names a, b, and c here in order to simplify some of the expressions we will write.) Suppose that we have defined the consistency constraint "$a - b = c$" on this database. (That is, "deposits" — "withdrawals" = "balance".) Initially, the value of item a is 100 at all nodes; the value of item b is 40 at all nodes; and the value of item c is 60 at all nodes. Thus, the consistency constraint is satisfied. Now suppose that we have two updates. Update A is "increase a by 10 and c by 10" (i.e., "deposit 10 dollars") and update B is "increase b by 5 and decrease c by 5" (i.e., "withdraw 5 dollars"). Update A arrives first and receives OK votes at nodes x and y and is accepted by node y. Node y sends out "perform update A" messages to all nodes. (These messages indicate that the new value for item a is 110 and the new value for item c is 70.) Update A is then performed with timestamp t_1 at nodes x and y but for some reason the "perform update A" message to node z is delayed.

Then update B arrives and receives OK votes at nodes x and y. "Perform update B" messages indicating that the new value of item b is 45 and the new value of item c is 65 are sent out to all nodes. Update B is performed at nodes x and y with timestamp t_2, where $t_2 > t_1$.

The databases at nodes x and y first contain $a = 100, b = 40, c = 60$; then $a = 110, b = 40, c = 70$ (after A); and finally $a = 110, b = 45, c = 65$ (after A and B). All these databases are consistent (i.e., $a - b = c$). However, at node z, update B is performed even though the "perform update A" message has not arrived. Node z has no way of knowing that this message is missing since it did not vote for update A. This leaves the database at node z with $a = 100, b = 45, c = 65$, which is an inconsistent state. Any queries performed at z after B has been performed and before the "perform update A" message arrives, will get an inconsistent view of the database.

Of course, when the "perform update A" message arrives at node z, the value of a will be changed to 110, leaving the database consistent once again. Notice that the value of c (equal to 65) is not affected by the "perform update A" message because the timestamp of item c in the database is t_2, while the timestamp of update A is t_1, which is less than t_2.

3.2 Consistency of the Distributed Voting Algorithm for Updates.

In order to understand more fully why it is that the distributed voting algorithm may temporarily leave inconsistent data at some nodes, we will briefly discuss the type of consistency provided by the distributed voting algorithm. It turns out that not even the distributed update algorithm with only update transactions provides consistency in the sense that the centralized locking algorithm does.

OBSERVATION. Not all schedules S for update transactions produced by the distributed voting algorithm are consistent.

PROOF OF OBSERVATION. Using the notions of transactions and consistency defined in section 2.2, we can write the update transactions of the previous example (section 3.1) as:

$$\begin{aligned} A = (\ & (A, r, d[a, x]), \quad (A, r, d[c, x]), \quad (A, w, d[a, y]), \\ & (A, w, d[c, y]), \quad (A, w, d[a, x]), \quad (A, w, d[c, x]), \\ & (A, w, d[a, z]), \quad (A, w, d[c, z])) \end{aligned}$$

and

$$B = (\ (B, r, d[b, x]),\ (B, r, d[c, x]),\ (B, w, d[b, y]),\ (B, w, d[c, y]),\ (B, w, d[b, x]),\ (B, w, d[c, x]),\ (B, w, d[b, z]),\ (B, w, d[c, z])).$$

The schedule obtained by executing A and B as described in the example is

$$S = (\ (A, r, d[a, x]),\ (A, r, d[c, x]),\ (A, w, d[a, y]),\ (A, w, d[c, y]),\ (A, w, d[a, x]),\ (A, w, d[c, x]),\ (B, r, d[b, x]),\ (B, r, d[c, x]),\ (B, w, d[b, y]),\ (B, w, d[c, y]),\ (B, w, d[b, x]),\ (B, w, d[c, x]),\ (B, w, d[b, z]),\ (B, w, d[c, z]),\ (A, w, d[a, z]),\ (A, w, d[c, z])).$$

By examining S, we observe that A $\prec$ B (because action $(A, w, d[c, x])$ preceeds action $(B, r, d[c, x])$) and B $\prec$ A (because action $(B, w, d[c, z])$ preceeds action $(A, w, d[c, z])$). Since the relation "$\prec$" is cyclic, S is inconsistent. (End proof of observation.)

The B $\prec$ A relation in S is produced by the last action $(A, w, d[c, z])$. However, this is not a "normal" write action because it is never actually performed. Recall that update A had obtained timestamp t_1 while item c at node z had timestamp t_2 greater than t_1. Hence, the timestamp mechanism at node z simply ignores action $(A, w, d[c, z])$ and item c keeps its old value of 65.

Thus, in a sense, the relation B $\prec$ A does not represent a real dependency of transactions A and B. In other words, B $\prec$ A means that an action of B must come before an action of A in schedule S, but in this case this is not strictly true. The action of A (i.e., $(A, w, d[c, z])$) could be moved ahead of the B action (i.e., $(B, w, d[c, z])$) and the end effect of the actions would be unchanged. That is, let schedule S′ be the schedule obtained from S by moving action $(A, w, d[c, z])$ as follows:

$$S' = (\ (A, r, d[a, x]),\ (A, r, d[c, x]),\ (A, w, d[a, y]),\ (A, w, d[c, y]),\ (A, w, d[a, x]),\ (A, w, d[c, x]),\ (B, r, d[b, x]),\ (B, r, d[c, x]),\ (B, w, d[b, y]),\ (B, w, d[c, y]),\ (B, w, d[b, x]),\ (B, w, d[c, x]),\ (B, w, d[b, z]),\ (A, w, d[c, z]),\ (B, w, d[c, z]),\ (A, w, d[a, z])).$$

If we start with some initial state of the database and perform schedule S′ we obtain the same final state as if we had started with the same initial state and then performed schedule S. This is true because, in either case, action $(A, w, d[c, z])$ has

no effect on the final state. In S′, the value written by this action is overwritten by the following action (without any other actions having read the value) while in S, the action is not actually performed. In this special sense, schedules S and S′ are equivalent.

We will use the term "*end equivalent*" to describe two schedules whose final effect on the system is the same. A schedule which is end equivalent to a serial schedule is end consistent. (The terms *mutual consistency* [THOM76] and *convergence consistency* [GRAY79] have been used for what we call end consistency here.)

Notice that schedule S′ is a consistent schedule because the "$\prec$" relation for S′ has no cycles (i.e., we have eliminated the B $\prec$ A relation). Therefore the original schedule S is end equivalent to some serial schedule. We will now show that in general, any schedule S produced by the distributed voting algorithm, with update transactions only, is end equivalent to some serial schedule. But before we do so, it is useful to recall how timestamps are assigned to update transactions in the distributed voting algorithm (see chapter 3):

The timestamp of transaction T, ts(T), is assigned to T when T is accepted and must be (1) larger than the timestamps of the items read by T at T's originating node, and (2) larger than the clock times at all the nodes visited in the voting process. Also notice that when T requests OK votes at a node, that node reads all the timestamps in the base set of T. If any of these timestamps is larger than the corresponding timestamp read by T at its originating node, then T is rejected. Therefore, if T is accepted, ts(T) will also be larger than all the timestamps encountered in the voting process for the items in the base set of T.

THEOREM 4. Any schedule S of update transactions produced by the distributed voting algorithm is end equivalent to some serial schedule.

PROOF OF THEOREM 4. Let S be the schedule produced by the distributed voting algorithm for a set of update transactions. (We assume that all actions of transactions that were rejected by the distributed voting algorithm have been removed from S. Since rejected transactions only read data, their actions are not important in determining end equivalence.) Let $T_1, T_2, \ldots T_n$ be the transactions in S (i.e., the transactions that completed successfully) and let $ts(T_i)$ be the timestamp of transaction T_i, $1 \leq i \leq n$.

STEP 1. We transform schedule S into schedule S′ by moving all write actions that were not performed because of a timestamp conflict. That is, for all occurrences

$$S = (\ldots(T_i, w, e), \ldots (T_j, w, e), \ldots)$$

where $ts(T_i) > ts(T_j)$, we move action (T_j, w, e) to immediately precede (T_i, w, e). The new schedule S' is end equivalent to S because the effect of action (T_j, w, e) is null in S and is overwritten (without having been read by any other action) in S'.

Notice that in schedule S' all write actions are "normal" actions. That is, write actions in S' actually write a value (and a timestamp) into the database. Also notice that there can be no read actions of T_j between actions (T_i, w, e) and (T_j, w, e) in schedule S. To see why this is true, let x be the node where both T_i and T_j receive OK votes. Also, let z_j be the instant when T_j gets its OK vote at node x, and let z_i be the instant when T_i gets its OK vote at node x. Since $ts(T_j) < ts(T_i)$, z_j must occur before z_i. Since all reads of T_j occur before z_j, and z_i occurs before (T_i, w, e), then all reads of T_j must occur before action (T_i, w, e). This last statement implies that the actions of transaction T_j are still in a proper order in schedule S'. In other words, all the reads still precede all the write actions of T_j in S'.

STEP 2. We now have to show that S' is consistent and thus equivalent to some serial schedule. To do this, we will show that if $T_p \prec T_q$ in S', then $ts(T_p) < ts(T_q)$. Once we show this, it immediately follows that "$\prec$" is acyclic and that S' is equivalent to a serial schedule.

If $T_p \prec T_q$, then for some $i > j$,

$$S' = (\ldots(T_p, a_i, e), \ldots (T_q, a_j, e), \ldots)$$

where (1) either a_i or a_j is a write action and (2) there is no k such that $i < k < j$ and $c_k = e$ and $a_k =$ write. There are three cases we must consider now.

Case 1 of step 2. $a_i = a_j =$ write. In this case, $ts(T_p)$ must be less than $ts(T_q)$ because otherwise, in step 1, we would have moved (T_q, a_j, e) to precede (T_p, a_i, e).

Case 2 of step 2. $a_i =$ write and $a_j =$ read. When T_p writes e, it also writes timestamp $ts(T_p)$. Thus when T_q reads e, it will see this value and T_q will be assigned a timestamp larger than $ts(T_p)$. Therefore, $ts(T_p) < ts(T_q)$.

Case 3 of step 2. $a_i =$ read and $a_j =$ write. This case is somewhat more complex than the previous two. First notice three important facts: (1) Notice that T_p and T_q conflict because the write set of T_q shares item e with the read set of T_p; (2) Notice that since both T_p and T_q were accepted by a majority of OK votes, there must be a node x that voted OK for both updates; (3) Let t_0 be the timestamp read by T_p in action (T_p, r, e). Since timestamps for data items can only increase in value, t_0 must be less than $ts(T_q)$, which is the value of the

timestamp written by T_q in action (T_q, w, e). Within case 3, there are now two subcases we must now consider:

Subcase A of case 3 (step 2). Suppose that T_p arrived at x before T_q did. If at the time when T_q arrives, T_p has not been performed, then T_q will be delayed (because T_p and T_q conflict). Hence, in any case, T_q will see T_p performed at x. After T_p is performed at x, the clock at x must have a higher value than $ts(T_p)$. Since $ts(T_q)$ must be larger than the clock reading at node x when T_q receives its vote, we see that $ts(T_q) > ts(T_p)$.

Subcase B of case 3 (step 2). Now suppose that T_q arrived at node x before T_p did. This implies that when T_p votes at x, it sees a timestamp of $ts(T_q)$ for item e. Since this value is less than t_0, T_p will be rejected. Since this is a contradiction, T_q will not arrive before T_p.

We have shown that in all cases, $ts(T_p) < ts(T_q)$, and thus, $T_p \prec T_q$ implies that $ts(T_p) < ts(T_q)$. This in turn implies that S' is equivalent to some serial schedule and that S is end equivalent to that same serial schedule. Hence, the distributed voting algorithm provides end (or convergence) consistency for updates. (*End proof of theorem 4.*)

The result of theorem 4 implies that any update transaction T_p sees a consistent view of the database when performed with the distributed voting algorithm. To check this, consider the set of values read by T_p in schedule S, where as before, S is the schedule produced by the distributed voting algorithm. These values are exactly the same values read by T_p in schedule S' produced after step 1 above. This is true because the values read by any action are not affected by the transformation of step 1. Since S' is consistent, we find that T_p in S does indeed see a consistent view of the database. Thus, the distributed voting algorithm satisfies one of the requirements for a concurrency control mechanism. (See chapter 2.) It is easy to show that the distributed voting algorithm satisfies the second requirement. Suppose that at a given time we stop receiving new transactions and finish processing all existing transactions. The schedule obtained from this, S, is a valid distributed voting schedule. Schedule S is end equivalent to a serial schedule S'. Since S' is serial, it leaves the database in a state where all the implicit consistency constraints are true. Thus, S also leaves the database in the same state.

(It is hard to compare end consistency to any of the degrees of consistency defined in [GRAY76]. In a sense, end consistency is similar to degree 3 consistency because they both provide protection from arbitrary transactions. That is, with consistency degrees 0, 1, or 2, there are transactions that may leave the database permanently inconsistent. Neither degree 3 nor end consistency have

this problem. On the other hand, end consistency seems weaker than degree 3 consistency because with end consistency (e.g., like the distributed voting algorithm) the databases at a node may be inconsistent between "perform update" messages and not useful for local consistent queries.

Earlier, we showed that local concurrency control is not enough for consistent queries in the distributed voting environment. However, the observation that updates do see a consistent view of the database suggests a way to execute consistent queries. Simply handle queries as if they were updates with an empty write set (i.e., dummy update). Thus, queries will follow the voting protocol and will see a consistent view of the database. Unfortunately, this method is much less efficient than local reads with concurrency control.

By generalizing the example of section 3.1 to an N node network, we can see that any consistent query algorithm will have to visit at least a majority of nodes in order to guarantee consistency. Therefore, the efficiency of any other consistent query algorithm for the distributed voting environment will be similar to the efficiency of the query algorithm described in the previous paragraph. (It might be possible to modify the distributed voting algorithm in order to allow consistent queries to be executed at a single node. For example, by including in the "perform update" messages the timestamps that were read at a transaction's originating node, and by forcing nodes to wait until they see these timestamps locally before performing the update, we can force the "perform update" messages to be executed in the correct order at all nodes. We have not studied this modified algorithm and its performance.)

3.3 Current Queries in the Distributed Voting Environment.

The currency requirement for queries in the distributed voting environment can be expressed as follows: A current query submitted to the system at time t must reflect any updates that have been accepted at any node before and up to time t. In the following discussion, we assume that a node that accepts an update transaction performs the update locally before honoring any queries. Thus, the effect of any update accepted at node x at time t will be seen by all queries received at node x after time t.

Just as in the centralized locking environment, there are many alternatives for dealing with current queries. One simple algorithm for query Q submitted at node x at time t is to send Q to all nodes and to ask them to execute Q. (This can be done serially or in parallel.) When node x collects all the answers,

it chooses the values with the most recent timestamps. That is, for each item referenced by Q, node x will collect N values and will choose the value which has the largest timestamp (where N is the number of nodes). The values obtained in this fashion will reflect all updates accepted before time t (and possibly other updates accepted after time t) because these updates have been performed at least at one node.

The number of nodes consulted for a current query can be reduced to a majority of nodes by having query Q also check the list of pending updates at each node it visits. A pending update at node x is an update that node x has voted OK on, but an accept or reject message has not yet arrived at node x. If query Q finds a pending update at node x, then it should wait until the outcome of the update is decided before proceeding. This wait is necessary because there is a chance that these pending updates are performed before time t at a node that Q does not visit. After having visited a majority of nodes, Q can guarantee that all updates performed before time t have been reflected in the data it has read.

It is impossible to have a current query algorithm that visits less than a majority of nodes. If the nodes not visited by query Q constitute a majority, then they could have accepted an arbitrary number of updates, before time t, without Q and the rest of the nodes finding out.

3.4 Current and Consistent Queries in the Distributed Voting Environment.

Current and consistent queries in the distributed voting environment can be processed as dummy updates using the distributed voting algorithm. As we have seen in section 3.2, this algorithm provides any transaction with a consistent view of the data. This view is also current because this algorithm also makes transactions wait for other pending transactions, as was described in section 3.3.

3.5 Summary.

In section 3 (and its subsections), we have shown how current as well as consistent queries can be executed when the distributed voting algorithm is used to coordinate updates. The consistent query algorithm is less efficient than its counterpart in the centralized locking environment. Similarly, most of the current query algorithms seem to be less efficient than the centralized locking versions.

The main reason for these differences is that there are no sequence numbers to order updates in the distributed voting system. Timestamps are used for sequencing in the distributed voting algorithm, but the timestamp of an update is not as helpful as the sequence number of an update. For example, from an update's timestamp we cannot tell how many other updates have been previously performed in the system.

4. QUERIES WITH THE ELLIS TYPE ALGORITHMS.

4.1 Consistent Queries.

It is easy to study the consistency of the Ellis type algorithms. (See chapter 3.) The original Ellis ring algorithm (OEA) actually performs updates one at a time, so that all schedules produced by that algorithm are not only consistent but serial as well. In this case, consistent queries can be processed locally at any node with local concurrency control. The proofs of these facts are so simple and similar to our previous proofs that we will not present them here.

The other more efficient Ellis type algorithms (MEAS and MEAP) also provide the same type of consistency as the OEA. In the modified Ellis algorithm with sequential updates (MEAS) and in the modified Ellis algorithm with parallel updates (MEAP), an update transaction T with base set B does not perform any action at node x until all other transactions that previously referenced items in B have been completely finished at node x. This implies that if $T_p \prec T$ (i.e., T depends on T_p), then T_p cannot depend on any transactions that follow T in a legal schedule. Therefore, any cycle $T_p \prec T \prec T_1 \prec T_2 \prec \ldots \prec T_n \prec T_p$ is impossible and any schedule produced by the MEAS or the MEAP algorithms is consistent. It is also easy to show that local concurrency control provides local queries with a consistent view of the database.

Therefore, all the Ellis type algorithms are just as efficient for processing consistent queries as the centralized locking algorithm.

4.2 Current Queries.

In the Ellis type algorithms, all nodes must be "locked" before an update is performed at any node. This means that any node can find out if a new update is being performed somewhere in the system. This implies that current queries can be executed locally without consulting any other nodes. This is an important advantage of these algorithms.

The current query algorithm for the OEA environment is very simple and efficient. When a current query Q arrives at node x at time t, this node checks the database state. If the database is idle at node x, then all updates that have been performed up to that instant (time t) have been performed locally and Q can be executed immediately. If the database state is passive, then some update A might have been performed before time t at some other node but not at node x. Therefore, Q must wait until the next "perform update" message (corresponding to A) arrives. After A is performed at node x, query Q can be executed. If the database state is active, then node x itself is trying to obtain locks for a new update A'. But since the state is still active, the locking process for A' did not finish before time t and A' was not performed anywhere before time t. Thus in this case, Q can also be executed immediately.

The current query algorithm for the MEAS and the MEAP algorithms are very similar to the above algorithm. The only difference is that the current query algorithm must check the state of every item referenced in the query. For every passive state found, the query is delayed until the update involving that item arrives. After these waits, the query can be executed.

5. PERFORMANCE OF THE QUERY ALGORITHMS.

We have seen that for every update algorithm there are consistent and current query algorithms. Some of these new algorithms are more efficient than others. For example, the consistent query algorithm corresponding to the centralized locking and the Ellis type algorithms are much simpler and efficient than the consistent query algorithm corresponding to the distributed voting algorithm. The current query algorithm for the Ellis type algorithms, is more efficient than the current query algorithm for the centralized locking algorithm, which in turn is more efficient than the one for the distributed voting algorithm. (The second part of this last statement may not be true in some cases where the central node in the centralized strategy is congested. See chapter 6.)

The overall system performance will strongly depend on the query types and the fraction of the total transactions that they represent. For example, if most transactions are free reads, then all algorithms will perform identically. If most transactions are current queries, then a system that uses an Ellis type update algorithm will perform the best because current queries can be executed with very low overhead. If most transactions are consistent queries, then the centralized locking or an Ellis type algorithm will perform best. The case where all transactions are updates has already been studied, and we discovered that the centralized locking algorithm performs best in most cases of interest.

When we have different transaction types running, the choice of algorithm will depend on the particular percentages of transaction types and the system parameters. However, the trends should be obvious by now. For example, if p transaction are updates while $1 - p$ are current queries, then as p decreases towards 0, the centralized locking algorithm will be less attractive as compared to the original Ellis ring algorithm (OEA). At some low value of p, the OEA will become the best choice. A detailed simulation could give us a good approximation to the value of p where the switch-over occurs. But unless we have a particular system in mind, the exact value of p is hard to evaluate. Therefore, we will not perform any such simulations here.

6. SOME CONCLUSIONS.

From our study we can reach the following general conclusion. Out of the algorithms we studied, the centralized locking algorithm (i.e., MCLA-h) seems to be the best algorithm for handling a combination of update and read only (query) transactions except if:

(1) The central node is heavily loaded (due to the centralized locking) and updates constitute a large part of the load. In such cases, the distributed voting algorithm may handle updates more efficiently. If updates are not frequent, then the increased response time of updates in the centralized locking system will be offset by the reduction in the response time of queries.

(2) Most transactions are current queries. In this case, the Ellis type algorithms will operate more efficiently because current queries can be executed at a single node.

CHAPTER 10

TRANSACTIONS WITH AN INITIALLY UNSPECIFIED BASE SET

Up to now we have assumed that all transactions specify fully at their inception the set of items that they will reference. We have called this set of referenced items the base set of the transaction, so we call this assumption the *base set assumption*. In this chapter we will study transactions that do not specify their base set initially. In other words, these transactions must read some item values before deciding what other items to read.

In section 1 we discuss how the fact that transactions do not initially specify their base set affects the algorithms we have presented in previous chapters. In section 2 we study how the MCLA-h algorithm can be modified to deal with transactions that do not specify their base set initially. (The rest of the algorithms we have studied either are not affected by the base set assumption, or the modifications needed by them are very similar to the modifications given for the MCLA-h algorithm.) Finally, in section 3, we discuss the performance of the modified MCLA-h algorithm.

1. OVERVIEW.

The elimination of the base set assumption only affects some of the algorithms we have studied. Notice that none of the query algorithms of chapter 9 are affected. In all these query algorithms, all the data needed by a query is read at a single node with local concurrency control, so that a query does not have to specify its base set beforehand. (Recall that in some query algorithms like the DVA algorithm, the query must visit other nodes before completing. But even in these cases, all the data is read at a single node.) In other words, to read data at a node, a query can request some *local* locks, read the data that is locked locally, perform some computations based on the data read, and then decide that it wants to read more data. To read the additional data, the query

simply requests more local locks, reads the additional data, and possibly repeats the cycle. Thus, it is not necessary that queries specify their base set initially. (Notice that this can lead to local deadlocks.)

The update algorithms that are not based on locking are not affected by the base set assumption either. The algorithms that do not use locking are the DVA and the CCA algorithms. No modification is required in these algorithms in order to handle transactions that do not specify their base set initially. In the distributed voting algorithm, the item values and their timestamps are read initially before any votes have been obtained. Thus, the node that is doing this initial read is free to read some values, then compute some, and then based on the computations, it can decide to read other items. When the read phase has completed, the voting phase can commence just as if all the item values had been read in a single operation. The other algorithm that does not use locks is a complete centralization algorithm where all values are read and all computations performed at the central node. In this case, there is no problem with initially unknown base sets either.

Therefore, in this chapter we will only study the locking update algorithms. Since the modifications needed to cope with transactions that do not specify their base set initially are very similar for all the locking algorithms, we will concentrate on one of the update algorithms, the MCLA-h algorithm.

In this chapter we will continue to assume that no failures occur in the system. We still assume that the database is completely replicated at all nodes. Finally, since we will only study update algorithms, in this chapter we can assume that all transactions are update transactions (i.e., no queries).

2. STRATEGIES FOR THE MCLA-h ALGORITHM.

In the case of the centralized locking algorithm (MCLA-h), the base set is needed beforehand so that locks can be requested from the central node. When the base set is initially unknown, it will be impossible to request all locks as a first step. There are three possible alternatives:

STRATEGY 1. Enlarge the base set. In some cases we can get away with simply requesting a few more locks from the central node in order to cover all alternatives. For example, a transaction might read the balance of a bank account in order to record a withdrawal. If the new balance is negative, a special entry must be made in the overdraft record. We can process this transaction by simply

requesting locks for the balance and the overdraft records, even if we are not sure we will need the second item. The advantage of this strategy is that we only request locks once. However, the disadvantage is that we unnecessarily restrict concurrent execution of other updates because of the extra locks we hold. In our example, many transactions may request the lock for the overdraft record even though in most cases they will not use it. Thus, most of the transactions could run concurrently but will actually run serially because they unnecessarily request the lock for the overdraft record.

STRATEGY 2. Request locks as they are needed. When this strategy is followed, an update first specifies an initial set of items it would like to read. Locks are requested and obtained, and the update reads the items and computes. If the transaction discovers that it would like to read some more items, then it requests more locks from the central node. After these new values are read, more could be requested and so on. This method does not seem very attractive if the number of requests to the central node is not small. In the centralized locking algorithm, we want to avoid "visits" to the central node as much as possible because this is the system bottleneck.

However, for many transactions, a second or third lock request will be very rare, and most transactions will run with only a single request to the central node. In the bank example above, we can request the lock for the overdraft record only when it is actually needed. Thus, only the few transactions that find negative balances will have the extra overhead of requesting the additional lock.

When locks are requested as they are needed, there is a danger of deadlocks. These deadlocks can be detected by the central node. Some transactions may have to be backed out, but this does not represent a problem because no item values have been modified by a transaction requesting locks.

STRATEGY 3. Read without locks and then request locks needed. (A similar strategy is followed by the distributed voting algorithm.) Using this method, an update reads all the data it needs at its originating node, without holding locks. This way, an update can read data and compute in stages. Once the update is ready to write the new values, locks are requested from the central node for all items that were referenced.

Of course, the problem with this strategy is that by the time the locks are obtained, the values that were initially read might be invalid. Therefore, we need a mechanism for detecting conflicting updates that were performed during the read. When such a conflict is discovered, the update will have to be restarted. There are several alternatives for processing the second attempt.

One way is to release all the locks obtained and to start the update once

more from scratch. (Dummy updates with the update's sequence number will have to be sent out to all nodes.) On the second attempt, we may use strategies 1 or 2 (above) because we might now have a better idea as to what items will be needed.

Another alternative is to keep the locks form the first attempt, reread the data to obtain the current values, and then to proceed using strategy 2. In other words, the fact that some updates conflicted and modified some values that were read does not necessarily imply that another base set will be needed.

For example, if we use this idea on the bank account example, the update will read the balance (with no locks) at node x and will then discover that it needs the overdraft record. This item will also be read at node x, and the first phase will complete. Then both locks will be requested, but when they are granted, may discover that some other conflicting update was performed between the time the update started reading and the time when the locks were granted. Assuming that a conflict did occur, the data read is obsolete, but there is no need to throw away all our previous work. Chances are that the new values will not increase the base set needed. So we can reread the balance and the overdraft record locally and reexecute the computations. When we do this, we may find a new overdraft record value, but this should not force us to read other items. We may also find a new balance and that we do not need the overdraft record after all, but this does not force us to read other items either. (That is, the base set is now smaller, so it does not matter if we have obtained an extra lock.) In some special cases, we may find a completely different balance which may force us to read some other items. (For example, we may now need an "overflow" record because the account has too much money !) In many applications, occurrences of drastic base set changes due to item value changes will be rare. In such cases, a read without locks strategy will allow most updates to complete with only one lock request to the central node.

2.1 Mechanisms for Detecting Conflicts.

The mechanism to detect conflicting updates can be fairly simple. When an update starts its reading phase (with no locks) at node x, it makes a copy of the set of performed updates. Let "Copy" be this copy of Done-set[x]. The values read by the update will reflect all updates in this set. When the locks are obtained later, the update will receive a sequence number s and a hole list l. If all updates with lower sequence number than s and not in hole list l are in set

Copy, then the data initially read is current and the update can proceed (i.e., it can be performed at all nodes). If some update is missing from set Copy, then the update must restart as outlined at the end of section 2.

As a matter of fact, the check can be made at the central node itself after all locks are granted. The central node can send out the "perform update" messages directly because the new update values have already been computed. This way, we save some time because we do not wait for the grant message to reach node x. Furthermore, if the "perform update" messages are sent out by the central node itself, the locks obtained by an update will be released immediately, so we might as well do away with locking completely. We can also do away with the hole lists because the list of updates that have obtained locks but not released them (i.e., the hole list) will always be empty. Only the sequence numbers issued by the central node will still be used to properly sequence each update.

In summary, this is how an update would be processed. When update A is received at node x, the sequence number of the last update performed at node x is recorded. Let this number be s. Then update A will read the values it needs and computes some new values. These new values, together with sequence number s are sent to the central node for authorization. When the central node receives them, it checks that the latest sequence number issued by the central node is indeed s. If this is not true, A is rejected and node x is informed. (In this case, update A must be started from scratch because no locks are held by it.) If s is the last sequence number issued, then A is accepted. Update A is assigned the next sequence number (i.e., $s+1$) and the "perform update" messages (with the new values that were computed at node x) are sent out to all nodes.

The problem with this mechanism (the original one we described or the simplified version described in the previous paragraph) is that some updates may be unnecessarily rejected. For example, suppose that an update A has started reading data at node x and a second update A', which does not conflict with A, is performed at node x. When update A arrives at the central node for authorization, it will be rejected because it did not see update A'. (Notice that A' has a lower sequence number than A.) Nevertheless, update A could have been accepted because A' did not modify any of the values read by A.

There is another mechanism which we can use for detecting conflicts which is more efficient than the one described above. The idea is that the local concurrency controller at each node detects the conflicts. As an update A is reading at node x without global locks (i.e., those issued by the central node), it sets local locks. These local locks are held until A obtains global locks and completes. Therefore, when a "perform update" message for a conflicting update A' arrives

at x, the concurrency control will immediately detect the conflict. If update A is still reading, it should be halted and restarted in order to avoid wasting more time on it. If update A has finished reading and is waiting for locks from the central node, then node x remembers that the data obtained by A is no good. When the grant message for A arrives, either the global locks are released and A restarted, or the data is reread as described previously. In any case, the local locks of A are released and update A' is performed.

After update A obtains global locks, it must still wait until all updates with lower sequence number and not in its hole list are performed locally at node x. If after this step no conflicts have been detected by the local concurrency control, then update A can be performed. The new values produced by A are stored in the local database, the local locks are released, and the "perform update" messages are sent out to all nodes as before.

With this new mechanism, only updates that actually read values that were subsequently modified by another update will be rejected. The use of the local concurrency control to detect conflicts does not produce any overhead since this control is required at all nodes in order to perform the updates correctly.

2.2 The Other Locking Algorithms.

In the case where the other locking algorithms are used for updates that do not specify their base set initially, we have similar problems to the ones of the MCLA-h algorithm. The problems do not appear in the original Ellis algorithm (OEA) because, there, updates lock the entire database regardless of what they will read. But in the other algorithms (e.g., MEAS, MEAP, WCLA, TWCLA), updates must obtain locks for the items they will read, before they actually read any data. In these cases, we have the three same options that we had for the MCLA-h algorithm: (1) enlarge the base set, (2) request locks as they are needed, and (3) read without locks and then request locks. Since these solutions are so similar to the ones for the MCLA-h algorithm, we will not discuss them here.

3. PERFORMANCE OF THE DIFFERENT STRATEGIES.

Only the performance of the locking algorithms is affected by the elimination of the base set restriction. All other update algorithms operate as before, and we

can therefore use the performance results obtained in chapter 6. In this section we will concentrate on the change in performance of the MCLA-h algorithm due to the initially unknown base sets. The elimination of the base set restriction will affect the other locking algorithms in a similar way.

Studying the performance of the MCLA-h algorithm in the case where the base set is initially unknown is not simple. We have outlined several different strategies that could be used for this case (e.g., enlarge the base set, read without locks, etc.) and in each case the performance depends on how well the update transactions suit the strategy. Such factors as how many extra locks are needed to cover all possible base sets (for the enlarge the base set strategy) or how many lock requests to the central node are needed by an update (for the request locks as needed strategy) will entirely define the performance of these algorithms. Unfortunately, the update model that we have used so far does not take into account any of these factors, and it is hard to add the factors without application knowledge.

Fortunately, at least it is possible to study the "read without locks" strategy and obtain a rough estimate of how this strategy performs. In appendix 8, we present a simple analysis of the "read without locks" strategy which provides us with an approximation for the average response time of updates. (The local conflict detection mechanism is used; an infinite hole size limit is assumed.) The results are plotted in figure 10.1. We can observe that the increase in response time is very small. That is, the number of updates that are rejected is so small that it only becomes significant when the system is heavily loaded. In these cases where the system is close to saturation, a small load increase due to the rejected updates can increase the average response time of all updates significantly. In all other cases, the increase in average response time is small.

The results of figure 10.1 are obtained assuming that updates reference items at random within the database. Of course, in many applications, there are certain items that are frequently referenced by updates. In this case, the number of conflicts and update rejections will be larger, and the response time of updates will be worse than what is shown in figure 10.1.

However, the assumption of random reference was also made for the analysis of the other update algorithms, so that it is fair to compare the results of figure 10.1 with the previously obtained results. In figure 10.2, we graph the results for the "read without locking" MCLA-h algorithm together with the results for the distributed voting algorithm. To make the comparison realistic, we should make the "I_s" parameter for the centralized locking algorithm smaller than for the distributed voting algorithm. (See chapter 6.) Recall that the I_s parameter

Figure 10.1

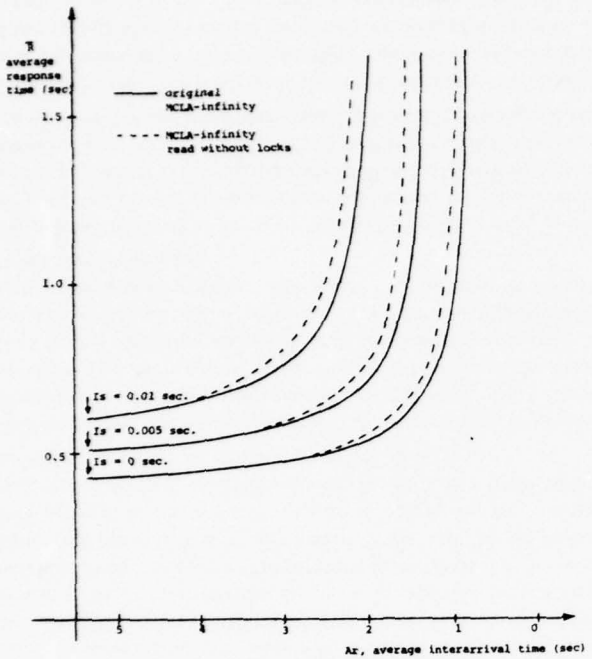


Figure 10.1. The MCLA-h algorithm: The read without locks strategy. N=6, M=1000, Bs=5, Id=0.025 sec., T=0.1 sec., h=infinity.

Figure 10.2

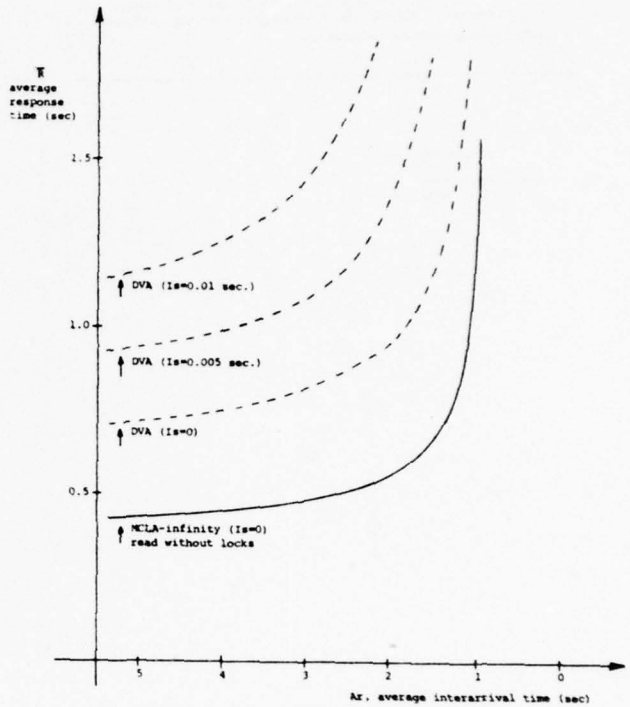


Figure 10.2. The read without locks MCLA and the DVA algorithms. N=6, M=1000, Bs=5, Id=0.025 sec., T=0.1 sec., Rt=1 sec.

(IO time slice) is the IO time needed to read or write a timestamp or a lock. For a locking algorithm, I_s is near zero because the lock table can be kept in main memory as a hash table. Therefore, in figure 10.2 we give the results for the MCLA-h algorithm (with read before locking, h = infinity) with $I_s = 0$. In chapter 6, 0.025 seconds was used as a typical value for I_s for the DVA algorithm, but by using caches this value may be reduced somewhat. In figure 10.2, we give the response time of the distributed voting algorithm for $I_s = 0.01$, $I_s = 0.005$, and $I_s = 0$ seconds because a true value should be in this range (probably closer to 0.01 seconds). The MCLA-h algorithm performs better than the distributed voting algorithm in most cases of interest. If $I_s = 0$ for the distributed voting algorithm (which is unlikely), this algorithm performs better than the centralized locking algorithm when the system is heavily loaded. (This is not shown in figure 10.2. This effect appears in figure 6.2 of chapter 6.) However, for the more realistic values of $I_s = 0.01$ or $I_s = 0.005$ seconds, the MCLA-h (read without locks) performs better for all values of the interarrival time (A_r).

In figure 10.3 we compare the results for the MCLA-h algorithm (read without locks, h = infinity) with the completely centralized algorithm, CCA. In the completely centralized algorithm, all updates are totally performed at the central node. Recall that in this algorithm no global locks are required; the local concurrency control at the central node is sufficient. Thus, for this algorithm, the IO time slice parameter I_s is always zero. Also notice that updates do not need to specify their base set beforehand.

The MCLA-h (read without locks, h = infinity) algorithm with $I_s = 0$ performs better than the completely centralized algorithm, as can be seen in figure 10.3. If the I_s parameter is increased for the MCLA-h algorithm, then the completely centralized algorithm can perform better. However, this is not a fair comparison because $I_s = 0$ is the most likely case for the MCLA-h algorithm.

The results obtained from the analysis of the "read without locks" strategy for the MCLA-h algorithm show that our previous conclusions regarding the performance of the update algorithms (for completely duplicated databases, updates only) are not altered. The MCLA centralized locking algorithm still gives the best average response time for updates in most cases of interest, even when the base set is initially unknown. Furthermore, in the analysis of the MCLA-h algorithm in appendix 8 we did not consider many possible simplifications that could improve efficiency. (For example, we did not consider that updates could be aborted while they were reading and computing because of a conflict. In the analysis, we assumed that all updates requested locks from the central node, even when the updates knew for sure that they would be rejected.

Figure 10.3

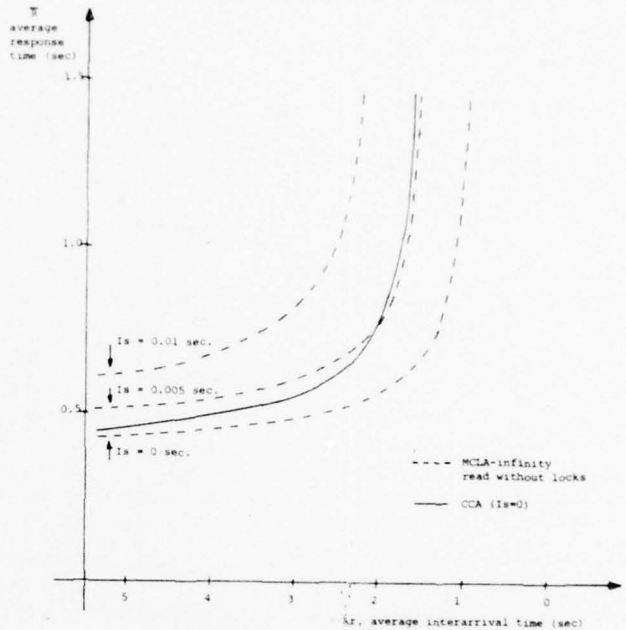


Figure 10.3. The read without locks MCLA and CCA algorithms. N=6, M=1000, Bs=5, Id=0.025 sec, T=0.1 sec.

We have not analyzed the "increase the base set" or the "request locks as needed" strategies. However, we have discovered that the "read without locks" strategy performs fairly well. The other strategies should only be used when they improve the performance. That is, if for a give application the other strategies do worse than the "read without locks" one, then the latter one should be used.

CHAPTER 11

PARTITIONED DATA AND MULTIPLE CONTROLLERS

In most distributed databases, the data is not completely replicated at all nodes in the system as we have assumed up to this point in this thesis. In this chapter we eliminate this assumption. We study distributed databases where the data is partitioned and we present several algorithms for processing transactions in this environment.

In section 1 we present a partitioned data model and we show how the algorithms of chapter 3 can still operate in this case. In section 2 we introduce the concept of multiple controllers and we discuss how a partitioned distributed database system with multiple controllers can operate more efficiently. In section 2 we also present an update processing algorithm for the partitioned data, multiple controller case. Then, in section 3, we study query processing algorithms for the partitioned data, multiple controllers environment, and we study their performance.

To simplify the presentation, we will first assume that no failures occur in the system and that transactions specify their base set initially. Then, in section 4, we will consider how these assumptions can be relaxed in the partitioned data, multiple controller environment. In sections 4.1 and 4.2 we will look at transactions that do not specify their base set beforehand, while in section 4.3 we will discuss how the algorithms presented in this chapter can be made crash resistant.

1. PARTITIONED DATA.

In this section we will discuss how our algorithms can be modified to handle partitioned databases. Up to this point in our research, we have assumed that every data item is replicated at every node in the system. In a partitioned distributed database, items are not replicated at all nodes. As a matter of fact, some items might not be duplicated at all. That is, there might exist a single

copy of some items. From the point of view of a single node, it has a fraction or a partition of the database. This partition can be identical to, completely disjoint from, or can overlap the partitions at other nodes.

1.1 The Partitioned Data Model.

We will now extend the database model we defined in chapter 2 in order to model partitions. We chose a simple model which explicitly shows where the data is replicated. As before, the database is a fixed set of M shared named resources called items [ESWA76]. Each item has a name and some values associated with it. For simplicity, we use the integers between 1 and M as the names of the items in the database. (E.g., item 10, item j). In addition, each item i has associated with it a set $S(i)$. Set $S(i)$ is the set of nodes which have a copy of the value of item i. That is, each element of $S(i)$ is the node identification number (between 1 and N) of a node where a value of item i is stored. We assume that all sets $S(i)$ are not empty. We represent the values associated with item i by $d[i, x]$, where x is a node in $S(i)$. (For nodes y not in $S(i)$, $d[i, y]$ is undefined.)

In our model, the storage locations of one item are completely independent from the location of other items. However, for convenience we may group items that have identical storage characteristics into "fragments". A fragment F is a set of items that have the same S sets. We use the notation $S(F)$ for the set of nodes where F is stored. (That is, $S(F)$ equals $S(i)$ for all items i in F.)

From the point of view of the user, the locations of the items in the system (i.e., $S(i)$) is irrelevant. The user transactions still view the database system as if it were a single local database where all item values existed. Therefore, our model of a transaction will be not be changed. A transaction T simply reads the values of a set of items (the base set), performs some computations and generates new values for a subset of the items that were read (the write set). As before, we also have a set of consistency constraints or assertions defined on the database. These user constraints are specified only in terms of the items; the location of the item values is again irrelevant. (Later, the location information can be added to the user constraints as was done in chapter 9.) We assume that a transaction that is run by itself on a single database with all items values available will preserve the consistency of the database.

1.2 Transaction Processing With Partitioned Data.

Let us assume that a transaction T specifies fully at its inception the items it will reference; that is, T specifies its base set $B_S(T)$. (We will study the elimination of this restriction in section 4.) For the time being, we will concentrate on update transactions. Possible simplifications for read-only transactions will be discussed in section 3.

We also assume that the system has a *directory* which gives the location of the item values. A directory is a mapping that produces the set $S(i)$ (or $S(F)$) given the item name i (or the fragment name F). By consulting the directory, a transaction T will be able to find out the set $S(i)$ corresponding to every item in $B_S(T)$.

The directory itself is a distributed database and can be partitioned as was described in section 1.1. For example, the directory information for a certain fragment F can be located at some nodes and not at others. A transaction needing to find out where F is located must visit one of the nodes that have a copy of the $S(F)$ set. There are two main problems that arise now: (a) How does the transaction know what nodes have a copy of $S(F)$, and (b) How can we update the elements of $S(F)$? These are hard problems in themselves. They correspond to the area of distributed directory management, which we consider to be beyond the scope of this thesis. Nevertheless, here we will make some short comments before moving on to the problems we really want to address in this chapter.

We do not wish to have a "second level" directory which can tell us where the "first level" directory is located because this would only push down the problems to this "second level" directory. One simple solution is to replicate the complete (first level) directory at all nodes. Another solution is to find out the location of data by broadcasting a request "Where is item i (or fragment F) located?" to all nodes. The nodes with a copy of the corresponding partition of the directory would respond "Item i (or fragment F) is located at nodes $S(i)$ (or $S(F)$)" and the nodes that do not have the right information would respond "I do not know". By remembering in caches the location of the commonly referenced items, nodes can speed up the lookup procedure. Notice that if the information in a cache becomes obsolete, there is no serious problem. For example, suppose that a node x incorrectly believes that a value of item i is located at node y. When node x requests the value of item i from node y, node y will inform node x that it does not have the value. Node x will then have to locate item i through the longer broadcast procedure. Also notice that it is highly convenient for the nodes in $S(i)$ themselves to have the directory information for item i.

The directory information can be updated, but the concurrency control mechanism for doing this is different from the one we discuss in this chapter. The reason for this is that considerably more safeguards must be taken when modifying a directory. In this thesis we will not discuss directory updating; we will assume that the directory is static.

Once the system has all the necessary information, it has to decide how to run the transaction. There are two distinct issues involved here. The first, which we will study in this chapter, is how to provide each transaction with a consistent view of the database; that is, how to deal with the concurrent execution of the transactions when the data is partitioned. A second and different issue is how and in what order to perform the read operations and computations in order to minimize the effort and delays involved. In our previous research we did not have this second problem because all the data required by a transaction was available at any node. But now there is a choice to be made as to where the data will be obtained, and the different order of performing the read and computation steps at different nodes can greatly influence the effort required to perform a transaction. In this chapter we will totally avoid this problem because it is very hard and would obscure the solution to the first. Therefore, we assume that the values of the items in the base set of a transaction can be obtained from any node that has them and this can be done in any order. (Also notice that the database model we have chosen is not good for studying the second problem.)

We now concentrate on the concurrency problem for partitioned data. It turns out that any of the update algorithms for the completely duplicated case of chapter 3 can be extended to the partitioned data case. The reason for this is that a node does not need to have the value of an item, $d[i, x]$, in order to lock the item or vote on an update involving the item. Thus, the algorithms virtually remain the same. The main differences are that the item values may not be available locally and that the nodes must ignore "perform update" messages that involve items that they do not have.

Although this solution is simple, it has disadvantages in some cases. But before we discuss these problems and their solution, it will be helpful to illustrate the idea of using the algorithms of chapter 3 for partitioned data with the MCLA-h and the DVA algorithms. Some of the disadvantages will become evident as we discuss these two algorithms.

1.2.1 The MCLA-h Algorithm for Partitioned Data.

If we use the MCLA centralized locking algorithm with hole size limit of

h (MCLA-h) (see chapter 3), an update will be processed as follows. Assume that an update A is initially submitted at node x. The first step is for node x to request locks for all items referenced by A from the central node. Even though the central node might not have the value of the items needed by update A, it still grants locks using the MCLA-h protocol. After a possible delay at the central node, update A obtains the locks, a sequence number and a hole list. The hole list is the list of the currently executing updates and is used to speed up the execution of update A. A grant message to node x informs the node that A is ready for execution because no other updates can interfere with the items referenced by A. Thus, node x sets out to read the values of the items in the base set of A ($B_S(A)$). First node x must obtain set $S(i)$ for all items i in $B_S(A)$. (As mentioned earlier, this is done with help of a directory.)

After this, node x knows where it can read the values for the items in $B_S(A)$. Reading the values might involve communicating with several other nodes. At any of these nodes where values of A are read, the read operation must be delayed until all updates with sequence numbers less than A's sequence number (and not in A's hole list) are performed. (See chapter 3.) This guarantees that update A obtains a consistent view of the database, even though the values for A have been read at several nodes. In other words, all and only updates with sequence number less than A's sequence number will be reflected on the items read by update A. After node x obtains all the values needed by update A, it proceeds to compute the new values for the items in the write set of A, $W_S(A)$. Finally, a "perform update" message, which includes the new values, is sent to all nodes. Even though a node does not have any item values involved in update A, it still must receive the "perform update" message because the node needs to know that A has been performed. Such nodes simply add A's sequence number to their list of performed updates and do nothing more. When the central node receives the "perform update" message for update A, it releases A's locks and deletes A's sequence number from the hole list.

1.2.2 The Distributed Voting Algorithm for Partitioned Data.

The distributed voting algorithm can also be used for partitioned data. Unfortunately, in order to be able to vote on updates, all nodes must keep the timestamps of all items, even though the nodes might not have the values of all items. This is analogous to the MCLA-h case where the central site must keep lock information for all items. Since the timestamp information is more voluminous than the locking information, this represents a serious problem for

the distributed voting scheme. Furthermore, the timestamp information must be kept at all nodes, while the locking information in the central locking strategy is only kept at a single site.

When a node x receives an update from a user, it proceeds as follows. First, the item values and their timestamps are requested from any nodes that have them. Then the voting protocol is followed in exactly the same fashion as before. When update A is accepted, the "perform update A" message (i.e., the "accept" message) must be sent to all nodes because all nodes must modify their timestamps for the items referenced by update A. Nodes that also have the value of an item referenced by A, update this value.

2. MULTIPLE CONTROLLERS.

There are two main disadvantages with the solutions for partitioned data we have proposed so far. Both disadvantages stem from the fact that there is a single unified control structure for the complete system. The first problem is that every node must be aware of all updating activity in the system. That is, every single node must process every update in order to record the timestamp or sequence number information regarding the update. This processing must be done regardless of whether the node contains any item values that are involved in the update. In the completely duplicated database case, it made sense to have a single control structure because all nodes had to process all updates anyway. However, in a partitioned database, this is no longer the case.

The second main disadvantage of the proposed solutions is that the nodes that enforce the concurrency control (i.e. the central node in the centralized locking algorithm or the voting nodes in the distributed voting algorithm) must resolve conflicts involving all updates. The single control structure residing at these nodes creates a performance bottleneck because all updates must pass through these nodes to obtain the proper authorization. If all transactions reference random items in the database (as we have assumed up to now), then there is little we can do to avoid these bottlenecks. However, in a partitioned database, we expect transactions to have special reference patterns that will allow us to eliminate the bottlenecks by having several independent control structures. We will illustrate this idea through an example.

2.1 An Example.

Suppose that a certain company has a distributed database system with two nodes. One of the nodes, node x, is located at one of the company's two plants. The database at node x contains all the data pertaining to that installation. Similarly, node y contains all the data corresponding to the second plant. The database is hence partitioned in a very natural way. We also expect that most transactions will deal exclusively with one of the database fragments. In other words, transactions that only reference data at one of the nodes will be very common. Of course, there will always be some transactions that will involve data at both nodes, but we expect the number of such transactions to be low.

If we choose a single centralized controller at say node x, we will have a very inefficient system. Similarly, if we select a distributed voting scheme where all transactions must obtain votes at both nodes, we will run into the same problems. A more efficient and natural way to solve the concurrency control problem in this example is to have two independent *controllers*, one for each database partition or fragment. The controller at node x can have total control over its fragment (i.e., the data at node x). Transactions that only reference items at node x will only have to communicate with this controller. Similarly, the controller at node y has control over its items and it can grant access to those items. This way we eliminate the bottleneck through the use of two controllers, each processing its share of the transactions.

The complication in this scheme occurs when processing the few transactions that reference items in both fragments. Such transactions must coordinate their update with both controllers. One way to do this is to first request "locks" for the items handled by one controller, and then to request the rest of the "locks". This protocol is not as efficient as simply requesting locks (or control) from a single controller, but the fact that only a few transactions require this "two level" protocol (as we assumed at the beginning of this example) should make the overall system performance good.

2.2 Controllers.

Returning to the general case, we see that we can have several independent controllers. Each controller will be in charge of the concurrency control of a set of items, and for convenience we assume that all the items supervised by the same controller have the same storage set $S(i)$. Hence, a controller is in charge

of a fragment of the database.

A controller is not a node. As a matter of fact, one controller may be distributed among several nodes. A controller is simply one or more software "modules" which exercise control of the data using any one of the previously studied algorithms. For example, a centralized locking controller for fragment F has a single module which grants locks for any update referencing items in F. A distributed voting controller for fragment F' has j modules on j different nodes. These nodes may or may not have values of items in fragment F'. An update involving any items in F' must get a majority of OK votes from the j modules. In order to vote, each module of the distributed voting controller must keep the latest timestamps it has seen for each item in fragment F'. If the module does not reside at a node where the values for F' are stored, then the module needs its own copy of the timestamp values. Controllers that use the other update algorithms of chapter 3 can also be designed.

2.3 Multiple Controller Model.

We will now extend our model to include the concept of "partitioned" or multiple controllers. Our model should emphasize the independence of data storage and the data control. That is, the distribution of the data in the system is different from the distribution of the control. Our model should allow us to have completely duplicated databases with a single overall controller, completely duplicated databases with multiple controllers, or partitioned data together with partitioned control. We would also like our model to make each controller completely independent from other controllers. Each controller must have total control over the items assigned to it, and we can use any algorithm (distributed or centralized) to enforce this control.

To extend the model, we associate each item i in the database with a controller $C(i)$. Thus, each item can be represented by the tuple $(i, V(i), S(i), C(i))$, where i is the name of the item, $S(i)$ is the set of nodes where values of i are stored, $C(i)$ is the name of the controller in charge of the item, and $V(i)$ is the set of values:

$$V(i) = \{\mathrm{d}[i,x] \mid x \in S(i)\}.$$

We use the integers between 1 and C_m to name the controllers in the system, where C_m is the total number of controllers. (Notice that C_m should be less than or equal to the total number of items M.) Each controller J ($1 \leq J \leq C_m$) has associated with it an algorithm $A(J)$, a set of nodes $M(J)$ where the modules for

the controller reside, and a set of items controlled by it, $I(J)$. Algorithm $A(J)$ is the protocol (e.g., centralized locking algorithm, distributed voting algorithm) that is used by the modules that compose the controller in order to perform the concurrency control of the items in $I(J)$. Each node in $M(J)$ has a module of controller J. For consistency, we assume that $C(i) = J$ for all items $i \in I(J)$ ($1 \leq J \leq C_m$). For simplicity, we have assumed that controllers are in charge of fragments. That is, $S(i) = S(k)$ for all $i, k \in I(J)$, where $1 \leq J \leq C_m$. We will also use the C notation to denote the controller of a fragment. In other words, $C(F)$ is the controller of fragment F. (Notice that $C(F)$ must be equal to $C(i)$ for all items i in fragment F.)

2.4 Processing With Multiple Controllers.

Now that we have extended our distributed database model to include more than one controller, we must describe how transactions are processed in this environment. In this and the following section, we first consider update transactions with a known base set. In section 3 we will discuss read-only transactions, while in section 4 we will study some of the problems that arise when a transaction does not specify its base set initially.

When all of the items referenced by an update transaction T have the same controller J (i.e., $C(i) = J$ for all items $i \in B_S(\mathrm{T})$), then update T can be processed completely by controller J following algorithm $A(J)$. Since controller J has complete authority over all the items referenced by T, no other controllers have to be contacted. Thus, messages must be sent out to some or all of the modules in $M(J)$ so that they authorize (e.g., accept, grant) the update T. Then update T must be performed at all nodes that have values of the items referenced by T. Since $S(i) = S$ for all items $i \in B_S(\mathrm{T})$, update T must be performed at all nodes in S. Each of these nodes processes the "perform update" message for T and its sequencing information (e.g., timestamps, sequence numbers, hole lists) in the usual manner.

If the items referenced by T have different controllers, then the update must be coordinated with all the controllers involved. Let $J_1, J_2, \ldots J_m$ be the controllers needed for update T. Notice that the problem of performing T under the control of $J_1, J_2, \ldots J_m$ is different from the problem of performing an update transaction T' with one controller distributed over m modules. In the latter case, each of the m modules shares control of the same items with the other modules, and thus algorithms that only obtain authorization from a majority of modules can be

designed (e.g., the distributed voting algorithm). In the case of m independent controllers, each of the m controllers must authorize the update because each controller has complete control over some of the items involved in the update.

2.5 The Update Algorithm for Partitioned Data With Multiple Controllers.

Although many variations are possible, there are four basic steps that must be followed to perform update transaction T under the control of controllers $J_1, J_2, \ldots J_m$:

STEP 1. Obtain an authorization (e.g., locks, majority of OK votes) for update T from controllers $J_1, J_2, \ldots J_m$. For example, if $A(J_k)$ is the centralized locking algorithm, then we request locks (from the single node in $M(J_k)$) for items i in B_S(T) such that $C(i) = J_k$. If $A(J_k)$ is the distributed voting algorithm, we obtain values and timestamps for items i in B_S(T) such that $C(i) = J_k$, and then we obtain OK votes from a majority of the modules in $M(J_k)$. After a controller authorizes an update, in effect it has "locked" all the items that it controls referenced by update T. No other updates can reference these items until update T completes. Notice that in the distributed voting algorithm, the entries for T in the pending lists at each module act as locks. No conflicting updates can receive OK votes at these nodes while T has not been accepted.

The authorization step we have just described can be done serially, in parallel, or by nodes:

(a) SERIALLY. In step 1, we can request authorization from each controller one at a time. If a controller rejects a request, we wait and try later. To avoid deadlocks, we can order the controllers a priori and we only request update authorizations in increasing controller number. (If the base set is initially unknown, we might not be able to follow this order. Also, if we optimize the transaction processing, we might destroy the ordering. In both cases we can then have deadlocks, which are briefly discussed in section 2.6.)

(b) IN PARALLEL. Messages requesting authorization can be simultaneously sent to all m controllers. If a rejection message is received from any controller, we must try again. With this strategy deadlocks can arise and we need a mechanism for detecting them. (See section 2.6.)

(c) BY NODES. The set M defined by $M(J_1) \bigcup M(J_2) \bigcup \ldots \bigcup M(J_m)$ contains all the nodes that have modules involved in T. The update transaction T can "visit" each of the nodes in set M. At each node, all the authorization required from that node will be obtained. In this way, each node is only visited once,

reducing the number of messages transmitted. Deadlocks are also possible with this method.

STEP 2. Obtain data for update T. After having cleared update T with controller J_k (where $1 \leq k \leq m$), the values of items i in B_S(T) such that $C(i) = J_k$ can be read. Any node of the set $S(i)$ can be selected as a source of data. In some cases, the data values can be read before the update is cleared with controller J_k. For example, if $A(J_k)$ is the distributed voting algorithm, the item values and their timestamps are read before the first OK vote is received. However, if more current timestamps are encountered in the voting process, the data read may have to be discarded. Notice that the reading of the item values can also be done serially, in parallel, or by nodes. The messages for doing this can be interleaved with the messages to the controllers for step 1.

STEP 3. Compute update and perform at all nodes. Once all controllers have been "locked" and the data obtained, the new update values can be computed. The new values for the items in the write set of T (W_S(T)) must be sent to all nodes that have copies of the items. These "perform update" messages can be sent in parallel to improve response times. A "perform update" message must also include all the sequencing information issued by all the controllers. Before each item value is modified, this information must be checked. For example, if for some item i, $C(i) = J_k$, $A(J_k)$ is the MCLA-h algorithm and node x is in $S(i)$, then node x will receive the "perform update" message with a sequence number s and a hole list issued by controller J_k. Node x will not update item i until all previous "perform update" messages with sequence number issued by controller J_k, less than s, and not in the hole list have been completely processed at node x. Similarly, if $A(J_k)$ is the distributed voting algorithm, then when node x updates item i, it also updates the timestamp of the item to be the timestamp of update T issued by controller J_k.

When a single node x has database fragments F_1 and F_2 controlled by separate controllers J_1 and J_2 respectively, then node x can get a combined "perform update" message involving items in F_1 and F_2. Node x has several options as to how to handle the update. One option is to split the message into two messages: one for the items in F_1 and one for the items in F_2. The submessage for F_1 would contain all the sequencing information issued by controller J_1, and the other submessage would have the J_2 information. Then node x would process the messages independently. Thus, the fragments could be updated at different times.

Another option is for node x to process the "perform update" message as a single update message. In this case, node x would first process the J_1 information

and will delay the update until the portion corresponding to the F_1 fragment is cleared. Then node x would process the J_2 information. Only when the complete update is cleared for both fragments would node x actually update the values of all the items involved.

The first alternative is more efficient but requires somewhat more complex control in order to split the "perform update" messages and process them independently. The second alternative is simpler for the case where read-only transactions must be processed because the updates are performed as complete units at each node. (See section 3.)

STEP 4. Inform controllers $J_1, J_2, \ldots J_m$ that update T has completed. After update T has been performed, all controllers must be informed so that they can release their "locks" on the items. This step can be done in conjunction with step 3. In other words, the "perform update" messages can also serve as "release locks" messages. When a controller module is told that update T has completed, it updates its state information (e.g., hole list, lock table, pending list, timestamp table, state).

2.6 Deadlocks.

The update processing algorithm for multiple independent controllers may cause deadlocks to occur because updates compete for exclusive access to the items. One way to deal with deadlocks is to prevent them by a priori ordering the controllers and by only "locking" the controllers in that particular order. Unfortunately, this is not possible in many cases, and a deadlock detection and recovery mechanism becomes necessary. (See step 1 above). The mechanisms for deadlock elimination are well known and many papers have been written on the subject [GRAY77, MENA78]. In this thesis, we will not deal with the deadlock detection and elimination problem because we consider it to be beyond the scope of this research.

2.7 Performance.

In section 2.5 we have described how updates can be processed in a partitioned distributed database system with multiple controllers. The overall system performance will of course depend on the particular update algorithms chosen for the controllers. But other factors will probably have a greater influence. These

factors are the reference patterns of the transactions and how well the data and control partitions fit these patterns. Therefore, it is very important to partition the data and the control in a way that reduces inter-fragment or inter-controller transactions. In other words, distributed database coupling [GARC78b] must be reduced by properly distributing the data and its control.

Since the system performance heavily depends on factors which are external to the update algorithms, it is hard to obtain performance results for the partitioned data, multiple controller case. However, when we design each controller in the system, we can use the performance results we have obtained in this thesis. Thus, we can view each controller and the fragment of data it controls as a fairly independent subsystem. The operation of each of these subsystems is virtually identical to the operation of the completely duplicated database, one controller case we have studied. So by optimizing the update algorithm for each subsystem, we can improve the overall performance.

3. READ-ONLY TRANSACTIONS WITH PARTITIONED DATA AND MULTIPLE CONTROLLERS.

Up to this point, we have concentrated on update transactions in the partitioned data and control environment. In the following sections (3.1 and 3.2) we will discuss read-only transactions or queries and how they can be handled.

3.1 Consistent Queries.

In chapter 9, we discovered that when some algorithms (e.g., the centralized locking and the Ellis type algorithms) were used for updates, consistent queries could be performed at a single node with local concurrency control. If all the item values referenced by a query are available at a single node x and all items are controlled by the same centralized or Ellis controller J, then a consistent query can be performed at node x with local concurrency control only. If the distributed voting algorithm is used, then the query must be cleared with a majority of the modules of the controller J. However, if the query references item values at different nodes or items controller by different controllers, then some synchronization is needed before the query can obtain a consistent view of the database. This is true even if the centralized or the Ellis type algorithms are

used for the controllers. The synchronization is needed in order that the same set of updates be reflected on all the items read.

The synchronization can of course be provided by the general update algorithm described in section 2.5. But in some cases, the algorithm can be made more efficient. The efficiency gains depend on the update algorithm being employed by the controllers. Thus, if the distributed voting algorithm is employed by the controllers for updates, we do not expect many gains over using the update algorithm for queries. (See chapter 9.) On the other hand, if the MCLA-h algorithm is used by all controllers, then we expect the query algorithms to operate more efficiently due to the existence of sequence numbers for the updates.

In the rest of section 3.1 we illustrate how an efficient query algorithm can be designed for the case of a system where all controllers use the MCLA-h algorithm. This is an important and interesting case because of its simplicity and because we expect to take advantage of the available sequence numbers. The sequence numbers will be used to coordinate consistent queries without the intervention of the central controllers. Thus, by avoiding the controllers (which are the potential bottlenecks) and by obtaining the data directly from the nodes that have it, we expect to have a more efficient algorithm. Similar query algorithms can be designed for some of the other controller types (or even mixed types), but these will be left as an exercise for the reader. In order to clarify the presentation for the MCLA-h case, we first describe the query algorithm for the partitioned data, one controller case. Then in section 3.1.3 we generalize the algorithm to the multiple controller case.

3.1.1 Consistent Queries With Partitioned Data and a Single MCLA-h Controller.

In this section, we describe a query processing algorithm for the case of a partitioned database where there is a single MCLA-h controller, residing in a single module at the central node, in charge of updates. The reason why a query Q cannot simply read the data it needs (with local concurrency control, see chapter 9) is that query Q might need data from several nodes, and each of these nodes might have performed a different set of updates. Thus, the basic idea of the query processing algorithm is to collect all the data from whatever nodes have it, and at the same time, make sure that exactly the same updates have been reflected on the data collected.

We now present the query algorithm. Recall that in the MCLA-h algorithm, every node x keeps a set of performed updates done-set[x]. (See appendix 1.)

Each element of this set is the sequence number of an update that has been performed at node x. (This set can be stored efficiently by combining all continuous lower sequence numbers into a single entry. In this chapter we will ignore such simplifications because they obscure the presentation.)

3.1.2 Query Algorithm A1.

STEP 1. Consistent query Q arrives at node x. Node x analyzes it and discovers that data must be requested from nodes $y_1, y_2, \ldots y_k$. Node x copies the current done-set[x] into temporary variable $P(x)$. From that instant on, node x saves all "perform update" messages received at node x that involve items in B_S(Q) (the base set of Q). (In step 4 below, we see why these messages are needed.)

STEP 2. Node x sends out the necessary "request data" messages to nodes $y_1, y_2, \ldots y_k$. A copy of $P(x)$ is appended to all these messages. When all "request data" messages are answered, node x continues processing at step 4 below.

STEP 3. This step is performed by any node y_i receiving a "request data" message from node x. Node y_i waits until all sequence numbers in $P(x)$ (from message) have been performed locally (i.e., until $P(x)$ is a subset of done-set[y_i]). Then node y_i copies done-set[y_i] into variable $P(y_i)$ and initiates the read operation using local concurrency control. Thus, all updates in $P(x)$ will be reflected in the data read at node y_i. (Unfortunately, not only updates in $P(x)$ will be reflected; updates in the larger set $P(y_i)$ will be reflected too. This is why node y_i will send $P(y_i)$ to node x.) After the requested values are read, node y_i sends the values and a copy of $P(y_i)$ back to node x.

STEP 4. When node x receives answers to all of its "request data" messages, it will have all the necessary data for Q. All updates in $P(x)$ will have been "seen" by the items read. However, some of the data may have seen other updates not in $P(x)$. Therefore, node x must make sure that all these extra updates are performed on all the data. The needed updates are the updates in $P(y_1) \bigcup P(y_2) \bigcup \ldots P(y_k)$ but which are not in $P(x)$. Call this set of missing updates θ. The updates in θ have either been saved by node x (in step 1) or have not yet arrived at node x. All the saved updates in θ are performed, and as the rest of the "perform update" messages arrive at node x, the updates are performed on the data for query Q. When all updates in θ have been performed, the data for Q is consistent and can be given to the user. (End of algorithm A1.)

Notice that if all the data requested by query Q is located at a single node,

then the A1 algorithm simplifies to reading the data at the node with local concurrency control.

3.1.3 Consistent Queries With Partitioned Data and a Multiple MCLA-h Controllers.

In this section, we describe a query processing algorithm for the case of partitioned database where there are multiple MCLA-h controllers. We assume that each node in the system performs updates as complete units (i.e., the second alternative of step 3 in section 2.5). That is, if a read is executed at a node with local concurrency control, then exactly the same set of updates will be reflected by all the items read at the node. (A slightly more complex algorithm can be designed for the case where the fragments involved in an update may be actually updated within a node at different times. We will not consider this other query algorithm here.)

The basic idea for the query algorithm is the same as for algorithm A1: The same set of updates must be seen by all items read, even if the values read reside on different nodes. The algorithm for this case is slightly more complex than algorithm A1 because (1) there is no unique sequence number ordering for all updates, and (2) not all nodes see all the "perform update" messages. Thus, our new algorithm, A2, must coordinate sequence numbers issued by different controllers. In addition, several nodes must participate in collecting recent "perform update" messages for the final synchronization. In algorithm A1, this step was done by a single node because it could catch all the relevant messages.

In the multiple controller case, a node x keeps a collection of performed update sets instead of a single set done-set[x]. Let done-set[J_i, x] be the sequence numbers, issued by controller J_i, of the updates that have been performed at node x. We illustrate how these sets are used through an example.

Suppose that node x has the values for fragments F_1, F_2, and F_3. Each of these fragments is controlled by controller J_1, J_2, and J_3 respectively. Say node x receives a "perform update" message for update transaction T, and assume that update T involves items in fragments F_1, F_2, and F_3. The "perform update" message must include three sequence numbers s_1, s_2, and s_3 and three hole list l_1, l_2, and l_3, each issued by controllers J_1, J_2, and J_3 respectively. Before performing the update T, node x checks that done-set[J_1, x] includes all sequence numbers less than s_1 but not in list l_1. Similar checks are done with done-set[J_2, x] and done-set[J_3, x]. After the three checks (and three possible delays), s_1 is added to done-set[J_1, x], s_2 is added to done-set[J_2, x], s_3 to done-set[J_3, x] and the update T

is performed. The additions to the done sets are done as a single atomic operation, and the update T is performed with local concurrency control.

Now consider what happens in the above example if node x only has fragment F_1 stored. The node will receive exactly the same "perform update" message for T as before. The difference is that node x only checks that all sequence numbers less than s_1 and not in list l_1 are in done-set[J_1, x]. After this check, s_1 is added to done-set[J_1, x] and the part of update T that involves fragment F_1 is performed at node x. Notice that node x may ignore the sequencing information for fragments F_2 and F_3 in the message because these fragments are not stored at node x. However, if node x discards this information, a query Q that reads fragment F_1 at node x and fragment F_2 at some other node y, will have no way of knowing that if Q sees T at node x, it must also see update T at node y. In other words, in order that Q reads consistent data, all nodes where Q reads must have either all performed update T or none should have performed T. Therefore, we need a special mechanism at node x so that this node can respond to query Q as follows: "OK, here is the data in F_1 you requested, but if you are also reading data from fragments F_2 or F_3, make sure that you see the update with sequence numbers s_2 (for controller J_2) and s_3 (for controller J_3) performed on these fragments."

One simple way to do this is to have node x (and all nodes) have a done-set[J_i, x] for all controllers J_i where $1 \leq i \leq m$. When node x starts performing update T, it would add sequence numbers s_2 and s_3 to done-set[J_2, x] and done-set[J_3, x] respectively, even though node x was not performing an update on data controlled by J_2 and J_3. Then node x could give the response for query Q described above. (Of course, this mechanism may be omitted if queries follow the query algorithm and lock all controllers involved in the query. But this is what we want to avoid by having a special simplified query algorithm.)

Before we describe the query algorithm for partitioned data and multiple MCLA-h controllers, let us summarize how the "perform update" message for an update T is processed at node x. Suppose that the update references data controlled by controllers J_1, J_2, ... J_k. Then the "perform update" message for update T will have k sequence numbers s_1 (of J_1), s_2 (of J_2), up to s_k (of J_k). The message also contains k hole lists. Suppose that node x has data controlled by J_1, J_2, ... J_l, where $l \leq k$. Then node x will check s_1, s_2, ... s_l (and the corresponding hole lists) against done-set[J_1, x], done-set[J_2, x], ... done-set[J_l, x] in the usual way. Once all checks are passed, the sequence numbers s_1, s_2, ... s_k are added to done-set[J_1, x], done-set[J_2, x], ... done-set[J_k, x] (as one atomic operation) and T is performed with local concurrency control.

3.1.4 Description of Query Algorithm A2.

The query algorithm works in two phases. In the first phase, query Q visits the necessary nodes reading data. Then a second phase may be necessary where Q returns to each node to process some missing updates. Suppose that query Q must read data from fragment F located at node y. Also assume that fragment F is controlled by controller J. Before visiting node y, query Q visits other nodes where it finds out that certain updates involving F must be seen if Q is to see a consistent view of the database. Thus, as Q visits nodes to read data, it collects a set of sequence numbers corresponding to updates that must be seen by F. This set, which we will call $P(J)$, is simply the union of the done-set$[J, x]$ for all nodes x visited before node y. When query Q finally arrives at node y (in the first phase), it contains a set $P(J)$ of updates that must be seen by F. Therefore, before reading fragment F at node y, query Q must wait until all the updates described by $P(J)$ are performed. Once this is done, node y reads the item values in fragment F that are requested by query Q. (Notice that Q may see other additional updates for F that were not required by $P(J)$.)

After having read fragment F at node y, query Q continues its visit to other nodes and it may find that there are other updates that must have also been seen by F for consistency. That is, done-set$[J, x]$ at some new node x may be larger than the set that was actually seen at node y. To fix this problem, Q will have to visit node y once more (in the second phase) in order to obtain the updates missed on the first visit when the data was read. However, to see the missing updates, Q cannot reread the data in F at y because this may still add more updates to the set of needed updates of other fragments. The solution is to have node y temporarily save all "perform update" messages (that involve items referenced by Q) between the time Q is first seen and the time when Q returns for the second phase visit. This way, any updates missed by Q on the first visit, will either be saved or are still to arrive at node y. Only the missing updates and no other updates will be performed on the data that was previously read by Q at node y. By following this protocol, we guarantee that all fragments will see exactly the same set of updates.

We will now present the algorithm in a more detailed way, but with fewer comments.

3.1.5 Query Algorithm A2.

STEP 1. Consistent query Q arrives at node x. Node x analyzes it and

discovers that the data must be requested from nodes $y_1, y_2, \ldots y_n$ and that controllers $J_1, J_2, \ldots J_k$ are in charge of all the items referenced by Q. (Node x itself may be one of the nodes y_i where data will be read.) State variables $P(J_1), P(J_2), \ldots P(J_k)$ are carried by Q as it visits the nodes. Variable $P(J_j)$ will represent the updates that must be seen by the items controlled by J_j. Initially, $P(J_j)$ is set to the empty set, for $1 \leq j \leq k$. Step 2 below describes the first phase processing that must be performed by the nodes $y_1, y_2, \ldots y_n$, while step 3 describes the second phase processing required when the nodes are visited for a second time. Thus, we start step 2 at node y_1.

STEP 2. Phase One. We are visiting the nodes $y_1, y_2, \ldots y_n$ in order. Say we are currently at node y_i. Assume that at this node we will read data from fragments $F_1, F_2, \ldots F_l$. The controller for each fragment F_z is $C(F_z)$, and of course $C(F_z)$ is one of $J_1, J_2, \ldots J_k$ for $1 \leq z \leq l \leq k$. For each fragment F_z, we perform the following four substeps:

2A) Wait until F_z sees all updates in $P(C(F_z))$. That is, query Q waits until $P(C(F_z))$ is a subset of done-set$[C(F_z), y_i]$.

2B) We are ready to read data in F_z. Save a copy of done-set$[C(F_z), y_i]$ in save-set$[C(F_z), y_i, Q]$. At the same time, start collecting all "perform update" messages that are processed at y_i in save-message$[C(F_z), y_i, Q]$.

2C) Update "P" variables. For all j such that $1 \leq j \leq k$ do "$P(J_j) := P(J_j) \bigcup$ done-set$[J_j, y_i]$".

2D) Read data in fragment F_z that is requested by query Q.

Notice that steps 2B and 2C are performed as an atomic operation. The read in step 2D is executed with local concurrency control so that only updates that were registered in step 2C are seen by Q. After phase one processing (i.e., step 2) is completed at node y_n, the second phase is started at y_n.

STEP 3. Phase Two. At the completion of phase one, $P(J_j)$ represents the updates that must and will be seen by the items controlled by controller J_j (for $1 \leq j \leq k$). The updates in $P(J_j)$ will not change in the second phase. In the second phase, we visit nodes $y_1, y_2, \ldots y_n$ (in any order) to perform any missed updates. Say we are at node y_i. Assume that at this node we read data from fragments $F_1, F_2, \ldots F_l$ in phase one. Let $C(F_z)$ be the controller of fragment F_z. For each fragment F_z ($1 \leq z \leq l$), we perform the following two substeps:

3A) The sequence numbers in $P(C(F_z))$ but not in save-set$[C(F_z), y_i, Q]$ are the sequence numbers (issued by $C(F_z)$) of the updates that are missing for fragment F_z. These updates are either in save-message$[C(F_z), y_i, Q]$ or have not arrived at node y_i. The missing updates that are in save-message$[C(F_z), y_i, Q]$ are immediately performed on the data read by Q and no more messages

are collected in save-message[$C(F_x)$, y_i, Q].

3B) Query Q waits until the rest of the missing updates arrive. As they arrive, they are performed on the data that was originally read by Q.

STEP 4. After the second phase completes at all nodes, query Q can report the values obtained to the user. (End of algorithm A2.)

3.1.6 Performance of the Consistent Query Algorithms (A1 and A2).

At first sight, algorithms A1 and A2 may seem somewhat complex, so it is natural to ask the question "Is it not better to use the update algorithm of section 2.5 (which locks controllers) for queries instead of either query algorithm A1 or A2?" The answer to this question is not simple because in some cases it might indeed be better to use the update algorithm.

The advantages of the query algorithms (A1 and A2) over the update algorithm used for queries are that (1) There is no need to visit (i.e., send any messages to) any controllers, and (2) There is no need to lock the controllers for the duration of the queries (and thus slow updates down). The potential disadvantages of the query algorithms are that (1) Nodes where data is read must be visited twice (This is only true in algorithm A2.) (2) There may be delays as we wait for "perform update" messages to arrive, and (3) There is a need for an additional protocol (i.e., the query algorithm itself) that will make the system more complex.

Clearly, if the system is processing many transactions and the central controllers are heavily loaded, it would be best to avoid the controllers. If the controllers do not have heavy loads, then the query algorithms may not be so attractive. For example, consider the case where node x has the central controller modules for 10 fragments $F_1, F_2, \ldots F_{10}$. Each of these fragments is located in nodes $y_1, y_2, \ldots y_{10}$. To process a query Q that references all 10 fragments, we could use the update algorithm. This would involve requesting locks (one message to node x), sending read requests to nodes $y_1, y_2, \ldots y_{10}$ (10 messages in parallel), collecting the data (10 more messages), and releasing the locks (1 message to node x). If instead we use algorithm A2, we would visit nodes $y_1, y_2, \ldots y_{10}$ twice (20 messages) which would take longer because these visits must be done serially. (Some parallelism is possible in the second phase, but this feature was not included in algorithm A2.)

However, there are many cases where the query algorithms perform better, even if the controllers are not heavily loaded. For example, if the controllers of the previous example are located in 10 different nodes $x_1, x_2, \ldots x_{10}$ and if we

use the update algorithm for query Q, then we would first need to visit 10 nodes requesting locks, and after the reads, we would have to release the locks at all these nodes. It is not clear what strategy will give the lowest response time for Q in this case, but the number of messages and service requests at nodes is clearly larger if we use the update algorithm. Also notice that as the number of nodes where data must be read goes down, the A2 algorithm becomes more attractive. For example, if Q reads data from two nodes, y_1 and y_2, query Q will visit node y_2 only once and node y_1 twice. The number of message transmissions needed is only four (counting the messages to start the processing and the message with the results for the user.)

Finally notice that disadvantage "2" (given at the beginning of this subsection) of the A1 and A2 query algorithms is not a serious problem. The delays waiting for "perform update" messages should not be significantly greater than the delays waiting for locks in the update algorithm.

3.2 Current Queries.

Processing current queries in the partitioned data multiple controller case is simple because no coordination between the different controllers (and their fragments) is needed. If a query makes sure that the data it reads from each fragment is current, then the collection of all the data obtained will also be current. Therefore, a current query can use the current query algorithms of the one controller case independently for each fragment and then simply combine the results obtained. (See chapter 9.)

4 THE OTHER ASSUMPTIONS.

In the second part of this thesis (chapters 7 through 11) we have considered the effects of the four major assumptions that were made for the performance analysis of chapter 4. In chapter 7 we studied failures; in chapter 9 we looked at read-only transactions; in chapter 10 we discussed transactions that do not specify their base set initially; and finally in this chapter we have studied partitioned data with multiple controllers. However, we have not yet studied a distributed database system where all four assumptions are eliminated at once. For example, in this chapter we have assumed that no failures occur and that

transactions specify their base set initially. And in chapter 7 we studied failures in the completely replicated data case.

In this section we will attempt to convince the reader that the ideas we have presented so far can be extended to a general system with none of the restrictions we have mentioned. Such a general system is quite complex. Demonstrating in detail how our algorithms can be combined and extended to the general case, and analyzing the performance of such a general system is a very hard task. Here we will not do this; we will only outline some of the principal ideas. Much research is still required in this area.

We will organize section 4 as follows. In subsection 4.1 we discuss how transactions (including queries) that do not specify their base set initially can be processed in the partitioned data, one controller case. In subsection 4.2 we study transactions (including queries) that do not specify their base set beforehand in a partitioned data, multiple controller environment. Then, in subsection 4.3, we discuss how the partitioned data, multiple controller algorithms for transactions (including queries and transactions that do not specify their base set initially) can be made resilient.

4.1 Transactions With Initially Unknown Base Sets in the Partitioned Data, One Controller Case.

As long as there is a single controller, the fact that the data may be partitioned and transactions do not specify their base set initially does not introduce any new problems.

In the algorithms where locks are used (i.e., MCLA-h and Ellis type) we still have the same three alternatives for processing transactions that do not specify their base set. (See chapter 10.) The only difference is that the "read without locks" strategy may not be as attractive now because the vulnerable read without locks period may be longer. In other words, if data has to be read at several nodes, this will take a longer time and the values read will be vulnerable to conflicts with other updates over this longer period. This increases the probability of rejection and hurts the performance of this strategy.

When the distributed voting algorithm is used, transactions that do not specify their base set at their inception can still be performed with partitioned data as described in section 1.2.2.

Queries that do not specify their base set initially can be processed as usual in the partitioned data one controller environment. For example, algorithm A1

(section 3.1.2) can be used when the controller is a MCLA-h one. The fact that the nodes where the query will read (i.e., $y_1, y_2, \ldots y_n$ in A1) are initially unknown does not affect this algorithm.

4.2 Transactions With Initially Unknown Base Sets in the Partitioned Data, Multiple Controller Case.

The update algorithm for the multiple controller case (section 2.5) can be used with a few minor modifications when transactions do not specify their base sets initially. The main difference is that we will be unable to "lock" all controllers in parallel and that deadlocks cannot be avoided by ordering the controllers.

In this environment, an update transaction T proceeds as follows. Update T first decides that it wants to read some items in fragment F_i which is controlled by controller J_i. If J_i works with locks (e.g., $A(J_i)$ = MCLA-h) and update T knows beforehand what items in F_i it needs, then T can request the locks and then read. If J_i uses locks but J_i does not know initially all the items it wants to read, then T uses any of the strategies discussed in chapter 10 (e.g., read without locks). If controller J_i does not use a locking algorithm, then update T simply reads the items and then obtains authorization from controller J_i.

After having read the data in fragment F_i, update T decides what other fragment it wishes to read. The process is then repeated in exactly the same fashion. (The lock or authorization form controller J_i is not released until the end of the transaction.) Notice that after having read items in F_i, update T might decide to read some more items in the same fragment. Thus, T might "visit" controller J_i more than once. This should not represent any serious problem, as long as T does not request the same items twice or as long as controllers can identify such occurrences.

After having read all the data it needs, update T can perform the update and inform all controllers involved of its completion in the usual fashion. As was mentioned earlier, there is a danger of deadlocks and a special mechanism is needed to detect them and recover from them.

The query algorithms for the multiple controller case need some minor modifications to deal with unpredictable reads. Queries in this environment must now carry with them enough synchronization information to cover a read of any data. For example, in algorithm A2 (section 3.1.5), we will need a $P(J_i)$ variable in each query for all possible controllers J_i, even though some of these controllers will not be involved in the query at all. Other than this, algorithm A2 remains

unchanged.

4.3 Crash Recovery.

In chapter 7 we presented some techniques for making the update algorithms resilient. We believe that the same techniques (e.g., logs, two phase commit protocol) can be extended to the more general case we are considering here.

Handling read-only transactions (queries) in a failure environment is relatively simple because these transactions cannot in any way alter the database. One simple way to process queries is to restart them from scratch whenever a failure is detected. Notice that queries can be interrupted and aborted at any point, so there is no problem with leaving a query unfinished.

The fact that some transactions do not specify their base set initially does not affect the recovery protocols. Even if transactions read data without locks or request locks as they need them, they can still use a two phase commit protocol to actually perform the updates and the transaction can still be cancelled before updates are committed.

Data partitioning with a single controller should not introduce any new problems because the one controller can be made resilient exactly as in the simple case we have already considered in chapter 7. When we have multiple controllers, each one must be made resilient. Of course, some coordination between controllers is be required in the face of a failure, but the same basic techniques that we have discussed can be applied to each controller. For example, a MCLA-h controller in a multiple controller system can still attempt to reclaim its locks after a transaction has failed to release them. Similarly, when the MCLA-h controller fails, a majority of the nodes that have data fragments that were controlled by the crashed controller can elect a new controller. This new controller can collect all state information available at the nodes that elected it and can finish or cancel all transactions that were authorized by the old controller.

To illustrate what we mean, we will now briefly show how the two phase commit protocol for the MCLA-h algorithm (section 4.1 of chapter 7) and the cancelling protocol for the same algorithm (section 4.2 of chapter 7) can be modified for the partitioned data, multiple controller case. In order to avoid the serious problems that arise when a network is partitioned, we will require that a majority of nodes in $S(F)$ be active and able to communicate with each other before any transactions involving F are processed.

4.3.1 The Two Phase Commit Protocol for the Partitioned Data Multiple MCLA-h Controller Case.

The two phase commit protocol for update transactions in the partitioned data, multiple MCLA-h controller case is very similar to the protocol that was used in the complete replication environment. The main difference is that the node that is coordinating the update transaction (called the master node) waits for acknowledgments to the "intend to perform" messages from a majority of nodes in each $S(F)$ set, for each fragment F referenced by the transaction. After these acknowledgments arrive, the master node can send out the "commit" messages.

We now give an example to show how this works. Suppose that an item i is duplicated at nodes x_1 and x_2, while item j is replicated at nodes x_2, x_3 and x_4. That is, $S(i) = \{x_1, x_2\}$ and $S(j) = \{x_2, x_3, x_4\}$. Suppose that the MCLA-h controller for item i, $C(i)$, is at node x_2 and that the MCLA-h controller for item j, $C(j)$, is located at node x_4. A transaction T wishes to read item i and then update item j. As was discussed in section 2, T must visit controllers $C(i)$ (at node x_2) and $C(j)$ (at node x_4) and request locks for those items. At $C(i)$ and at $C(j)$, transaction T obtains pairs of sequence, version numbers. These numbers are appended to the messages generated by T. After obtaining locks at both controllers, T has exclusive access to the two items and can proceed.

Once T has computed the new value for item j, the system performs the update and releases the locks using the two phase commit protocol. In this protocol, the master (which can be any node) sends out "intend to perform" messages informing all nodes involved in T (i.e., x_1, x_2, x_3, x_4) that T has completed. The master only has to wait for a majority of acknowledgments from each $S(i)$ set involved. For example, if the master gets acknowledgments from nodes x_1, x_2 and x_3, then it can send out the "commit" messages because a majority of nodes in each set $S(i)$, $S(j)$ have responded. The time when the required acknowledgments are received by the master node is called the commit point. When a node receives a "commit" message, it updates item j if it has a copy of the item. If the node has a controller involved in T, then the commit message also causes the locks to be released. Notice that no acknowledgment is necessary for the commit message.

Due to failures, some nodes that participated in T may not find out about T's completion (e.g., node x_4). These nodes will eventually discover that they missed this information because the sequence number mechanism. (See chapter 7.) When a node discovers this, it obtains the missing information from other nodes. If the information cannot be found, the node attempts to cancel T. (See section 4.3.2.)

The two phase commit protocol guarantees that either no values are stored at all or that all values produced by T are eventually stored at all nodes involved. When a node in $S(F)$ acknowledges receipt of the "intend to perform" message for T, it makes a commitment to remember T (and the values it produced) and to do everything in its power to see that T completes correctly. The node remembers T by placing the information in the prepare message in a "prepare" list. We assume that the information in this list cannot be lost. (Log entries can be made to make the prepare list safe. See chapter 7.)

When the master node for T receives a majority of acknowledgments from the nodes in $S(F)$, it knows that the update to F cannot be lost. In the case of failures, we know that at least one member of any working majority of nodes in $S(F)$ will have a record of T and will "speak up" for T. Thus, after receiving a majority of acknowledgments from the nodes in each S set involved in T, the master node can send out the commit messages. When a node in $S(F)$ receives a "commit" message, it adds T's sequence and version numbers to its list of performed transactions (which is kept by all nodes); it writes out a log entry; it performs the update on F indicated by T; and finally it removes T from the prepare list.

Due to failures, a transaction may be unable to get the majority of acknowledgments needed for committing. In such a case, the transaction "times out" and the system attempts to cancel the transaction. This cancelling protocol is described in the next section.

4.3.2 The Cancelling Protocol for the Partitioned Data Multiple MCLA-h Controller Case.

A transaction will only be cancelled if no data has been committed by the transaction. Thus, the first step in the cancelling protocol is to verify that the transaction had not reached the commit point. Notice that if a transaction T has reached the commit point, then a majority of nodes in each $S(F)$ set, for each fragment F referenced, have a record of T. Hence, if a single fragment F can be found where a majority of nodes in $S(F)$ have no record of T, then T can be cancelled.

To cancel a transaction T we proceed as follows. First, a node w is selected to be the master node for the cancellation. Any node can be the master, and several such nodes may be attempting to cancel T concurrently. We assume that node w knows that T referenced fragments $F_1, F_2, \ldots F_k$. (The protocol can easily be modified to handle the case where only one fragment is known initially.) Node

w sends out messages to controllers $C(F_1)$, $C(F_2), \ldots C(F_k)$ asking them if they can cancel T. Each controller responds either that T can be cancelled or that it does not know if T can be cancelled. Controllers do not take any action on T at this point. However, if a controller says that T can be cancelled, it makes sure that T can not reach the commit point in the future.

When node w receives answers from all controllers, it decides if T will be cancelled. If at least one controller said that T could be cancelled, then T has not committed and is cancelled. If all controllers say that they do not know if T can be cancelled, then T may have committed and node w attempts to complete T. (Notice that in this case all controllers found a record of T. Thus, all the update values produced by T are known and T can be completed.) The decision of node w is broadcast to all controllers, which then carry out the decision.

When a controller $C(F)$ wishes to know if T can be cancelled (in response to node w's first message), $C(F)$ sends out "propose to cancel T" messages to all nodes in $S(F)$. When a node y in $S(F)$ receives the "propose to cancel T" message, it checks to see if it has a record for T. That is, node y checks if it has previously received an "intend to perform" or a "commit" message for T. If y has such a record, it informs the controller. If y has no record of T, then it sends a "have seen proposal to cancel T" message to $C(F)$. With that message, node y makes a commitment not to acknowledge any "intend to perform" messages for T it might receive later. Thus, node y remembers the "propose to cancel T" message until it hears from the controller again. (We assume that node y cannot forget its commitment.)

If $C(F)$ receives a majority of "have seen proposal to cancel T" messages, then $C(F)$ knows that T has not committed and that T will not commit in the future. Thus, $C(F)$ can answer node w that T can be cancelled. On the other hand, if $C(F)$ discovered a record of T among the nodes in $S(F)$, then it must answer that it does not know if T can be cancelled because as far as it knows, T could have committed. In this case, T's record (including its update values) is sent to w.

When controller $C(F)$ receives a command from node w to actually cancel T, it does this using a two phase commit protocol similar to the one used by transactions to commit. This guarantees either that T is cancelled at all nodes in $S(F)$ (as far as F is concerned) or that T is not cancelled at all. A node in $S(F)$ finally cancels T by recording a null or dummy update. (See chapter 7.) That is, T is processed as if T has committed, except that no values are stored in the database. Similarly, a command from w to complete T because it could not be cancelled causes $C(F)$ to distribute the update values for T to nodes in $S(F)$ and

to commit them using a two phase commit protocol.

This concludes chapter 11. As we have indicated, some of the ideas presented in this chapter require additional research. In the next (and last) chapter, we will identify these areas for further research.

CHAPTER 12

CONCLUSIONS

In this thesis we have studied the performance of update algorithms for replicated data in a distributed database. We started by presenting a set of update algorithms for replicated data. Then we analyzed the performance of these algorithms using a simple performance model. In the second part of the thesis, we looked at the assumptions that were made in the performance analysis. We studied how these assumptions could be eliminated and how this affected the performance of the update algorithms.

At the end of a thesis like this one, we would like to be able to conclude that a certain algorithm was the "best" among the ones that we studied. And it would be even better to conclude that one of the new update algorithms we presented here (e.g., the MCLA algorithm) was the "best". Unfortunately, we are unable to do so. Choosing a "best" algorithm would be like selecting the best programming language or the best data model. With so many factors and issues to be considered, it is simply not possible to choose an absolute best. However, from a certain point of view, or for a particular task, one algorithm might indeed be superior to another algorithm. In this thesis, we have limited ourselves to such relative and limited comparisons.

The most general conclusion we can reach is that the centralized control algorithms can be an attractive alternative to the distributed control algorithms in many cases of interest. In most of the cases that we studied, the MCLA algorithm performed better than the other algorithms, but we must keep in mind that these results were obtained with a very simple performance model. Even though the model was simple, we tried to be realistic, and hence we believe that in a real distributed database system, the centralized control algorithms will still perform well.

The most important result of this thesis is not the conclusion of the previous paragraph. We believe that a more important contribution is that we have analyzed and shed some light on some of the issues involved in designing update algorithms for distributed databases. We have pinpointed the principal

performance parameters of distributed databases; we have presented an analysis technique for studying the update algorithms; we have identified some of the potential bottlenecks in the algorithms and we have shown ways to decrease their impact (e.g., hole lists). We have studied the problems of crash recovery and how they affect performance; we have analyzed read only transactions and we have outlined how consistent and current queries can be processed efficiently; we have studied partitioned distributed databases and the available options for transaction processing in this environment; we have introduced the concept of multiple controllers for a partitioned distributed database in order to improve performance; and we have analyzed the different strategies for processing transactions that do not specify their base set initially.

To end this thesis, we will briefly indicate some of the problems that we believe require additional research. In doing so, we will mention some of the shortcomings of the work we have presented here.

(a) The performance model presented in chapter 4 is very simple. We believe that it is possible to improve the model to reflect some additional features of distributed databases. For example, instead of assuming constant transmission times, we could model the communications network more accurately. (The model we choose for the communications network depends on the network architecture we select.) By doing this, we will be able to study some of the communication problems that were not addressed in this thesis. For example, we could assess whether the communication facility at a central node becomes a serious bottleneck. When improving the performance model, we must keep in mind that increasing the complexity of the model had its drawbacks. For example, as the number of parameters in the model grows, it becomes harder to comprehend the relationship among all the variables. Also, having a complex model might make the analysis of the algorithms extremely hard and possibly make the simulations the only reasonable tool for studying the algorithms.

(b) In this thesis we concentrated on improving the centralized control algorithms, somewhat neglecting the distributed voting algorithm. That is, it might be possible to design new distributed voting algorithms that are more efficient than the DVA algorithm we studied here. For example, a distributed voting algorithm that broadcasts vote requests in parallel to all nodes might be more efficient than the one where votes are requested one at a time. Also, it might be possible to modify the DVA algorithm so that consistent queries can be executed at a single node. (See end of section 3.2 in chapter 9.) Additional research is required in this area.

(c) Another area that needs a great amount of research is the area of crash recovery. The crash recovery algorithms presented in chapters 7 and 11 have to be specified more formally. Eventually, the correctness of the algorithms will have to be proved, but this is a hard task. In the near future, it might only be possible to do a detailed case analysis to show that all the single failure conditions possible have been considered in the crash resistant algorithms. In chapters 7 and 11 we only concentrated on the crash recovery techniques for the MCLA algorithm. It is also necessary to study crash recovery in the other update algorithms. The techniques presented in this thesis will probably be useful for the other algorithms, but some specific techniques will be applicable for each particular algorithm. Once we have made several update algorithms resilient, it would be very interesting to perform a detailed and complete performance comparison of the resilient algorithms.

(d) In this thesis we discovered that knowledge of the intended system application is needed in order to study the performance of a general distributed database system like the one of chapter 11. This knowledge is also needed to study update algorithms that take advantage of restricted transaction types. (See chapter 8.) Thus, another interesting research problem is to choose some representative distributed database applications and to study the performance of the general update algorithms (or the restricted transaction type algorithms) in these environments. Then, one could try to characterize the applications through a small set of parameters, in order to understand what algorithms perform better in what classes of applications.

(e) In this thesis, we avoided many interesting research issues in the area of distributed databases. Many of these were mentioned in chapter 1. Out of these, the ones that seem to be closely related to the update algorithms are distributed directory management, deadlock detection and elimination, optimal transaction processing, and creation and elimination of data items. Research in any one of these areas would be very valuable.

APPENDIX 1.

This appendix gives a detailed description of the MCLA-h algorithm. We describe the algorithm as a set of procedures written in a very informal Algol-like language. In this language, comments are preceded by "<<" and terminated by ">>". The objective of this appendix is to present the algorithm in a clear and simple fashion. Therefore, we will not include many modifications that can make the algorithm more efficient. These modifications are left to the reader as an exercise.

First we define the data structures used by the algorithm:

h = the limit of the hole list copies (an integer constant).
c = the node number of central node (an integer constant).
number-of-nodes = the *number of nodes in the network (an integer).*

At the central node, c:
Central-sequence-number = an integer; equal to the current sequence number.
Central-hole-set = the hole list at the central node. In this appendix, we call the hole list a "set" because it is implemented as a set. Operations on sets are described below. The elements of the central-hole-set are the sequence numbers of the holes.
Deferred-set = a set of the deferred updates at the central node due to large hole lists. Each element of this set is an update.
Locked(i) = boolean; true if item "i" is locked. There is a locked(i) for each item "i" in the database.
Lock-queue(i) = a queue of updates that are waiting for update "i" to become free. Operations on queues are described below. There is a queue for each item in the database.

At each node, n, in the system:
Done-set(n) = the set of all the updates that have been performed at this node. Each element in this set is the sequence number of a completed update.
Waiting-set(n) = the set of the updates that *have been deferred* at this node because an update with a lower sequence number is missing. Each element of this set is an update.

Each update A has the following fields:
Base-list(A) = *a pointer to the list of items referenced* by update A. Each element of this list is described below. Notice that the base-list(A) is what we have called the base set of A. However, in this appendix we call the base set a list because it is implemented as a list. The elements of base-list(A) are ordered by increasing item number in order to prevent deadlocks.
Remaining-list(A) = a pointer to the list of items that must still be locked by update A. This field is initially undefined.
Update-values(A) = the new values for the items being updated by A. These values can be stored in any convenient way.
Request-node(A) = the node number of the node where A originated.
Sequence-number(A) = the sequence number of update A.
Hole-set(A) = the copy of the hole list that is used by A. Each element in this set is a sequence number.

Each element, P, in the list of items mentioned above contains:
Item(P) = the item referenced.
Link(P) = pointer to the next element of the list. If link(P) is null, then there are no more elements.

The following functions are defined for any (FIFO) queue Q:
Add(X, Q) : adds element X to the end of queue Q. This function does not return a value.
Remove(Q) : returns the element at the head of queue Q. The element is deleted from the queue.
Is-empty(Q) : returns true if Q is empty.

The following functions are defined for any set S:
Size(S) : returns the number of elements in set S.
Insert(X, S) : adds element X to set S. No value is returned.
Delete(X, S) : removes element X from set S. If X is not in S, the function does nothing. No value is returned in any case.

Finally, the set operators "is-subset-of", "element-of", and "union" are defined in the obvious way.

We now give the procedures that describe the algorithm. Each procedure is called with two parameters: the update the procedure is going to work with and the node where the procedure is going to be executed.

```
Procedure Arrival-of-update( update A; node n );
  begin  << Update A has just arrived at node n from a user.  This
            procedure will initialize A and will request A's locks from
            the central node, c. >>
  request-node(A):= n;  remaining-list(A):= base-list(A);
  request-locks( A, c );
  end arrival-of-update;


Procedure Request-locks( update A; node c );
  begin << This procedure should only be executed at the central node c.
           This procedure will attempt to lock all of the remaining
           unlocked items in A. >>
  ptr:= remaining-list(A);
  << ptr is a local variable which points to the list of items to be
     locked. >>
  while ( ptr not null ) do
    begin  << Attempt to lock item item(ptr).  >>
    it:= item(ptr);  << save in local variable. >>
    if not locked(it) then
      locked(it):= true    << item was free so lock it >>
    else
      begin  << could not lock so we must wait >>
      remaining-list(A):= ptr;  << Save our position for later. >>
      add( A, lock-queue(it) );
      exit this procedure;
      end;
    ptr:= link( ptr );
    end;  << end of while loop >>
  << We have now obtained all lock for A's items. >>
  central-sequence-number:= central-sequence-number + 1;
  sequence-number( A ):= central-sequence-number;
  hole-set( A ):= central-hole-set;  << Copy the hole set. >>
  insert( sequence-number(A), central-hole-set );
  if size( hole-set(A) ) > h then
    insert( A, deferred-set )  << hole set too big; defer A >>
  else grant( A, request-node(A) );
  end request-locks;
```

```
Procedure Grant( update A; node n );
  begin << The locks for update A have been granted by the central
           node.  This procedure will initiate the update itself. >>
  if {1,2,3,..., sequence-number(A) - 1 } is-subset-of
      [ done-set(n) union hole-set(A) ] then
    begin << can proceed with A >>
    compute actual update values, store them in update-values(A) and
    update the local copy of the database;
    insert( sequence-number(A), done-set(n) );
    check-waiting-updates(n);
    for i:= 1 step 1 until number-of-nodes do
      if (i not n) then perform-update( A, i );
    end
  else
    insert( A, waiting-set(n) )  << must wait for other updates >>;
  end grant;


Procedure Perform-update( update A; node n );
  begin <<  This procedure will perform update A locally.  >>
  if n = c then central-update( A, c )
  else
    begin  << this is a non-central node >>
    if {1,2,3,..., sequence-number(A) - 1 } is-subset-of
       [ done-set(n) union hole-set(A) ] then
      begin  << can perform A >>
      update local database as indicated by update-values(A);
      insert( sequence-number(A), done-set(n) );
      check-waiting-updates( n );
      end
    else
      insert( A, waiting-set(n) )    << must wait for other updates >>;
    end;
  end perform-update;
```



```
Procedure Central-update( update A; node c );
  begin << This procedure should only be executed at the central node c.
           This procedure will perform update A at the central node
           and will release A's locks. >>
  update local database as indicated by update-values( A );

  << Next, we delete hole A from hole list. >>
  delete( sequence-number(A), central-hole-set );
  for B element-of deferred-set do
    begin << check if hole list of update B is now smaller than h >>
    delete( sequence-number(A), hole-set(B) );
    if size( hole-set(B) ) <= h then
      begin  << hole-set(B) has less than or equal to h elements >>
      delete( B, deferred-set );
      grant( B, request-node(B) );
      end;
    end;

  << Now release A's locks. >>
  ptr:= base-list( A );
  while (ptr not null) do
    begin  << release lock of item item(ptr) >>
    it:= item(ptr);    locked(it):= false;
    if not is-empty( lock-queue(it) ) then
      begin  << release a waiting update >>
      B:= remove( lock-queue(it) );
      << Now B can continue its locking process. >>
      locked(it):= true;
      remaining-list( B ):= link( remaining-list(B) );
      request-locks( B, c );
      end;
    ptr:= link( ptr );
    end;
  end central-update;


Procedure Check-waiting-updates( node n );
  begin  << An update has just been performed at non-central node n.
            This procedure checks if any other updates were waiting for
            the completion of the update.  >>
  for B element-of waiting-set( n ) do
    begin
    if {1,2,3,..., sequence-number( B ) - 1 } is-subset-of
       [ done-set( n ) union hole-set( B ) ] then
            perform-update( B, n );
    end;
  end check-waiting-updates;


<< End of MCLA-h algorithm and end of Appendix 1. >>
```

APPENDIX 2.

This Appendix gives a detailed description of the modified Ellis ring algorithm (MEAP). We describe the algorithm as a set of procedures written in a very informal Algol-like language. In this language, comments are preceded by "<<" and terminated by ">>".

First we define the data structures that are used by the procedures:

For each update A we define the following fields:
Base-set(A) = A pointer to the list of items referenced by update A. This list is described below. The elements of Base-set(A) are ordered by increasing item number in order to prevent deadlocks.
Remaining-set(A) = A pointer to the list of items that must still be locked by update A. Initially is undefined.
Request-node(A) = The node number of the node where A originated.
Forward-item(A) = If update A holds forward locks on an item, then forward-item(A) is the item number of that item. Notice that at most A can hold the locks of one forward item. If forward-item(A) is undefined, then A has no forward locks.

Each element, P, in the list of items mentioned above contains:
Item(P) = the item referenced.
Update-final(P) = True if this is a final update for this item.
Link(P) = Pointer to the next element in the list. If link(P) is undefined, then there are no more elements.

For each item i at node n we have:
State(i,n) = Idle, passive or active.
Lowest-priority(i,n) = Minimum priority of the set of updates that have passive locked item i at node n. This value is only defined when state(i,n) is passive.
Internal-queue(i,n) = The internal queue for item i at node n.
External-queue(i,n) = The external queue for item i at node n.

We also define the following functions:
Successor(n) : Returns the node number of the node that follows node n in the ring.
Add(A,q) : Adds update A to the end of queue q. This function does not return a value.
Remove(q) : Returns the update at the head of queue q. The update is deleted from the queue.
Is-empty(q) : Returns true if queue q is empty.

We now give the procedures that describe the algorithm. Each procedure is called with two parameters: the node where the procedure is to be executed and the update it is going to work on.

```
Procedure Arrival-of-update( update A;  node n );
  begin  << Update A has just arrived at node n from a user.  This
            procedure will initialize A and will start the locking
            process. >>
  request-node(A):= n;  forward-item(A):= undefined;
  remaining-set(A):= base-set(A);  internal-request(A,n);
  end arrival-of-update;
```



```
Procedure Internal-request( update A;  node n );
  begin  << This procedure attempts to lock all remaining items at
            A's originating node n. >>
  ptr:= remaining-set(A);  << ptr points to list of items to be locked >>
  While " ptr not undefined " do
    begin  << Attempt to lock item item(ptr) >>
    it:= item(ptr);  << save in local variable >>
    if state(it,n) = idle then state(it,n):= active
    else
      begin  << Could not lock so we must wait. >>
      remaining-set(A):= ptr;  << Save our position for later >>
      add( A, internal-queue(it,n) );  exit this procedure;
      end;
    ptr:= link(ptr);
    end;
  << We have now obtained all locks for A at this node. >>
  remaining-set(A):= base-set(A);  external-request( A, successor(n) );
  end internal-request;

Procedure External-request( update A; node n );
  begin  << This procedure attempts to lock all items referenced by A
            at node n.  Update A did not originate at node n. >>
  if request-node(A) = n then  begin
                               locks-obtained( A, n );
                               exit this procedure;
                               end;
  ptr:= remaining-set(A);
  while "ptr not undefined" do
    begin  << Attempt to lock item(ptr). >>
    it:= item(ptr);
    if state(it,n) = idle then
      begin    state(it,n):= passive;
               lowest-priority(it,n):= request-node(A);  end
    else if state(it,n) = passive then
      << Do not change state(it,n) >>
       lowest-priority(it,n):= min(lowest-priority(it,n), request-node(A));
    else if state(it,n) = active and request-node(A) < n then
      begin  <<  We must wait for item it to become available. >>
       remaining-set(A):= ptr;
       if forward-item(A) not undefined then
         release-forward-lock( A, n );    << Avoids deadlocks. >>
       add( A, external-queue(it,n) );  exit this procedure;
      end;
    << If none of the above cases, then leave state(it,n) as active. >>
    ptr:= link(ptr);
    end;
  << We have now obtained all locks for A at this node. >>
  remaining-set(A):= base-set(A);  external-request( A, successor(n) );
  end external-request;
```

```
Procedure Locks-obtained( update A; node n );
  begin  << We have obtained locks at all nodes for all of the items
            referenced by A and we are back at A's originating node n >>.
  << First we decide what items will have update-final true. >>
  ptr:= base-set(A);
  While "ptr not undefined" do
    begin
    if is-empty( external-queue( item(ptr), n ) ) then
      update-final(ptr):= true else update-final(ptr):= false;
    ptr:= link(ptr);
    end;
  Compute actual update values, store them in A, and update the
  local copy of the database;
  perform-update( A, successor(n) ); <<Initiates updates at other nodes.>>
  end locks-obtained;

Procedure Perform-update( update A; node n );
  begin  <<  This procedure performs update A at node n. >>
  if request-node(A) = n then begin
                              finish( A, n );  exit this procedure;
                              end;
  Update local database as indicated by A;
  << Now release locks if necessary. >>
  ptr:= base-set(A);
  While "ptr not undefined" do
    begin
    if update-final(ptr) = true  and
       state(item(ptr), n) = passive  and
       lowest-priority(item(ptr),n) >= request-node(A)  then
      release-internal-request(item(ptr),n);
      << Last procedure releases any requests in  the internal queue
         and sets the state to idle. >>
    ptr:= link(ptr);
    end;
  perform-update( A, successor(n) );  << Go on to next node. >>
  end perform-update;

Procedure Finish( update A; node n );
  begin  << Update A has arrived at A's originating node n after
            being performed at all nodes. We must therefore release
            all of A's locks at node n and must start up any other
            updates waiting for these items. >>
  ptr:= base-set(A);
  while "ptr not undefined" do
    begin  << Check item(ptr). >>
    it:= item(ptr);
    if is-empty( external-queue(it,n) ) then
      release-internal-request(it,n)
    else release-external-request(it, n, update-final(ptr) );
    ptr:= link(ptr);
    end;
  stop;  << Update A has been completed. >>
  end finish;
```



```
Procedure Release-internal-request( item i; node n );
  begin  << This procedure releases any locks on item i at node n and
            if there is an update waiting on i, it is started up. >>
  state(it,n):= idle;
  if not is-empty( internal-queue(i,n) then
    begin
    B:= remove( internal-queue(i,n) );
    << Update B can now lock item i and can continue. >>
    state(i,n):= active;  remaining-set(B):= link( remaining-set(B) );
    internal-request( B, n );  << Lock rest of items. >>
    end;
  end release-internal-request;

Procedure Release-external-request( item i; node n;
                                    boolean update-final );
  begin  << This procedure releases the update waiting for item i at
            node n.  We assume that external-queue(i,n) is not empty. >>
  B:= remove( external-queue(i,n) );  << Queue should now be empty. >>
  state(i,n):= passive;  lowest-priority(i,n):= request-node(B);
  if update-final then forward-item(B):= undefined
  else forward-item(B):= i;
  << Now update B can continue its locking process. >>
  remaining-set(B):= link( remaining-set(B) );
  external-request( B, n );   << Lock remaining items. >>
  end release-external-request;

Procedure Release-forward-lock( update A;  node n );
  begin  << This procedure releases the forward lock held by update A. >>
  << Forward-item(A) should not be undefined. >>
  for k:= (n + 1)  step 1  until number-of-nodes  do
    release-an-item( forward-item(A), k );
  if forward-item(A) > item( remaining-set(A) ) then
    release-an-item( forward-item(A), n );
  << Remaining-set(A) points to the the item that A will wait on at
     node n. >>
  end release-forward-lock;

Procedure Release-an-item( item i;  node n );
  begin  << This procedure will release item i at node n.  Update A
            has "inherited" these locks from an update with higher
            priority. >>
  << Assert:  state(i,n) = passive  and
              lowest-priority(i,n) > request-node(A).
   Question for the reader: Why should the above assertion be true?  >>
  release-internal-request( i, n );
  end release-an-item;

<< End of the modified Ellis ring algorithm (with sequential updates)
   and end of Appendix 2. >>
```

APPENDIX 3.

```
begin "program"


require "{} {}" delimiters;
define crlf = {('15&'12)};
define cr = {('15)};
define $ = {comment};

$  This SAIL program computes the average response time of an update
   in the MCLA centralized locking algorithm.  For an explanation of this
   program, see chapter 4.



external integer !skip!;
integer bk;  string rep;
integer j;
real temp,N,M,Ar,Bs,Is,Id,T,lambda,EY,EY2,EZ,EZ2;
real Pw,locktime,restime,oldrestime,Lnc,Lc;
real rate,Xc,Xc2,roo,Wc,Xnc,Xnc2,Wnc,Rnc,Rc;

procedure SolveSystem;
  begin "compute"
  rate← (2*N + 1) + Pw*N;
  Xc← N * 2 * Is * EY;
  Xc← Xc + Id*EY;
  Xc← Xc + N * (Is*EY + Id*EZ);
  Xc← Xc + Pw * N * Is *( EY - 1 );
  Xc← Xc / rate;
  Xc2← N * 4 * Is*Is * EY2;
  Xc2← Xc2 + Id*Id * EY2;
  Xc2← Xc2 + N * ( Is*Is*EY2 + Is*Id*(EY + EY2) + Id*Id*EZ2 );
  Xc2← Xc2 + Pw * N * 4*Is*Is*( EY2/3 - EY/2 + 1.0/6.0 );
  Xc2← Xc2 / rate;
  roo← lambda * rate * Xc;
  Wc← ( ( lambda * rate /2 ) * Xc2 )/( 1 - roo );
  rate← N + 1;
  Xnc← ( Id*EY + N * Id*EZ )/rate;
  Xnc2← ( Id*Id*EY2 + N * Id*Id*EZ2 )/rate;
  roo← lambda * rate * Xnc;
  Wnc← ( ( lambda * rate /2 ) * Xnc2 )/( 1 - roo );
  Rnc← 2 * Wnc + Wc + 2*Is*EY + Id*(EY + EZ) + 2*T;
  Rc← 3 * Wc + 3 * Is * EY + Id*( EY + EZ );
  restime← ( (N - 1)*Rnc  + Rc )/N  +  Pw*(locktime/2 + Wc + Is*(EY-1));
  end "compute";


procedure conflict!analysis;
  begin "conflict!analysis"
  Lnc← T + Wnc + Id*EY + T + Wc + Is*EY + Id*EZ;
  Lc←  Wc + Id*EY + Wc + Is*EY + Id*EZ;
  locktime← ( (N-1)* Lnc  +  Lc )/N;
  Pw← ( EY * EY / M ) * N * lambda * locktime;
  end "conflict!analysis";
```



```
while true do
  begin "main"
  print("number of nodes    N = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then N← temp else print(N,crlf);
  print("number of items    M = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then M← temp else print(M,crlf);
  print("interarrival time  Ar = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Ar←temp else print(Ar,crlf);
  print("mean base set     Bs = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Bs←temp else print(Bs,crlf);
  print("IO slice          Is = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Is←temp else print(Is,crlf);
  print("IO data           Id = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Id←temp else print(Id,crlf);
  print("transmission time  T = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then T← temp else print(T,crlf);

  lambda← 1/Ar;    EY← 1/(1 - exp( -1/Bs ));
  EY2←  EY * EY * (1 + exp( -1/Bs ));
  EZ←  ( EY + 1.0 )/2.0;
  EZ2←  EY2/3 + EY/2 + 1.0/6.0;

  Pw← 0;  j← 0;  locktime← 0;  oldrestime← 0;
  SolveSystem;  $ Result is restime, Wc, Wnc;
  do begin
     j← j + 1;
     conflict!analysis;  $ Result is Pw, locktime;
     oldrestime← restime;
     SolveSystem;  $ Result is restime, Wc, Wnc;
     end
  until ( j > 5 ) or ((restime - oldrestime)/restime < .01 );
  print(crlf,"==> mean response time = ", restime,
            " iterations = ", j, crlf, crlf);
  end "main"
end "program"
```

APPENDIX 4.

```
begin "program"

require "{} {}" delimiters;
define crlf = {('15&'12)};
define cr = {('15)};
define $ = {comment};

$  This SAIL program computes the average response time of an update
   in the distributed voting algorithm (DVA).  For an explanation of this
   program, see chapter 4.


external integer !skip!;
integer bk;  string rep;
boolean odd;
integer i, j, k, h, g, l, skips;
integer N, M, MAJ, count;
real temp, sum, restime, oldrestime, remtime, P, Pc, Pc1, Pc2, Psc;
real Ar, Bs, Is, Id, T, lambda, limit, Rt, EY, EY2, EZ, EZ2;
real X, X2, roo, rate, P1, P2, exec!time;

real array pt, q[0:11, 0:20];
real array W, delay, rateRRIT, rateRV[0:20];
    $  rateRRIT[i] is "arrival rate of RRIT[i]" (in chapter 4) divided
       by lambda.  Similarly, rateRV[i] is "arrival rate of RV[i]"
       divided by lambda ;


integer procedure modulo(value integer x,y);
  begin
  while x < 0 do x← x + y;
  return( x mod y );
  end;
```



```
procedure SolveSystem;
  begin "SolveSystem"
  for i← 0 step 1 until N-1 do
    begin $ compute wait times w[i] at each node;
    rate← N + rateRRIT[i] + rateRV[i];
    X← rateRRIT[i]*( Is + Id )* EY;
    X← X + rateRV[i]* Is * EY;
    X← X + N*( Is + Id )*EZ;
    X← X/rate;
    X2← rateRRIT[i]*(Is + Id)*(Is + Id)*EY2;
    X2← X2 + rateRV[i]*Is*Is*EY2;
    X2← X2 + N*(Is + Id)*(Is + Id)*EZ2;
    X2← X2/rate;
    roo← lambda* rate * X;
    W[i]← ((lambda*rate/2)*X2)/(1 - roo);
    end;
  restime← 0;
  for i← 0 step 1 until N-1 do
    begin  $ compute response times;
    sum← 3*W[i];
    for j← 1 step 1 until MAJ - 1 do
      sum← sum + W[modulo(i+j, N)];
    temp← sum + (Is + Id)*EY + MAJ*( Is*EY + T ) +
          (Is + Id)*EZ + delay[i];
    restime← restime + temp;
    end;
  restime← restime/N;
  end "SolveSystem";


procedure special!case;
  begin "special!case"
  j← modulo(i + MAJ - 1, N);  $  Update B originated at node j ;
  sum←0;
  for k← 0 step 1 until N-1 do
      sum← sum + ( W[k] + Is*EY + T);
  Psc← (Pc1 + Pc2)* lambda * sum;
  if j geq i then
    begin  $ delay only;
    delay[i]← delay[i] + Psc*(sum/2 + W[j] + Is*EY);
    rateRV[j]← rateRV[j] + Psc;
    end
  else
    begin  $  dr case ;
    delay[i]← delay[i] + Psc*( W[i] + (Is + Id)*EY + sum + Rt);
    rateRRIT[i]← rateRRIT[i] + Psc;
    for k← 0 step 1 until N-1 do
      rateRV[k]← rateRV[k] + Psc;
    end;
  end "special!case";
```

```
procedure compute!pt!q;
  begin "compute!pt!q"
  for i← 0 step 1 until N-1 do
    begin
    rateRRIT[i]← 1;  rateRV[i]← MAJ;  delay[i]← 0;
    end;
  for j← 0 step 1 until N-1 do
    for i← 0 step 1 until MAJ - 1 do
      begin   $ Consider updates with i OK votes at node j ;
      pt[i,j]← (MAJ - i - 1)*(Is*EY + T) + T + W[j] + (Is + Id)*EZ;
      if i+1 = MAJ then pt[i,j]← pt[i,j] - T;
      for k← j+1 step 1 until j + (MAJ- i - 1) do
         pt[i,j]← pt[i,j] + W[modulo( k, N) ];
      q[i,j]← pt[i,j]*lambda;
      end;
  end "compute!pt!q";


procedure old!ts;
  begin "old!ts"
  $  compute delays and extra loads caused by old timestamps;
  for i← 0 step 1 until N-1 do
    begin
    P← Pc * ( W[i] + (N-1)*( W[i] + (Is + Id)*EY ) )* lambda;
    temp← W[i] + (Is +Id)*EY + W[i] + Is*EY + Rt;
    delay[i]← delay[i] + P*temp;
    rateRRIT[i]← rateRRIT[i] + P;
    rateRV[i]← rateRV[i] + P;
    temp← temp + T;
    for j← 1 step 1 until N - MAJ do
      begin
      P← Pc*((N - MAJ + 1) - j)*(W[modulo(i+j-1,N)] + Is*EY + T)*lambda;
      temp← temp + W[modulo(i+j, N)] + Is*EY + T;
      delay[i]← delay[i] + P*temp;
      rateRRIT[i]← rateRRIT[i] + P;
      for k← i step 1 until i+j do
        rateRV[modulo(k,N)]← rateRV[modulo(k,N)] + P;
      end;
    end;
  end "old!ts";
```



```
procedure conflicts;
  begin "conflicts"
  for i← 0 step 1 until N-1 do
    begin "conflicts main loop"
    $ compute effect of conflict at first voting at node i;
    temp← W[i] + (Is + Id)*EY + W[i] + Is*EY + Rt;
    for h← 0 step 1 until MAJ-1 do
      begin
      $ update at i conflicts with update from node j = i-h mod N;
      j← modulo(i - h, N);
      P1← Pc1 * q[h,i];    P2← Pc2 * q[h,i];
      if j geq i then
        begin  $ update is delayed at node i;
        delay[i]← delay[i] + P*(temp + pt[h,i]/2);
        rateRRIT[i]← rateRRIT[i] + P;
        rateRV[i]← rateRV[i] + P;
        delay[i]← delay[i] + P2*( W[i] + Is*EY + pt[h,i]/2 );
        rateRV[i]← rateRV[i] + P2;
        end
      else
        begin  $ update gets DR... then proceeds to get more DRs;
        remtime← pt[h,i]/2;  k← i;  exec!time← temp - Rt;  skips← 0;
        rateRRIT[i]← rateRRIT[i] + P1;
        rateRV[i]← rateRV[i] + P1;
        while remtime > 0 do
          begin
          k← modulo(k + 1, N);    skips← skips + 1;
          exec!time← exec!time + T + W[k] + Is*EY;
          rateRV[k]← rateRV[k] + P1;
          remtime← remtime - (T + W[k] + Is*EY);
          end;
        delay[i]← delay[i] + P1*( exec!time + T + Rt );
        delay[i]← delay[i] + P2*( exec!time - W[i] - (Is + Id)*EY );
        for l← (i+MAJ-1)+1 step 1 until (i+MAJ-1)+skips do
          rateRV[modulo(l,N)]← rateRV[modulo(l,N)] + P2;
        end;
      end;
```

```
  $ now compute effects of conflicts at nodes j+1, j+2, etc. ;
  temp← W[i] + (Is + Id)*EY + W[i] + Is*EY;
  if odd then limit← MAJ -1 else limit← MAJ - 2;
  for g← 1 step 1 until limit do
    begin
    j← modulo( i + g , N);
    $ request from node i conflicts with pending request at node j.
      The conflicting request originated at node j;
    P1← Pc1 * q[0,j];     P2← Pc2 * q[0,j];
    temp← temp + T + Is*EY + W[j];  $ Temp is exec. time up to node j;
    rateRRIT[i]← rateRRIT[i] + P1;
    for k← i step 1 until i + g do
      rateRV[ modulo(k,N)]← rateRV[modulo(k,N)] + P1;
    if j geq i then
      begin $ update is delayed ;
      delay[i]← delay[i] + P1*(temp + T + Rt + pt[0,j]/2);
      delay[i]← delay[i] + P2*(W[j] + Is*EY + pt[0,j]/2);
      rateRV[j]← rateRV[j] + P2;
      end
    else
      begin  $ Update is DR ;
      remtime← pt[0,j]/2;  k← j;  exec!time← temp;  skips← 0;
      while remtime > 0 do
        begin
        k← modulo(k + 1, N);     skips← skips + 1;
        exec!time← exec!time + T + Is*EY + W[k];
        rateRV[k]← rateRV[k] + P1;
        remtime← remtime - (T + Is*EY + W[k]);
        end;
      delay[i]← delay[i] + P1*(exec!time + T + Rt);
      delay[i]← delay[i] + P2*(Exec!time - temp + W[j] - W[k]);
      for l← (i+MAJ-1)+1 step 1 until (i+MAJ-1)+skips do
        rateRV[modulo(l,N)]← rateRV[modulo(l,N)] + P2;
      end;
    end;
  if not odd then special!case;
  end "conflicts main loop"
end "conflicts";
```



```
while true do
  begin "main"
  print("number of nodes    N = "); rep←intty; k← intscan(rep,bk);
  if !skip! = cr then N← k else print(N,crlf);
  print("number of items    M = "); rep←intty; k← intscan(rep,bk);
  if !skip! = cr then M← k else print(M,crlf);
  print("interarrival time  Ar = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Ar←temp else print(Ar,crlf);
  print("mean base set      Bs = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Bs←temp else print(Bs,crlf);
  print("IO slice           Is = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Is←temp else print(Is,crlf);
  print("IO data            Id = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Id←temp else print(Id,crlf);
  print("transmission time  T = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then T← temp else print(T,crlf);
  print("retry time         Rt = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Rt←temp else print(Rt, crlf);

  lambda← 1/Ar;    EY← 1/(1 - exp( -1/Bs ));
  EY2← EY * EY * (1 + exp( -1/Bs ));
  EZ← (EY + 1.0)/2.0;
  EZ2← EY2/3.0 + EY/2.0 + 1.0/6.0;
  Pc← EY * EZ / M;
  Pc1← EZ * EY / M;    Pc2← EZ*(EY - EZ)/M;
  k← n/2;
  if k*2 = n then odd← false else odd← true;
  MAJ← ( n/2.0  + 1.0 );
  for i← 0 step 1 until n-1 do
    begin
    rateRRIT[i]← 1;  rateRV[i]← MAJ;  delay[i]← 0;
    end;
  SolveSystem;
  count← 0;
  do begin
     compute!pt!q;
     old!ts;
     conflicts;
     count← count + 1;  oldrestime← restime;
     SolveSystem;
     end
  until (count > 5) or ((restime - oldrestime)/restime < .01 );

  print(crlf,"==> mean response time = ", restime,
        " iterations = ", count, crlf, crlf);
  end "main";
end "program"
```

APPENDIX 5.

```
begin "program"


require "{} {}" delimiters;
define crlf = {('15&'12)};
define cr = {('15)};
define $ = {comment};

$  This SAIL program computes the average response time of an update
   in the MCLA centralized locking algorithm.  This program is a
   modified version of the program in appendix 3.
   The correspondance between the program variables and the names
   used in the chapter 5 is as follows:  L is E[L], Lc is E[Lc],
   Lnc is E[Lnc], Lgc is E[L|C], Lcgc is E[Lc|C], Lncgc is E[Lnc|C],
   EY is E[Y], EY2 is E[Y*Y], EYgc is E[Y|C], EY2gc is E[Y*Y|C],
   EZ is E[Z], EZ2 is E[Z*Z], EZgc is E[Z|C], EYrem is E[REM|C],
   EY2rem is E[REM*REM|C], Pw is P(W), Pw1 is P(W1), Pw2 is P(W2),
   P2w is P(2nd wait), and Pc is P(C);



external integer !skip!;
integer bk;  string rep;
integer j;
real temp,N,M,Ar,Bs,Is,Id,T,lambda,EY,EY2,EZ,EZ2;
real Pw,L,restime,oldrestime,Lnc,Lc;
real rate,Xc,Xc2,roo,Wc,Xnc,Xnc2,Wnc,Rnc,Rc;
real EYgc, EY2gc, EZgc, Lgc, Lncgc, Lcgc;;
real Pc, P2w, Pw1, Pw2, EYrem, EY2rem;
```



```
procedure SolveSystem;
  begin "compute"
  rate← (2*N + 1) + Pw*N;
  Xc← N * 2 * Is * EY;
  Xc← Xc + Id*EY;
  Xc← Xc + N * (Is*EY + Id*EZ);
  Xc← Xc + Pw * N * 2 * Is *( EYrem + P2w*(EYrem-1)/2 );
  Xc← Xc / rate;
  Xc2← N * 4 * Is*Is * EY2;
  Xc2← Xc2 + Id*Id * EY2;
  Xc2← Xc2 + N * ( Is*Is*EY2 + Is*Id*(EY + EY2) + Id*Id*EZ2 );
  Xc2← Xc2 + Pw * N * 4*Is*Is*
       (EY2rem + P2w*(EY2rem/3 - EYrem/2 + 1.0/6.0) );
  Xc2← Xc2 / rate;
  roo← lambda * rate * Xc;
  if roo geq 1 then
      begin
      print(crlf, "******* SYSTEM IS UNSTABLE *******");
      roo← lambda← 0;
      end;
  Wc← ( ( lambda * rate /2 ) * Xc2 )/( 1 - roo );
  rate← N + 1;
  Xnc← ( Id*EY + N * Id*EZ )/rate;
  Xnc2← ( Id*Id*EY2 + N * Id*Id*EZ2 )/rate;
  roo← lambda * rate * Xnc;
  if roo geq 1 then
      begin
      print(crlf, "******* SYSTEM IS UNSTABLE *******");
      roo← lambda← 0;
      end;
  Wnc← ( ( lambda * rate /2 ) * Xnc2 )/( 1 - roo );
  Rnc← 2 * Wnc + Wc + 2*Is*EY + Id*(EY + EZ) + 2*T;
  Rc← 3 * Wc + 3 * Is * EY + Id*( EY + EZ );
  restime← ( (N - 1)*Rnc  + Rc )/N  +  Pw*(Lgc/2 + Wc + Is*(EYgc-1))
     + Pw2*Lgc + Pw*P2w*(Lgc/2 + Wc + Is*(EYrem - 1));
  end "compute";


procedure conflict!analysis;
  begin "conflict!analysis"
  Lnc← T + Wnc + Id*EY + T + Wc + Is*EY + Id*EZ;
  Lc← Wc + Id*EY + Wc + Is*EY + Id*EZ;
  L← ( (N-1)*Lnc + Lc )/N;
  Lncgc← T + Wnc + Id*EYgc + T + Wc + Is*EYgc + Id*EZgc;
  Lcgc←  Wc + Id*EYgc + Wc + Is*EYgc + Id*EZgc;
  Lgc← ( (N-1)* Lncgc  +  Lcgc )/N;
  Lgc← Lgc + ((Lgc/2)*N*lambda*EY/M)*Lgc;
  Pw1← Pc * N * lambda * L;
  Pw2← Pc*(Lgc/2 + Wc + Is*(EYgc-1))*N*lambda*Pw1;
  P2w← (EYrem*EY/M)*N*lambda*L;
  Pw← Pw1 + Pw2;
  end "conflict!analysis";
```

```
while true do
  begin "main"
  print("number of nodes    N = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then N← temp else print(N,crlf);
  print("number of items    M = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then M← temp else print(M,crlf);
  print("interarrival time  Ar = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Ar←temp else print(Ar,crlf);
  print("mean base set     Bs = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Bs←temp else print(Bs,crlf);
  print("IO slice          Is = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Is←temp else print(Is,crlf);
  print("IO data           Id = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Id←temp else print(Id,crlf);
  print("transmission time  T = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then T← temp else print(T,crlf);

  lambda← 1/Ar;    EY← 1/(1 - exp( -1/Bs ));
  EY2←  EY * EY * (1 + exp( -1/Bs ));
  EZ←  ( EY + 1.0 )/2.0;
  EZ2←  EY2/3 + EY/2 + 1.0/6.0;
  EYgc← 2 * exp(-1/Bs) * EY + 1;
  EY2gc← 6*exp(-1/Bs)*EY*( exp(-1/Bs)*EY + 1)  +  1;
  EZgc← ( EYgc + 1.0 )/2.0;
  EYrem← ( EYgc - 1.0 )/2.0;
  EY2rem← EY2gc/3.0 - EYgc/2.0 + 1.0/6.0;
  Pc← EY * EY / M;

  Pw← Pw1← Pw2← P2w← 0;  j← 0;  L← Lgc← 0;  oldrestime← 0;
  SolveSystem;  $ Result is restime, Wc, Wnc;
  do begin
     print(crlf,restime);
     j← j + 1;
     conflict!analysis;  $ Result is Pw, L, Lgc;
     oldrestime← restime;
     SolveSystem;  $ Result is restime, Wc, Wnc;
     end
  until ( j > 5 ) or ((restime - oldrestime)/restime < .01 );
  print(crlf,"==> mean response time = ", restime,
        " iterations = ", j, crlf, crlf);
  end "main"
end "program"
```


APPENDIX G.

```
begin "program"

require "{} {}" delimiters;
define crlf = {('15&'12)};
define cr = {('15)};
define $ = {comment};

$  This SAIL program computes the average response time of an update
   in the distributed voting algorithm (DVA).  This program is a modified
   version of the program in appendix 4.  In this program,
   EY is E[Y], EY2 is E[Y*Y], EYgc is E[Y|C], EY2gc is E[Y*Y|C],
   EZ is E[Z], EZ2 is E[Z*Z], EZgc is E[Z|C].  Variable ptgc[i,j] is
   defined as pt[i,j] given that the update involved has conflicted.
   rateRRIT[i] is "arrival rate of RRIT[i]" in chapter 4 divided by lambda.
   Similarly, rateRV[i] is "arrival rate of RV[i]" in chapter 4 divided by
   lambda.  However, here we initialize rateRRIT[i] and rateRV[i] to
   0.  The components of these arrival rates not due to conflicts
   are handled in procedure SolveSystem;


external integer !skip!;
integer bk;  string rep;
boolean odd;
integer i, j, k, h, g, l, skips;
integer N, M, MAJ, count;
real temp, sum, restime, oldrestime, remtime, P, Pc, Pc1, Pc2, Psc;
real Ar, Bs, Is, Id, T, lambda, limit, Rt, EY, EY2, EZ, EZ2;
real X, X2, roo, rate, P1, P2, exec!time;
real EYgc, EY2gc, EZgc, sumgc;

real array pt, ptgc, q[0:11, 0:20];
real array W, delay, rateRRIT, rateRV[0:20];


integer procedure modulo(value integer x,y);
  begin
  while x < 0 do x← x + y;
  return( x mod y );
  end;
```

```
procedure SolveSystem;
  begin "SolveSystem"
  for i← 0 step 1 until N-1 do
    begin $ compute wait times w[i] at each node;
    rate← N + ( 1 +  rateRRIT[i] ) + ( MAJ + rateRV[i] );
    X← rateRRIT[i]*( Is + Id )* EYgc + 1*(Is + Id)*EY;
    X← X + rateRV[i]* Is * EYgc + MAJ* Is * EY;
    X← X + N*( Is + Id )*EZ;
    X← X/rate;
    X2← rateRRIT[i]*(Is + Id)*(Is + Id)*EY2gc + 1*(Is+Id)*(Is+Id)*EY2;
    X2← X2 + rateRV[i]*Is*Is*EY2gc + MAJ*Is*Is*EY2;
    X2← X2 + N*(Is + Id)*(Is + Id)*EZ2;
    X2← X2/rate;
    roo← lambda*rate*X;
    if roo geq 1 then
      begin
      print(crlf, "******* SYSTEM UNSTABLE *******");
      roo← lambda← 0;
      end;
    W[i]← ((lambda*rate/2)*X2)/(1 - roo);
    end;
  restime← 0;
  for i← 0 step 1 until N-1 do
    begin  $ compute response times;
    sum← 3*W[i];
    for j← 1 step 1 until MAJ - 1 do
      sum← sum + W[modulo(i+j, N)];
    temp← sum + (Is + Id)*EY + MAJ*( Is*EY + T ) +
          (Is + Id)*EZ + delay[i];
    restime← restime + temp;
    end;
  restime← restime/N;
  end "SolveSystem";


procedure special!case;
  begin "special!case"
  j← modulo(i + MAJ - 1, N);  $  Update B originated at node j ;
  sum← sumgc← 0;
  for k← 0 step 1 until N-1 do
    begin
      sum← sum + ( W[k] + Is*EY + T);
      sumgc← sumgc + ( W[k] + Is*EYgc + T);
    end;
  Psc← (Pc1 + Pc2)* lambda * sum;
  if j geq i then
    begin  $ delay only;
    delay[i]← delay[i] + Psc*(sumgc/2 + W[j] + Is*EYgc);
    rateRV[j]← rateRV[j] + Psc;
    end
  else
    begin  $  dr case ;
    delay[i]← delay[i] + Psc*( W[i] + (Is + Id)*EYgc + sumgc + Rt);
    rateRRIT[i]← rateRRIT[i] + Psc;
    for k← 0 step 1 until N-1 do
      rateRV[k]← rateRV[k] + Psc;
    end;
  end "special!case";
```



```
procedure compute!pt!q;
  begin "compute!pt!q"
  for i← 0 step 1 until N-1 do
    begin
    rateRRIT[i]← 0;  rateRV[i]← 0;  delay[i]← 0;
    end;
  for j← 0 step 1 until N-1 do
    for i← 0 step 1 until MAJ - 1 do
      begin   $ Consider updates with i OK votes at node j ;
      pt[i,j]← (MAJ - i - 1)*(Is*EY + T) + T + W[j] + (Is + Id)*EZ;
      ptgc[i,j]← (MAJ - i-1)*(Is*EYgc + T) + T + W[j] + (Is + Id)*EZgc;
      if i+1 = MAJ then
        begin pt[i,j]← pt[i,j] - T;  ptgc[i,j]← ptgc[i,j] - T; end;
      for k← j+1 step 1 until j + (MAJ- i - 1) do
        begin
          pt[i,j]← pt[i,j] + W[modulo( k, N) ];
          ptgc[i,j]← ptgc[i,j] + W[modulo( k, N) ];
        end;
      q[i,j]← pt[i,j]*lambda;
      end;
  end "compute!pt!q";


procedure old!ts;
  begin "old!ts"
  $  compute delays and extra loads caused by old timestamps;
  for i← 0 step 1 until N-1 do
    begin
    P← Pc * ( W[i] + (N-1)*( W[i] + (Is + Id)*EY ) )* lambda;
    temp← W[i] + (Is +Id)*EYgc + W[i] + Is*EYgc + Rt;
    delay[i]← delay[i] + P*temp;
    rateRRIT[i]← rateRRIT[i] + P;
    rateRV[i]← rateRV[i] + P;
    temp← temp + T;
    for j← 1 step 1 until N - MAJ do
      begin
      P← Pc*((N - MAJ + 1) - j)*(W[modulo(i+j-1,N)] + Is*EY + T)*lambda;
      temp← temp + W[modulo(i+j, N)] + Is*EYgc + T;
      delay[i]← delay[i] + P*temp;
      rateRRIT[i]← rateRRIT[i] + P;
      for k← i step 1 until i+j do
        rateRV[modulo(k,N)]← rateRV[modulo(k,N)] + P;
      end;
    end;
  end "old!ts";
```

```
procedure conflicts;
  begin "conflicts"
  for i← 0 step 1 until N-1 do
    begin "conflicts main loop"
    $ compute effect of conflict at first voting at node i;
    temp← W[i] + (Is + Id)*EYgc + W[i] + Is*EYgc + Rt;
    for h← 0 step 1 until MAJ-1 do
      begin
      $ update at i conflicts with update from node j = i-h mod N;
      j← modulo(i - h, N);
      P1← Pc1 * q[h,i];    P2← Pc2 * q[h,i];
      if j geq i then
        begin  $ update is delayed at node i;
        delay[i]← delay[i] + P*(temp + ptgc[h,i]/2);
        rateRRIT[i]← rateRRIT[i] + P;
        rateRV[i]← rateRV[i] + P;
        delay[i]← delay[i] + P2*( W[i] + Is*EYgc + ptgc[h,i]/2 );
        rateRV[i]← rateRV[i] + P2;
        end
      else
        begin  $ update gets DR... then proceeds to get more DRs;
        remtime← ptgc[h,i]/2;  k← i;  exec!time← temp - Rt;  skips← 0;
        rateRRIT[i]← rateRRIT[i] + P1;
        rateRV[i]← rateRV[i] + P1;
        while remtime > 0 do
          begin
          k← modulo(k + 1, N);    skips← skips + 1;
          exec!time← exec!time + T + W[k] + Is*EYgc;
          rateRV[k]← rateRV[k] + P1;
          remtime← remtime - (T + W[k] + Is*EYgc);
          end;
        delay[i]← delay[i] + P1*( exec!time + T + Rt );
        delay[i]← delay[i] + P2*( exec!time - W[i] - (Is + Id)*EYgc );
        for l← (i+MAJ-1)+1 step 1 until (i+MAJ-1)+skips do
          rateRV[modulo(l,N)]← rateRV[modulo(l,N)] + P2;
        end;
      end;
```



```
    $ now compute effects of conflicts at nodes j+1, j+2, etc. ;
    temp← W[i] + (Is + Id)*EYgc + W[i] + Is*EYgc;
    if odd then limit← MAJ -1 else limit← MAJ - 2;
    for g← 1 step 1 until limit do
      begin
      j← modulo( i + g , N);
      $ request from node i conflicts with pending request at node j.
        The conflicting request originated at node j;
      P1← Pc1 * q[0,j];    P2← Pc2 * q[0,j];
      temp← temp + T + Is*EYgc + W[j];  $ Temp is exec. time up to node j;
      rateRRIT[i]← rateRRIT[i] + P1;
      for k← i step 1 until i + g do
        rateRV[ modulo(k,N)]← rateRV[modulo(k,N)] + P1;
      if j geq i then
        begin $ update is delayed ;
        delay[i]← delay[i] + P1*(temp + T + Rt + ptgc[0,j]/2);
        delay[i]← delay[i] + P2*(W[j] + Is*EYgc + ptgc[0,j]/2);
        rateRV[j]← rateRV[j] + P2;
        end
      else
        begin  $ Update is DR ;
        remtime← ptgc[0,j]/2;  k← j;  exec!time← temp;  skips← 0;
        while remtime > 0 do
          begin
          k← modulo(k + 1, N);    skips← skips + 1;
          exec!time← exec!time + T + Is*EYgc + W[k];
          rateRV[k]← rateRV[k] + P1;
          remtime← remtime - (T + Is*EYgc + W[k]);
          end;
        delay[i]← delay[i] + P1*(exec!time + T + Rt);
        delay[i]← delay[i] + P2*(Exec!time - temp + W[j] - W[k]);
        for l← (i+MAJ-1)+1 step 1 until (i+MAJ-1)+skips do
          rateRV[modulo(l,N)]← rateRV[modulo(l,N)] + P2;
        end;
      end;
  if not odd then special!case;
  end "conflicts main loop"
end "conflicts";
```

```
while true do
  begin "main"
  print("number of nodes    N = "); rep←intty; n← intscan(rep,ok);
  if !skip! = cr then N← n else print(N,crlf);
  print("number of items    M = "); rep←intty; m← intscan(rep,ok);
  if !skip! = cr then M← m else print(M,crlf);
  print("interarrival time  Ar = "); rep←intty; temp←realscan(rep,ok);
  if !skip! = cr then Ar←temp else print(Ar,crlf);
  print("mean base set      Bs = "); rep←intty; temp←realscan(rep,ok);
  if !skip! = cr then Bs←temp else print(Bs,crlf);
  print("2D slice           Is = "); rep←intty; temp←realscan(rep,ok);
  if !skip! = cr then Is←temp else print(Is,crlf);
  print("2D data            Id = "); rep←intty; temp←realscan(rep,ok);
  if !skip! = cr then Id←temp else print(Id,crlf);
  print("transmission time  T = "); rep←intty; temp←realscan(rep,ok);
  if !skip! = cr then T← temp else print(T,crlf);
  print("retry time         Rt = "); rep←intty; temp←realscan(rep,ok);
  if !skip! = cr then Rt←temp else print(Rt, crlf);

  lambda← 1/Ar;     Ey← 1/(1 - exp( -1/Bs ));
  Ey2← Ey * Ey * (1 + exp( -1/Bs ));
  Ez← (Ey + 1.0)/2.0;
  Ez2← Ey2/3.0 + Ey/2.0 + 1.0/6.0;
  Eygc← 2 * exp(-1/Bs) * Ey + 1;
  Eygc← G*exp(-1/Bs)*Ey*( exp(-1/Bs)*Ey + 1)  +  1;
  Ezgc← ( Eygc + 1.0 )/2.0;
  Pc← Pc1← Ey*Ez/M;
  Pc2← Ez*(Ey - Ez)/M;
  k← n/2;
  if k*2 = n then odd← false else odd← true;
  MAX← ( n/2.0  + 1.0 );
  for i← 0 step 1 until n-1 do
    begin
    rateRRIT[i]← 0;  rateRW[i]← 0;  delay[i]← 0;
    end;
  SolveSystem;
  count← 0;
  do begin
     print(crlf, resttime);
     computeptlq;
     oldtb;
     conflicts;
     count← count + 1;  oldresttime← resttime;
     SolveSystem;
     end
  until (count > 5) or ((resttime - oldresttime)/resttime < .01 );

  print(crlf,"==> mean response time = ", resttime,
        " iterations = ", count, crlf, crlf);
  end "main";
end "program"
```


APPENDIX 7.

In this appendix we compare two alternatives for handling hole lists when they exceed the limit h. The first alternative is to delay updates at the central node until their hole lists shrink to a size less than or equal to the hole size limit h. This alternative is used in the MCLA-h algorithm and we will call it the "delay at central node" strategy. The second alternative is to truncate the hole list at the central node and to send the update "grant" message immediately to the update's originating node. At that node the update may be delayed until updates that were truncated from the hole list are performed. We will call this second alternative the "truncating" strategy.

Consider an update A which originated at node x and whose locks have just been granted at the central node. Suppose that the hole list at the central node at that instant contains updates $B_1, B_2, B_3, \ldots B_j$. (Assume that $j > h$ or else there would be no delays.) Let $t_1, t_2, t_3, \ldots t_j$ be the times when the "perform update" messages for updates $B_1, B_2, B_3, \ldots B_j$ arrive at node x. The performance of the truncating alternative depends very much on how well the central node can predict or guess the values $t_1, t_2, t_3, \ldots t_j$. As we will see later, if the central node can indeed know the values, then the truncating strategy will be superior to the delay at central node strategy. If the central node cannot predict these values, then the truncating alternative may not be so attractive.

There are many heuristics that the central node could use to guess the times $t_1, t_2, t_3, \ldots t_j$, but it is almost impossible for us to evaluate these heuristics because they depend on the actual types of updates that are being performed. For example, the central node could predict an update's remaining execution time based on how long the update has been running. Or maybe the number of locks granted to an update B_i is an indication of the update's execution time and could thus be used for predicting the time t_i.

Since it is so hard for us to know the central node's ability to guess $t_1, t_2, t_3, \ldots t_j$, in this appendix we will simply consider two cases: In one case, the central node has perfect future knowledge and can exactly predict the times $t_1, t_2, t_3, \ldots t_j$, while in the other case, the central node has no idea what these values could be. Any real system (using a decent heuristic) will be somewhere between these two extremes, and hopefully, the results we obtain for the two special cases will be useful in choosing a strategy.

In the following discussion we use the same model that was used to study

the performance of the update algorithms in chapter 4. In particular, we assume that the network transmission time is a constant T.

A7.1 Assumptions.

To simplify the analysis, we will make some assumptions. We will assume that all the "perform update" messages for a given update arrive at all nodes at the same time. In our original model, the "perform update" message to the update's originating node arrives T seconds before the other messages because the originating node is the one that is sending the "perform update" messages. Our assumption is equivalent to saying that a message that a node sends to itself will also take T seconds to arrive. This small modification to the original model should not alter our results significantly. In the introduction to this appendix, we mentioned that t_i was the time when the "perform update" message for update B_i arrived at node x. The assumption that we have made implies that the "perform update" message for B_i arrives at *all* nodes at time t_i.

The next assumption we make is that all nodes will look at a "perform update" message for update B_i at a high CPU priority as soon as the message arrives and will release any other updates that were waiting for B_i. In a non central node, an update C could be waiting for B_i because B_i had a lower sequence number and B_i was not in C's hole list. If update C is not waiting for any other updates to be performed, then it will immediately be released, that is, C will be added to the queue of updates that are to be performed. We assume that the CPU time needed to look at the "perform update" message for B_i is negligible and hence update C will be queued for service at time t_i.

The central node also looks at a "perform update" message for update B_i as soon as it arrives and releases any waiting updates that can proceed. If update C is delayed at the central node because its hole list is too large, and the removal of B_i from the list causes it to shrink to size h, then the grant message for C will be sent immediately (e.g., at time t_i).

Our last assumption deals with the time that an update remains on the hole list at the central node. Let X_i be the time that update B_i remains on the hole list at the central node. Time X_i is the difference between the time when the "perform update" message for B_i arrives at the nodes and the time when all locks for B_i were obtained. We will assume that X_i is an exponentially distributed random variable. To justify this, we note that the main component in X_i is in computing the update values for B_i at B_i's originating node. The service time for computing these values is approximately exponential. (See chapter 4.)

The waiting time at that node will be roughly exponential because the node is approximately a M/M/1 system. Of course, the distribution of X_i is not exactly exponential because there is a constant $2T$ factor due to the two transmissions needed before B_i is removed from the hole list. However, we still choose to use the exponential distribution because (a) it is a simple distribution and (b) it is our best guess.

We also assume that random variables X_i for $1 \leq i \leq j$ are identically distributed and independent. We will let θ be the mean of the exponential distribution of X_i. The value of θ can be estimated from the analysis of chapter 4. If B_i originated at a non central node, then B_i on the average remains on the hole list for $T + \overline{W_{nc}} + I_d\,E[Y] + T$ seconds (where $\overline{W_{nc}}$ is the average IO wait time at B_i's originating node , I_d is the time to read one item from the database and $E[Y]$ is the average number of items in the base set of an update). If B_i originates at the central node, then on the average B_i remains for $\overline{W_c} + I_d\,E[Y] + T$ seconds (where $\overline{W_c}$ is the average IO wait time at the central node). Therefore we choose θ to be the weighted average

$$\theta = \frac{1}{N}\left(\overline{W_c} + I_d\,E[Y] + T\right) + \frac{N-1}{N}\left(T + \overline{W_{nc}} + I_d\,E[Y] + T\right) \qquad (1)$$

For example, using the typical values of chapter 6 (i.e., six nodes, interarrival time A_r of 6 seconds, $I_d = 0.025$ sec., $T = 0.1$ sec., etc.), we find that $\theta = 0.37$ seconds.

The analysis in chapter 4 assumed that the value of h was large enough so that no updates were unnecessarily delayed. However, the value of θ we have obtained is valid for any h (as long as the system is stable) because none of the quantities of equation (1) depend on h. The wait times $\overline{W_c}$ and $\overline{W_{nc}}$ only depend on the IO load at the nodes and these loads are independent of h. Notice that the value obtained for θ by dividing $\overline{H}$ (for large h) from figure 6.26 by $N\lambda$ is larger than the above because in the simulation, an update B_i remains on the hole list until after B_i releases its locks at the central node. Here we are assuming that B_i disappears from the hole list as soon as the "perform update" message for B_i arrives at the central node.

A7.2 The Delay at Central Node Alternative.

In this section we will compute an update's delay when the delay at central node strategy is used. Update A, which originated at node x, obtains its locks at the central node at time t_0. At that instant, updates $B_1, B_2, B_3, \ldots B_j (j > h)$ are

in the hole list. Update A must wait at the central node until its hole list shrinks in size and then a grant message can be sent to node x. Thus, on the average, update A's computations will be started at node x at time γ_{dc} :

$$\gamma_{dc} = t_0 + \delta_{dc} + T \tag{2}$$

where δ_{dc} is the average delay until $j-h$ of the "perform update" messages for the updates $B_1, B_2, B_3, \ldots B_j$ arrive at the central node. We will now compute δ_{dc}.

As stated earlier, exponentially distributed random variable X_i is the time that B_i remains on the hole list. At time t_0, update B_i has already been on the hole list for some time, but because of the memoryless property of the exponential distribution, the remaining time for update B_i given that it has remained until t_0 is also exponentially distributed with mean θ. In other words, random variables $Y_i = t_i - t_0$ $(1 \leq i \leq j)$, which are the remaining times for updates B_i on the hole list are independent identically distributed with an exponential distribution with mean θ, i.e.,

$$f_Y(y) = \frac{1}{\theta}\exp(-y/\theta) \quad \text{for } y \geq 0. \tag{3}$$

This implies that

$$\begin{aligned} \Pr[Y < y] &= 1 - \exp(-y/\theta), \\ \Pr[Y > y] &= \exp(-y/\theta). \end{aligned} \tag{4}$$

Let random variable Z be the delay until any $j-h$ of the j updates are removed from the hole list. Let the number of updates that do not fit in update A's hole list be m, that is,

$$m = j - h \tag{5}$$

We want the probability distribution function of Z so we can compute $E[Z] = \delta_{dc}$. The probability that Z is greater than z is given by

$$\Pr[Z > z] = \sum_{i=0}^{m-1} \Pr[\text{exactly } i \text{ updates were removed in } z \text{ sec.}]. \tag{5}$$

Notice that the events on the right are mutually exclusive and they represent the only ways in which z seconds could have gone by without m or more updates having been removed from the hole list. The probability of each of these events is the probability that exactly i updates are removed, times the number of ways

in which we can choose the i updates that finish out of the j total updates. Therefore,

$$\Pr[Z > z] = \sum_{i=0}^{m-1} \binom{j}{i} \Pr[Y < z]^i \Pr[Y > z]^{j-i}. \tag{7}$$

Using equation (4) and the fact that the cumulative probability distribution $F_Z(z)$ is $1 - \Pr[Z > z]$, we get

$$F_Z(z) = 1 - \sum_{i=0}^{m-1} \binom{j}{i} [1 - \exp(-z/\theta)]^i [\exp(-z/\theta)]^{j-i}. \tag{8}$$

The expected value of Z is given by

$$E[Z] = \int_0^\infty z f_Z(z)\, dz, \tag{9}$$

where

$$f_Z(z) = \frac{d}{dz} F_Z(z).$$

By integrating by parts in the above equation, one can show that

$$E[Z] = \int_0^\infty [1 - F_Z(z)]\, dz, \tag{10}$$

so substituting the value of $F_Z(z)$ found in equation (8), we find that

$$E[Z] = \int_0^\infty \sum_{i=0}^{m-1} \binom{j}{i} [\exp(z/\theta) - 1]^i \exp(-zj/\theta)\, dz. \tag{11}$$

Exchanging the integral and the sum and substituting $\exp(z/\theta) - 1$ by its binomial expansion [KNUT73, sec. 1.2.6, eq. 13],

$$E[Z] = \sum_{i=0}^{m-1} \binom{j}{i} \int_0^\infty \sum_{k=0}^{i} \binom{i}{k} \exp\left(\frac{zk}{\theta}\right) (-1)^{i-k} \exp\left(-\frac{zj}{\theta}\right) dz. \tag{12}$$

Again, exchanging the integral with the sum, we get

$$E[Z] = \sum_{i=0}^{m-1} \binom{j}{i} \sum_{k=0}^{i} \binom{i}{k} (-1)^{i-k} \int_0^\infty \exp\left(\frac{z(k-j)}{\theta}\right) dz. \tag{13}$$

Evaluating the integral,

$$E[Z] = \sum_{i=0}^{m-1} \binom{j}{i} (-1)^i \sum_{k=0}^{i} \binom{i}{k} \frac{(-1)^k \theta}{j-k}. \tag{14}$$

(Notice that $k - j < 0$.) Using [KNUT73, sec. 1.2.6, prob. 48], we simplify this to

$$E[Z] = \sum_{i=0}^{m-1} \binom{j}{i} (-1)^i \frac{\theta}{j\binom{i-j}{i}}. \tag{15}$$

Since

$$\binom{i-j}{i} = (-1)^i \binom{j-1}{i} = (-1)^i \frac{j-i}{j} \binom{j}{i} \tag{16}$$

then

$$E[Z] = \sum_{i=0}^{m-1} \frac{\theta}{j-i}. \tag{17}$$

Equation (17) can also be written as

$$\delta_{dc} = E[Z] = \theta(H_j - H_{j-m}), \tag{18}$$

where H_i is the well known sum of the first i harmonic numbers:

$$H_i = \sum_{k=1}^{i} \frac{1}{k}. \tag{19}$$

Equations (2) and (18) allow us to compute γ_{dc}, the time when the grant message for update A will arrive at node x when the delay at central node strategy is used:

$$\gamma_{dc} = t_0 + \theta(H_j - H_{j-m}) + T. \tag{20}$$

A7.3 The Truncating Alternative With No Future Knowledge.

In this section we will compute an update's delay when the central node truncates m holes in update A's hole list at random. Update A obtains its locks at the central node at time t_0. The grant message for A is immediately sent (at time t_0) and T seconds later it arrives at node x. Since some holes were eliminated, A will be delayed at x unless the updates that were truncated happen to have finished during the T seconds it took the grant message to reach x.

Let $B_1, B_2, B_3, \ldots B_m$ be the updates that were truncated, and let exponentially distributed random variables $X_1, X_2, X_3, \ldots X_m$ be the times that these updates remain on the hole list. If random variables $Y_1, Y_2, Y_3, \ldots Y_m$ are the remaining times of the updates on the hole list at time t_0, then these random variables are also exponential, independent and identically distributed with mean θ. Hence equations (3) and (4) apply for random variable Y_i.

Update A will have to wait at node x for all the truncated updates to complete. Let random variable U be the delay (starting at time t_0) until the m updates are removed from the hole list (i.e., the delay until their "perform update" messages arrive). If $U < T$ then update A will only be delayed for the T seconds needed to transmit the "grant" message from the central node to node x. On the other hand, if $U > T$, then A will be delayed U seconds. In other words, update A's computations will be started at node x on the average at time γ_{tr}:

$$\gamma_{tr} = t_0 + E[V], \tag{21}$$

where random variable V is defined by

$$V = \max(T, U). \tag{22}$$

To compute E[V], we first find the cumulative probability distribution function of U, $F_U(x)$. Random variable U can be less than a value x only if all of the m updates have been removed from the hole list in less than x seconds. That is,

$$\Pr[U < x] = \prod_{i=1}^{m} \Pr[Y_i < x]. \tag{23}$$

Using equation (4), we get

$$\Pr[U < x] = F_U(x) = [1 - \exp(-x/\theta)]^m. \tag{24}$$

Equation 24 can now be used to find the cumulative probability distribution function of V, $F_V(x)$. From equation (22), we observe that

$$F_V(x) = \begin{cases} 0, & \text{if } x < T; \\ F_U(x), & \text{if } x > T. \end{cases} \tag{25}$$

Next, we find E[V] as

$$E[V] = \int_0^\infty [1 - F_V(x)]\, dx \tag{26}$$

or

$$E[V] = T + \int_T^\infty [1 - [1 - \exp(-x/\theta)]^m]\, dx. \tag{27}$$

Using the binomial expansion of $[1 - \exp(-x/\theta)]^m$ and interchanging the integral with the sum, we get

$$E[V] = T - \sum_{k=1}^{m} \binom{m}{k} (-1)^k \int_T^\infty \exp(-xk/\theta)\, dx. \tag{28}$$

Evaluating the integral, we find that

$$E[V] = T - \sum_{k=1}^{m} \binom{m}{k} \frac{[-\exp(-T/\theta)]^k \theta}{k}. \tag{29}$$

Using [KNUT73, sec. 1.2.7, prob. 13], we simplify this to

$$E[V] = T + \theta \left\{ H_m - \sum_{k=1}^{m} \frac{[1 - \exp(-T/\theta)]^k}{k} \right\} \tag{30}$$

(where H_m is given in equation (19)). This can also be written as

$$E[V] = T + \theta \sum_{k=1}^{m} \frac{1 - [1 - \exp(-T/\theta)]^k}{k}. \tag{31}$$

Equations (21) and (31) give us γ_{tr}, the average time when processing of update A can start at node x when the truncating strategy is used and the updates

in the hole list are truncated at random:

$$\gamma_{tr} = t_0 + T + \theta \sum_{k=1}^{m} \frac{1 - [1 - \exp(-T/\theta)]^k}{k}. \tag{32}$$

If T/θ is small, equation (32) can be simplified further. If

$$\frac{(T/\theta)^k}{k} \qquad \text{for } k \geq 2$$

is negligible as compared to T/θ, then $\exp(-T/\theta)$ can be approximated by $1 - T/\theta$ and equation (32) becomes

$$\gamma_{tr} = t_0 + \theta H_m. \tag{33}$$

A7.4 Comparison.

Equations (20) and (32) can be used to compare the performance of the delay at central node with the truncating strategy when the central node has no future knowledge.

Before substituting actual values for the parameters, we can observe some general trends in the equation for γ_{dc} and γ_{tr} , the times when update A's computations can be started at node x for the delay at central node and the truncating alternatives respectively.

(a) As the transmission time T approaches 0, γ_{tr} approaches $t_0 + \theta H_m$. In this case, $\gamma_{dc} \leq \gamma_{tr}$ because $(H_j - H_{j-m}) \leq H_m$ and $j \geq m$. Therefore, for small transmission times (i.e., $\exp(-T/\theta) \ll 1$), the delay at central node strategy is superior.

(b) For $j = m$, $\gamma_{dc} = t_0 + T + \theta H_m$. Therefore, γ_{tr} will always be smaller than γ_{dc} because of the factor $[1 - \exp(-T/\theta)]^k$ in equation (32). (If $(T/\theta)^k/k$ for $k \geq 2$ is negligible as compared to T/θ , we can use equation (32) for γ_{tr}, and we find that γ_{tr} is smaller than γ_{dc} by T seconds for $j = m$.) However, as j increases while m is held constant, γ_{tr} remains at its same value while γ_{dc} starts decreasing because of the H_{j-m} factor. As j increases, γ_{dc} approaches $t_0 + T$ because H_{j-m} approaches H_j for large j. Thus, for some value of j, γ_{dc} will become smaller than γ_{tr}. This can be interpreted as follows: As the number of updates we can choose from in order to truncate the m updates that do not fit in A increases, the delay at central node alternative becomes more and more attractive.

(c) If j is held constant, then we see that both γ_{dc} and γ_{tr} decrease as m is decreased. At $m = j$ we saw that $\gamma_{tr} < \gamma_{dc}$, but this inequality may be reversed for some smaller value of m. In particular, for $m = 1$, $\gamma_{dc} = t_0 + T + \theta/j$ and $\gamma_{tr} = t_0 + T + \theta \exp(-T/\theta)$. If $j \geq \exp(T/\theta)$ (which is usually the case), we see that $\gamma_{dc} < \gamma_{tr}$ at $m = 1$. In other words, (if $j \geq \exp(T/\theta)$) as the fraction of the updates in the hole list that do not fit in A's hole list decreases, the delay at central node strategy performs better.

Table A7.1 shows the values of γ_{dc} and γ_{tr} for the typical parameter values used in chapter 6. In this case, the network transmission time T is 0.1 seconds and the mean of the hole remaining time distribution is $\theta = 0.37$ seconds (see section A7.1). For convenience we assume that $t_0 = 0$. Notice that in this case the approximation of equation (33) can be used. Also notice that only when $j = m$ (i.e., hole size limit h is 0) does the truncating alternative perform better. In all other cases, γ_{dc} is smaller and the delay at central node strategy is more efficient.

The results for this case are also shown in figure A7.1. In that figure, we plot γ_{dc} and γ_{tr} as a function of j, the number of updates in the hole list, for various values of h, the hole list size limit. Recall that $h = j - m$. In this figure we see that as h increases, the delay at central node strategy becomes more and more attractive over the truncating strategy with no future knowledge. In chapter 6, we stated that a value for h of 4 or 5 would be a good choice. For this value of h, the delay at central node strategy is definitely superior.

A7.5 The Truncating Alternative With Perfect Future Knowledge.

In this section we will compute an update's delay when the truncating strategy is used and the central node knows the times $t_1, t_2, t_3, \ldots t_j$. Recall that t_i is the time when the "perform update" message for update B_i arrives at all nodes ($1 \leq i \leq j$).

If the central node truncates update B_i from update A's hole list, then update A's computations at node x will be delayed at least until time t_i. Therefore, since the central node must truncate m updates, it should choose the ones with the smallest value of t. If the central node does this, then update A will be able to proceed at node x as soon as the first m "perform update" messages arrive. However, if this occurs before the "grant" message for A with the truncated hole list arrives at node x, then A's processing will start when the "grant" message arrives and not before. In other words, update A's computations will be started at node x on the average at time γ_{tp}:

Table A7.1

TABLE A7.1

COMPARISON OF THE TRUNCATING WITH NO KNOWLEDGE STRATEGY TO THE DELAY AT CENTRAL NODE STRATEGY FOR THE TYPICAL PARAMETER VALUES.

T = 0.1 sec., θ = 0.37 sec., t0 = 0,
N = 6, Ar = 6sec., Id = Is = 0.025 sec., M = 1000 items, Bs = 5 items.

The top entry in each box is γ_{tr}, the time when update A's computations are started at node x when the truncating with no knowledge strategy is used, while the bottom entry in each box is γ_{dc}, the time when A's computations are started when the delay at central node strategy is used.

γ_{tr}
γ_{dc}

m \ j	1	2	3	4	5	6	7
1	0.38 0.47	0.38 0.29	0.38 0.22	0.38 0.19	0.38 0.17	0.38 0.16	0.38 0.15
2		0.56 0.66	0.56 0.41	0.56 0.32	0.56 0.27	0.56 0.24	0.56 0.21
3			0.68 0.78	0.68 0.50	0.68 0.39	0.68 0.33	0.68 0.29
4				0.77 0.87	0.77 0.57	0.77 0.45	0.77 0.38
5					0.84 0.94	0.84 0.64	0.84 0.50
6						0.91 1.01	0.91 0.69
7							0.96 1.06

Figure A7.1

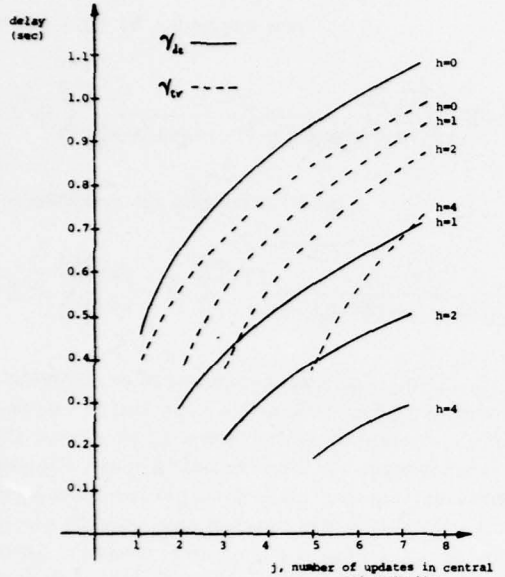


Figure A7.1. Comparison of the truncating with no knowledge to the delay at central node strategies. N=6, Ar=6 sec., M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

$$\gamma_{tp} = t_0 + \mathrm{E}[\mathrm{V}] \tag{34}$$

where random variable V is defined by

$$\mathrm{V} = \max(T, \mathrm{U}). \tag{35}$$

and random variable U is the delay from time t_0 to the time when the "perform update" messages for the m truncated updates have arrived. Notice that random variable U is exactly the same as random variable Z of section A7.2. Thus,

$$F_{\mathrm{U}}(x) = 1 - \sum_{i=0}^{m-1} \binom{j}{i} [1 - \exp(-x/\theta)]^i [\exp(-x/\theta)]^{j-i}. \tag{36}$$

The cumulative distribution function of V, $F_{\mathrm{V}}(x)$, is the same as $F_{\mathrm{U}}(x)$ if $x > T$ and is 0 otherwise. So using equation (26), we find the average value of V to be

$$\mathrm{E}[\mathrm{V}] = T + \int_T^\infty \sum_{i=0}^{m-1} \binom{j}{i} [1 - \exp(-x/\theta)]^i [\exp(-x/\theta)]^{j-i} dx. \tag{37}$$

Except for the lower integration bound, the integral is the same as the one in equation (11). Following the same steps we followed with that integral, we get

$$\mathrm{E}[\mathrm{V}] = T + \sum_{i=0}^{m-1} \binom{j}{i} (-1)^i \sum_{k=0}^{i} \binom{i}{k} \frac{(-1)^k \theta \exp((k-j)T/\theta)}{j-k}. \tag{38}$$

Unfortunately, the extra factor $\exp((k-j)T/\theta)$ does not allow us to simplify this equation as was done with equation (14). Hence, we will have to evaluate γ_{tp} with this equation.

A7.6 Comparison.

In this sub-section we will compare the performance of the truncating alternative with perfect future knowledge with the delay at central node alternative. We can make the following general observations regarding γ_{tp}, the time when

update A's computations are started at node x when the truncating alternative with perfect knowledge is used.

(a) If T is very small (i.e., $\exp(-T/\theta) \approx 1$), then $\gamma_{tp} = \gamma_{dc}$ and both the truncating strategy with perfect knowledge and the delay at central node strategy are equivalent (see equation (38)). If T is not small, then γ_{tp} will be smaller than γ_{dc} because of the $\exp(-T/\theta)$ factor in the sum of equation (38). (Compare this equation to equation (14).)

(b) As j increases for a fixed m, both γ_{tp} and γ_{dc} approach $t_0 + T$. γ_{tp} approaches $t_0 + T$ because in equation (38), $k < m$ and the limit as j goes to infinity of $\exp((k-j)T/\theta)$ is 0. In other words, as the number of updates we can choose from in order to truncate the m updates that do not fit in A increases, the delay in both strategies becomes the same and is equal to T seconds.

(c) If the hole list size limit h is 0 (i.e., $j = m$), then the truncating alternative with perfect knowledge should perform exactly like the truncating alternative with no knowledge because all j updates must be truncated in either case. This can be shown to be true by making $m = j$ in equation (38). Fortunately, in this special case, this equation can be simplified as follows: With $m = j$, γ_{tp} becomes

$$E[V] = T + \sum_{i=0}^{j-1} \binom{j}{i} (-1)^i \sum_{k=0}^{i} \binom{i}{k} \frac{(-1)^k \theta \exp(-(j-k)T/\theta)}{j-k}. \qquad (39)$$

Exchanging the summations and rearranging, we get

$$E[V] = T + \sum_{k=0}^{j-1} \frac{(-1)^k \theta \exp(-(j-k)T/\theta)}{j-k} \sum_{i=k}^{j-1} \binom{j}{i} \binom{i}{k} (-1)^i. \qquad (40)$$

Using [KNUT73, sec. 1.2.6, eq. 23] we find that

$$\sum_{i=k}^{j-1} \binom{j}{i} \binom{i}{k} (-1)^i = -\binom{j}{k} (-1)^j, \qquad (41)$$

so

$$E[V] = T - \sum_{k=0}^{j-1} \binom{j}{k} \frac{(-1)^{j-k} \theta \exp(-(j-k)T/\theta)}{j-k}. \qquad (42)$$

Finally, by changing the summation index variable k to $j - n$, where n goes from 1 to j, we see that equation (42) is the same as equation (29) with $m = j$. Therefore, in this case, $\gamma_{tp} = \gamma_{tr}$ and both truncating alternatives are equivalent. If $(T/\theta)^k/k$ for $k \geq 2$ is negligible as compared to T/θ, then γ_{tp} and γ_{tr} are approximately $t_0 + \theta H_j$ when $m = j$. Hence, either truncating strategy will save about T seconds in average response time as compared to the delay at central node strategy.

Table A7.2 shows the values of γ_{tp} and γ_{dc} for the typical parameter values of chapter 6, assuming that $t_0 = 0$. The network transmission time is $T = 0.1$ seconds and the average hole remaining time is $\theta = 0.37$ seconds. Recall that in this case, $T/\theta = 0.27$ and $(T/\theta)^k/k$ for $k \geq 2$ will be almost negligible compared to T/θ. Also notice that the difference in average response time of the two strategies is always less than or equal to T, with the truncating with perfect knowledge strategy always being superior, as expected. In figure A7.2 we plot the values of γ_{tp} and γ_{dc} as a function of j, the number of updates in the hole list, for various values of h, the hole list size limit.

A7.7 Conclusion.

If the central node has perfect future knowledge as to when updates will finish, then the truncating alternative is always superior. However, the saving as compared to the delay at central node strategy never seem to be more than T seconds in average response time. (In the cases we tested, the savings are always less than or equal to T seconds.)

If the central node has no idea when updates will finish, then the delay at central node alternative performs better than the truncating alternative in most cases of interest (e.g., $h \not\approx 0$). The equations we have obtained could be used to vary the strategy dynamically as each update is granted locks. Depending on the value of j (the number of updates in the hole list at the central node) and of m (the number of updates that must be truncated), the alternative with the lowest predicted response time could be chosen.

If the central node can only predict the termination of updates with a limited ability, then the strategy we choose will depend on the accuracy of these guesses. The best way to decide on a particular strategy in this case would be to perform actual tests in the real distributed database system.

Table A7.2

TABLE A7.2

COMPARISON OF THE TRUNCATING WITH PERFECT FUTURE KNOWLEDGE STRATEGY TO THE DELAY AT CENTRAL NODE STRATEGY FOR THE TYPICAL PARAMETER VALUES.

T = 0.1 sec., Θ = 0.37 sec., t0 = 0,
N = 6, Ar = 6sec., Id = Is = 0.025 sec., M = 1000 items, Bs = 5 items.

The top entry in each box is γ_{tp}, the time when update A's computations are started at node x when the truncating with perfect knowledge strategy is used, while the bottom entry in each box is γ_{dc}, the time when A's computations are started when the delay at central node strategy is used.

γ_{tp}
γ_{dc}

m \ j	1	2	3	4	5	6	7
1	0.38 0.47	0.21 0.29	0.15 0.22	0.13 0.19	0.12 0.17	0.11 0.16	0.11 0.15
2		0.56 0.66	0.31 0.41	0.23 0.32	0.18 0.27	0.15 0.24	0.14 0.21
3			0.68 0.78	0.40 0.50	0.29 0.39	0.23 0.33	0.20 0.29
4				0.77 0.87	0.48 0.57	0.35 0.45	0.28 0.38
5					0.84 0.94	0.54 0.64	0.40 0.50
6						0.91 1.01	0.59 0.69
7							0.96 1.06

Figure A7.2

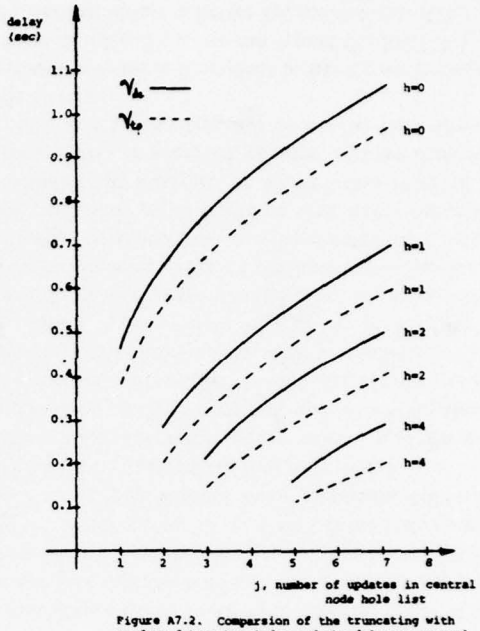


Figure A7.2. Comparsion of the truncating with perfect future knowledge and the delay at central node strategies. N=6, Ar=6 sec., M=1000, Bs=5, Is=Id=0.025 sec., T=0.1 sec.

A7.8 A Final Note.

In this analysis we studied the delay until update A's computations could be started at node x. The analysis did not take into account the following effect: In the truncating alternative, update A is released at node x when the "perform update" message of the last truncated hole list update B arrives at x. This means that it is more likely that A will wait longer for service at node x because we know that another request for service (mainly B's request) immediately preceded A's request. In the delay at central node strategy, this does not occur because B's "perform update" message arrives T seconds before A's "grant" message. Thus, update A's delay will be slightly larger than what we have predicted when a truncating strategy is used. Unfortunately, it seems hard to study these effects without using simulations. Finally, notice that since A and B have no items in common, a "smart" scheduler at node x could schedule the two requests "in parallel", thus eliminating the problem we have just described. (Exercise for the reader: Why do A and B have no items in common?)

(End of Appendix 7.)

APPENDIX 8.

In this appendix, we analyze the "read without locks and then request locks" strategy for the MCLA-h algorithm. We use the performance model and analysis techniques that were used in chapter 4 to analyze the MCLA-h algorithm. In particular, we assume that all databases are completely duplicated at each node and that all transactions are updates. We also assume negligible CPU times. The analysis in this appendix is simplified and rather pessimistic.

When the "read without locks" strategy is used, an update first reads data at its originating node and then requests locks at the central node. If there are no rejection at all, then the fact that the read and compute step was performed before the locks were requested instead of after (as in the original MCLA-h algorithm) does not affect the performance. Thus, if we assume that no rejections occurred, the performance of our "read without locks" algorithm will be the same as the performance of the MCLA-h algorithm. (See appendix 5.)

Our strategy to estimate the average response time of updates will be as follows. First we assume that no conflicts occurred, and we obtain the average response time from our previous analysis. Based on these results, we compute the average time during which an update is vulnerable to conflicts, and is hence vulnerable to rejection. Based on this value, we find the probability that an update is rejected. We assume that rejected updates place the same load on the system as updates that are not rejected. This is a pessimistic assumption because in reality, updates can be aborted as soon as a conflict is detected. Furthermore, rejected updates are not performed at all nodes so they never make these IO requests. However, to simplify the analysis, we assume that rejected updates produce the same service requests as accepted updates. But we should keep in mind that the response time in a real system will be smaller than the response time we obtain from this analysis.

Considering the rejected updates is equivalent to having an increase in the arrival rate of updates to each node. Thus, we can repeat the MCLA-h analysis with an increased arrival rate to obtain a better approximation. After this, we can recompute the update vulnerable period and the probability of rejection to repeat the MCLA-h analysis. We refine the results in this iterative fashion until there is no change in average response time or until we detect that the strategy does not converge.

After each MCLA-h analysis, we can compute the vulnerable period and the probability of rejection as follows. The vulnerable period of an update A starts when A arrives at its originating node and ends when A obtains all locks at the

central node. Thus, the average vulnerable period, $\overline{V_P}$, is

$$\overline{V_P} = \overline{W_{nc}} + I_d \mathrm{E[Y]} + T + \overline{W_c} + 2I_s \mathrm{E[Y]}$$

where $\overline{W_{nc}}$ and $\overline{W_c}$ are the average IO wait times at a non-central and central node, respectively, T is the transmission time, $I_d \mathrm{E[Y]}$ is the average time to read the data and $2I_s \mathrm{E[Y]}$ is the average IO time to obtain the locks at the central node. The vulnerable period of updates originating at the central node is slightly different but we ignore this here.

If we assume that all updates are performed at all nodes at the same time (see appendix 7), we can state that any "perform update" messages that arrive during the vulnerable period may cause a conflict with A. The number of such messages that arrive is

$$n_c = \text{number of completions} = N\lambda\overline{V_P}$$

where N is the number of nodes, and $N\lambda$ is the rate of update completions (which is also the original arrival rate of updates in a stable system.)

Probability $\Pr(C)$ is the probability that two updates conflict and is given by equation (23) of chapter 4. Hence, the probability that A does not conflict with one of the updates that completed in the vulnerable period is $1 - \Pr(C)$. The probability that A does not conflict with any of the updates is $(1 - \Pr(C))$ to the power n_c, and the probability of rejection of A is

$$\Pr(R) = 1 - [1 - \Pr(c)]^{n_c}$$

This implies that the rate of rejections is (arrival rate of updates) times $\Pr(R)$. This rate is added to the arrival rate of updates to give the total arrival rate for the next iteration of the MCLA-h analysis. The expected response of updates will be $(1 + \Pr(R))$ times the average response time of a simple update that is not rejected (assuming only one rejection per update transaction.)

We now we give a complete listing of the program that performs the analysis we have described. Notice that the program for the MCLA-h analysis (appendix 5) is part of the new program.


```
begin "program"


require "{} ()" delimiters;
define crlf = {('15&'12)};
define cr = {('15)};
define $ = {comment};

$  This program computes the average response time of an update
   in the MCLA centralized locking algorithm when the base set is read
   without locks initially.  This program is based on the program of
   appendix 5.  The new variables are "origLam" (the original
   value of lambda), "numComp" (the number of updates that complete in the
   vulnerable period), "oldLam" (the last value of lambda), "Pr" (the
   probability of rejection), and "j2" (a counter);

external integer !skip!;
integer bk;  string rep;
integer j, j2;
real temp,N,M,Ar,Bs,Is,Id,T,lambda,EY,EY2,EZ,EZ2;
real Pw,L,restime,oldrestime,Lnc,Lc;
real rate,Xc,Xc2,roo,Wc,Xnc,Xnc2,Wnc,Rnc,Rc;
real EYgc, EY2gc, EZgc, Lgc, Lncgc, Lcgc;;
real Pc, P2w, Pwl, Pw2, EYrem, EY2rem;
real origLam, oldLam, numComp, Pr;

real procedure power(real x, y);
  begin
  real temp;
  if x = 0 then return(0.0);  temp ← y*log(x);
  if temp < -20 then return(0.0) else return( exp(temp) );
  end;
```

```
procedure SolveSystem;
  begin "compute"
  rate← (2*N + 1) + Pw*N;
  Xc← N * 2 * Is * EY;
  Xc← Xc + Id*EY;
  Xc← Xc + N * (Is*EY + Id*EZ);
  Xc← Xc + Pw * N * 2 * Is *( EYrem + P2w*(EYrem-1)/2 );
  Xc← Xc / rate;
  Xc2← N * 4 * Is*Is * EY2;
  Xc2← Xc2 + Id*Id * EY2;
  Xc2← Xc2 + N * ( Is*Is*EY2 + Is*Id*(EY + EY2) + Id*Id*EZ2 );
  Xc2← Xc2 + Pw * N * 4*Is*Is*
       (EY2rem + P2w*(EY2rem/3 - EYrem/2 + 1.0/6.0) );
  Xc2← Xc2 / rate;
  roo← lambda * rate * Xc;
  if roo geq 1 then
      begin
      print(crlf, "******* SYSTEM IS UNSTABLE *******");
      roo← lambda← 0;
      end;
  Wc← ( ( lambda * rate /2 ) * Xc2 )/( 1 - roo );
  rate← N + 1;
  Xnc← ( Id*EY + N * Id*EZ )/rate;
  Xnc2← ( Id*Id*EY2 + N * Id*Id*EZ2 )/rate;
  roo← lambda * rate * Xnc;
  if roo geq 1 then
      begin
      print(crlf, "******* SYSTEM IS UNSTABLE *******");
      roo← lambda← 0;
      end;
  Wnc← ( ( lambda * rate /2 ) * Xnc2 )/( 1 - roo );
  Rnc← 2 * Wnc + Wc + 2*Is*EY + Id*(EY + EZ) + 2*T;
  Rc← 3 * Wc + 3 * Is * EY + Id*( EY + EZ );
  restime← ( (N - 1)*Rnc  + Rc )/N  +  Pw*(Lgc/2 + Wc + Is*(EYgc-1))
     + Pw2*Lgc + Pw*P2w*(Lgc/2 + Wc + Is*(EYrem - 1));
  end "compute";


procedure conflict!analysis;
  begin "conflict!analysis"
  Lnc← T + Wnc + Id*EY + T + Wc + Is*EY + Id*EZ;
  Lc← Wc + Id*EY + Wc + Is*EY + Id*EZ;
  L← ( (N-1)*Lnc + Lc )/N;
  Lncgc← T + Wnc + Id*EYgc + T + Wc + Is*EYgc + Id*EZgc;
  Lcgc←  Wc + Id*EYgc + Wc + Is*EYgc + Id*EZgc;
  Lgc← ( (N-1)* Lncgc  +  Lcgc )/N;
  Lgc← Lgc + ((Lgc/2)*N*lambda*EY/M)*Lgc;
  Pw1← Pc * N * lambda * L;
  Pw2← Pc*(Lgc/2 + Wc + Is*(EYgc-1))*N*lambda*Pw1;
  P2w← (EYrem*EY/M)*N*lambda*L;
  Pw← Pw1 + Pw2;
  end "conflict!analysis";
```



```
while true do
  begin "main"
  $ Read in parameters;
  print("number of nodes    N = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then N← temp else print(N,crlf);
  print("number of items    M = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then M← temp else print(M,crlf);
  print("interarrival time  Ar = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Ar←temp else print(Ar,crlf);
  print("mean base set     Bs = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Bs←temp else print(Bs,crlf);
  print("IO slice          Is = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Is←temp else print(Is,crlf);
  print("IO data           Id = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then Id←temp else print(Id,crlf);
  print("transmission time  T = "); rep←intty; temp←realscan(rep,bk);
  if !skip! = cr then T← temp else print(T,crlf);

  lambda← 1/Ar;    EY← 1/(1 - exp( -1/Bs ));
  EY2←  EY * EY * (1 + exp( -1/Bs ));
  EZ←  ( EY + 1.0 )/2.0;
  EZ2←  EY2/3 + EY/2 + 1.0/6.0;
  EYgc← 2 * exp(-1/Bs) * EY + 1;
  EY2gc← 6*exp(-1/Bs)*EY*( exp(-1/Bs)*EY + 1)  +  1;
  EZgc← ( EYgc + 1.0 )/2.0;
  EYrem← ( EYgc - 1.0 )/2.0;
  EY2rem← EY2gc/3.0 - EYgc/2.0 + 1.0/6.0;
  Pc← EY * EY / M;

  j2← 0;  origLam← lambda;  print(crlf,"lambda= ", lambda);
  do begin

    $ This is the original MCLA-h analysis;
    Pw← Pw1← Pw2← P2w← 0;  j← 0;  L← Lgc← 0;  oldrestime← 0;
    SolveSystem;  $ Result is restime, Wc, Wnc;
    do begin
       print(crlf,restime);
       j← j + 1;
       conflict!analysis;  $ Result is Pw, L, Lgc;
       oldrestime← restime;
       SolveSystem;  $ Result is restime, Wc, Wnc;
       end
    until ( j > 5 ) or ((restime - oldrestime)/restime < .01 );
    print(crlf,"==> mean response time = ", restime,
            " iterations = ", j);
    print(crlf,"  Wnc = ",Wnc,"  Wc = ", Wc);
    print(crlf,"  EY  = ", EY,"  EZ = ", EZ, crlf, crlf);
    $ End of the original MCLA-h analysis;

    $ Now we compute the probability of rejection;
    j2← j2 + 1;  oldLam← lambda;
    numComp← N*origLam*(Wnc + Id*EY + T + Wc + 2*Is*EY);
    Pr← 1 - power( (1-Pc), numComp );
    lambda← origLam*( 1 + Pr );
    print(crlf, " new res time is ", restime*(1 + Pr),
                " new lambda is ", lambda);
    end
  until (j2 > 5) or ((lambda - oldLam)/lambda < 0.001 );
  print(crlf, "END", crlf);

  end "main"
end "program"
```

REFERENCES

[ALSB76] P. A. Alsberg and J. D. Day, A Principle for Resilient Sharing of Distributed Resources,2nd Intl. Conf. on Software Engineering, San Francisco (1976) 562-570.

[ASCH74] F. Aschim, Data Base Networks-An Overview,Management Informatics Vol. 3 Num. 1 (1974) 13-28.

[BADA78] D. Badal and G. Popek, A Proposal for Concurrency Control for Partially Redundant Distributed Database Systems,3rd Berkeley Workshop on Distributed Data Management, San Francisco (1978) 273-285.

[BERN78] P. A. Bernstein, J. B. Rothnie Jr., N. Goodman, and C. A. Papadimitriou, The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case),IEEE Trans. on Software Engineering, Vol. 4, Num. 3 (1978) 154-168.

[CASE72] R. G. Casey, Allocation of Copies of a File in an Information Network,Spring Joint Computer Conf. (1972) 617-625.

[CHU69] W. W. Chu, Optimal File Allocation in a Multiple Computer System, IEEE Trans. on Computers, Vol. 18 Num. 10 (1969) 885-889.

[CHU74] W. W. Chu and G. Ohlmacher, Avoiding Deadlock in Distributed Databases,ACM National Symposium, Vol. 1 (1974) 156-160.

[CHU75] W. W. Chu and E. E. Nahouraii, File Directory Design Considerations for Distributed Databases,Intl. Conf. on Very Large Databases, Framingham (1975) 543-545.

[COFF71] E. G. Coffman, M. J. Elphick and A. Shoshani, System Deadlocks, Computing Surveys Vol. 3 Num. 2 (1971) .

[COMB75] P. G. Comba, Needed: Distributed Control,Intl. Conf. on Very Large Databases, Framingham (1975) 364-373.

[ELLI77] C. A. Ellis, Consistency and Correctness of Duplicate Database Systems,6th Symposium on Operating System Principles (1977) 67-84.

[ESWA76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, The Notions of Consistency and Predicate Locks in a Database System, Communications of the ACM Vol. 19 Num. 11 (1976) 624-633.

[FREU71] J. E. Freund, Mathematical Statistics,Prentice Hall (1971) .

[FRY76] J. P. Fry and E. H. Sibley, Evolution of Data-Base Management Systems,ACM Computing Surveys Vol. 8 Num 1 (1976) 7-42.

[GARC77] H. Garcia-Molina, Overview and Bibliography of Distributed Data Bases,Report HPP-77-27, Computer Science Department, Stanford University (1977) .

[GARC78] H. Garcia-Molina, Performance Comparison of Update Algorithms for Distributed Databases,Technical Note 143, Digital Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University (1978) .

[GARC78b] H. Garcia-Molina, Distributed Database Coupling,3rd USA-Japan Computer Conference, San Francisco (1978) 75-79.

[GORD78] G. Gordon, System Simulation,Prentice Hall (1978) .

[GRAP76] E. Grapa, Characterization of a Distributed Database System,UIUCDCS-R-76-831 Department of Computer Science, University of Illinois at Urbana-Champaign (1976) .

[GRAY76] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger, Granularity of Locks and Degrees of Consistency in a Shared Data Base,Modelling in Data Base Management Systems, North Holland (1976) 365-394.

[GRAY77] J. N. Gray, Notes on Database Operating Systems,Advanced Course on Operating Systems, Technical University Munich (1977) .

[GRAY79] J. N. Gray, Personal Communication (1979).

[JACK57] J. R. Jackson, Networks of Waiting Lines,Operations Research 5 (1957) 518-521.

[JOHN75] P. R. Johnson and R. H. Thomas, The Maintenance of Duplicate Databases,Networking Working Group RFC 677 (1975) .

[KLEI75] L. Kleinrock, Queueing Systems, Volumes 1 and 2,John Wiley and Sons (1975) .

[KNUT73] D. E. Knuth, The Art of Computer Programming, Vol. 1,Addison-Wesley (1973) .

[LAMP78] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System,Communications of the ACM Vol. 21 Num. 7 (1978) 558-564.

[LAMP] B. Lampson, and H. Sturgis, Crash Recovery in a Distributed Data Storage System, Technical Report XEROX PARC, Unknown Date.

[MAHM76] S. Mahmoud and J. S. Riordon, Optimal Allocation of Resources in Distributed Information Networks,ACM Trans. on Database Systems Vol. 1 Num. 1 (1976) 66-78.

[MARY77] F. J. Maryanski, A Survey of Developments in Distributed Database Management Systems,Technical report CS77-08, Kansas State University (1977) .

[MENA78] D. Menasce, G. Popek, and R. Muntz, A Locking Protocol for Resource Coordination in Distributed Databases,SIGMOD Intl. Conf.

on Management of Data, Austin (1978) .

[MERT74] K. Yamaguchi and G. Merten, Methodology for Transferring Programs and Data,*ACM SIGMOD Workshop* (1974) 141–155.

[METC76] R. Metcalfe and D. Boggs, Ethernet: Distributed Packet Switching for Local Computer Networks,*Communications of the ACM Vol. 19 Num. 7* (1976) .

[PAPO65] A. Papoulis, Probability, Random Variables, and Stochastic Processes, *McGraw Hill* (1965) .

[ROSE78] D. Rosenkrantz, R. stearns, and P. Lewis, System Level Concurrency Control for Distributed Database Systems,*ACM Trans. on Database Systems Vol. 3 Num. 2* (1978) 178–198.

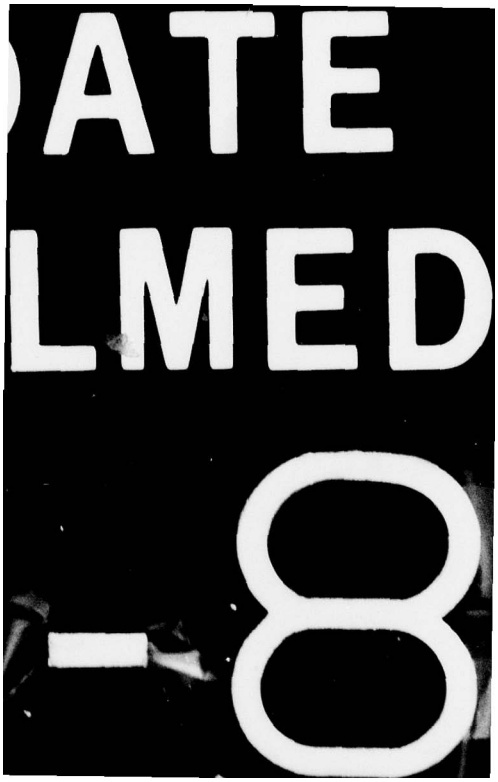
[ROTH77] J. Rothnie and N. Goodman, A Survey of Research and Development in Distributed Database Management,*3rd Intl. Conf. on Very Large Databases, Tokyo* (1977) .

[STON77] M. Stonebraker and E. Neuhold, A Distributed Data Base Version of INGRES,*2nd Berkeley Workshop on Distributed Data Management* (1977) 19–36.

[STON78] M. Stonebraker, Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES,*3rd Berkeley Workshop on Distributed Data Management* (1978) 235–258.

[THOM76] *R. H. Thomas, A Solution to the Update Problem for Multiple* Copy Data Bases Which Uses Distributed Control,*BBN Report 3340* (1976)

[WONG77] E. Wong, Retrieving Dispersed Data from SDD-1: A System for Distributed Databases,*2nd Berkeley Workshop on Distributed Data Management* (1977) 217–235.


ATE
LMED
-8