1

A

CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

Canada

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Division

Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

55597

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Ecrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

Giuseppe DiUbaldo

Date of Birth — Date de naissance

20 Feb 1954

Country of Birth — Lieu de naissance

Italy

Permanent Address — Résidence fixe

618 Westminister Avenue
Ottawa, Ontario
CANADA K2A 2V5

Title of Thesis — Titre de la thèse

Basic Timestamp ordering for
Concurrency Control in
Distributed Databases

University — Université

Carleton University

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

M. Sc.

Year this degree conferred — Année d'obtention de ce grade

1982

Name of Supervisor — Nom du directeur de thèse

S. A. Mahmoud

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film

L'auteur se réserve les autres droits de publication, ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur

Date

5 April 1982

Signature

G. DiUbaldo

BASIC TIMESTAMP ORDERING FOR
CONCURRENCY CONTROL IN
DISTRIBUTED DATABASES

by

GIUSEPPE DI UBALDO, B.Sc.

A thesis submitted to the Faculty of Graduate
Studies in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Systems Engineering
and Computing Science.

Carleton University

Ottawa, Ontario

April, 1982

The undersigned recommend to the Faculty of
Graduate Studies and Research acceptance of the thesis
"Basic Timestamp Ordering for Concurrency Control
in Distributed Databases"
submitted by Giuseppe Di Ubaldo in partial fulfilment
of the requirements for the degree of Master of Science:

Thesis Supervisor

Chairman, Department of Systems
and Computer Engineering

Carleton University

30 April 1982

ABSTRACT

One of the many problems to be dealt with in a database system is the synchronization of conflicting database requests. Concurrency control is the means of controlling concurrent transactions in order to maintain each transaction's view of atomicity and yet allow as high a degree of parallelism in execution as possible. Most approaches to concurrency control involve variations on the two main strategies: two-phase locking (2PL) and timestamp ordering (T/O). Timestamp ordering schemes can be classified as basic T/O or conservative T/O depending on whether restart or delay is the dominant synchronization tactic.

This thesis will examine the issues in concurrency control in the context of a distributed database. The ADD protocol [MAHM79], which uses basic T/O, will be investigated in detail. One assumption made in the ADD proposal is the ability of transactions to predeclare their resource requirements. Thus, the ADD* protocol will be presented as an extension to the ADD protocol, to handle the general case of dynamic read/write requests (nonpredeclaration). Both the ADD and ADD* protocols will be analyzed using formal protocol specification and verification techniques. In addition, analytic results will be presented for basic T/O and results of simulation runs will compare basic T/O and conservative T/O schemes. The factors contributing to transaction disordering and rejection in basic T/O will be investigated using an M/M/ ∞ network queueing model with feedback.

To Rachel,
whose love and constant support
helped make this work possible

ACKNOWLEDGEMENT

I would like to acknowledge all those whose assistance and support have made this work possible.

Special thanks are due to my thesis supervisor, Dr. S. A. Mahmoud, for his valuable discussions and suggestions throughout this work.

Thanks are also due to Dr. M. C. Woodside, Dr. H. M. Hafez and Mr. Barry Thomas for their discussions during the formulation of the material in Chapter 5.

Thanks are due to the National Research Council for their financial support during the course of this study.

I would especially like to thank Kathy Therrien who gave her time generously in the preparation of this document.

TABLE OF CONTENTS

	<u>Page No.</u>
Abstract	iii
Acknowledgement	v
Table of Contents	vi
List of Figures	ix
Chapter 1 INTRODUCTION	1
1.1 Thesis Objectives	4
1.2 The Process Synchronization Problem	5
1.3 Formal Analysis of Protocols	8
1.4 Performance Evaluation	11
1.5 Outline of Thesis	12
Chapter 2 DDB ENVIRONMENT	14
2.1 Database Definition and Environment	14
2.2 Transactions	17
2.3 Distributed DB Architecture	19
2.4 Execution Flow	21
2.5 Transaction Transparency	23
2.6 Transaction Processing Models	28
2.7 Issues in Correctness	34
2.7.1 Semantic Integrity	34
2.7.2 Serializability	38
2.7.3 Two-Phase Locking and Timestamp Ordering	41
2.7.4 Increasing Concurrency	43

Chapter 3	REVIEW OF CURRENT PROPOSALS	45
3.1	Basics of Concurrency Control Mechanisms	45
3.2	Two-Phase Locking	49
3.2.1	Deadlock Detection (2PL/DD)	50
3.2.2	Deadlock Prevention (2PL/DP)	51
3.2.3	Variations on 2PL	54
3.3	Timestamp Ordering	56
3.3.1	Basic Timestamp Ordering	56
3.3.2	Conservative Timestamp Ordering	59
3.3.3	Variations on Timestamp Ordering	62
3.4	Performance Evaluation of Concurrency Control	63
Chapter 4	THE ADD AND ADD* PROTOCOLS	67
4.1	Specification of the ADD Algorithm	67
4.2	Protocol Verification	75
4.2.1	Correctness	75
4.2.2	Transaction Termination	79
4.2.3	Reducing Restarts	81
4.3	Dynamic Request Generation	84
4.3.1	The ADD* Protocol: Dynamic Requests	84
4.3.2	Handling Restarts in ADD*	92
4.3.3	Verification of ADD*	96
4.3.4	Examination of Restarts in ADD*	98
4.4	Hybrid T/O Schemes	99
Chapter 5	PERFORMANCE ANALYSIS OF BASIC T/O	104
5.1	Factors Affecting Disorder	104
5.2	An M/M/ ∞ Model for Conservative T/O	106
5.2.1	Results of Analysis	109
5.3	An M/M/ ∞ Model for Basic T/O	110
5.3.1	Towards the Derivation of P[star]	113

5.3.2	Accounting for the Feedback Loop	120
5.3.3	Concerning the Feedback Process	129
5.4	Results Based on $P[\text{star}]$	132
5.5	Multiple DM Requests	137
5.6	Effects of DM Delay in Basic T/O	142
5.7	Effects of Non-Exponential Server	152
Chapter 6	SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH	157
6.1	Summary of Research	157
6.1.1	The ADD* Protocol	157
6.1.2	Protocol Analysis	158
6.1.3	The M/M/ ∞ Feedback Model	159
6.1.4	Effects of DM Node Delay	160
6.2	Suggestions for Further Research	161
Appendix		164
Glossary		167
References		169

LIST OF FIGURES

<u>Figure No.</u>		<u>Page No.</u>
1.1	Logical Structure of the DDB	2
2.1	Processing Layers of the ADD Architecture [TOTH80]	20
2.2	Functional Modules of the DP Layer within ADD	22
2.3	Functional Separation of TM/DM within ADD	26
2.4	Revised DDB Architecture	27
2.5	Example of Executions	39
3.1	Pseudocode Sketches of the Proposals Discussed	47
3.2	Evolution of a $\prec$ graph for a Concurrent Execution	48
4.1	Structure of ADD Protocol	69
4.2	Structure of Global Coordinator	70
4.3	The Master TCP Algorithm at TM_i	73
4.4	The Slave TCP Algorithm at DM_k	73
4.5	The RAT Monitor: Data Structure and Procedures	74
4.6	FSM for the TM and DM Transaction Control Processes	80
4.7	Structure of the ADD* Protocol	86
4.8	Master TCP at TM_i	90
4.9	Slave TCP at DM_k	90
4.10	Evaluate Monitor Procedure	91
4.11	Process Monitor Procedure	91
4.12	FSM for Slave TCP at DM_k	93

4.13	FSM for Master TCP	94
4.14	TABLE: Hybrid T/O Schemes	101
5.1	Queueing Model of [KAM081]	107
5.2	State-Transition-Rate Diagram for Tagged Customer	114
5.3	Logical Description of Transaction Flow	121
5.4	Probability of Acceptance for Open and Feedback Systems	126
5.5	Probability of a Star Customer for Conservative and Basic T/O	128
5.6	Random Processes of a Queueing System with Feedback [DISN80]	130
5.7	TABLE: Theoretical and Simulation Results for P[star]	131
5.8	Expected Retries for Basic T/O	134
5.9	Network Delay for T/O Schemes	135
5.10	TABLE: Simulation Results for Number of Retries and Normalized Network Delay	136
5.11	Multiple DM Accesses	138
5.12	P[acc] for Multiple DM System; $\alpha = 1$	140
5.13	P[acc] for Multiple DM System; $n = 10$	141
5.14	Network and Node Queueing Systems in Series	144
5.15	TABLE: Values of ρ_1 and ρ_2 Used in Simulations	146
5.16	TABLE: P[rej nonstar] for various E[Y] and E[Z]	147
5.17	Effects of Node Delay for Basic T/O	148
5.18	TABLE: Ratio of E[W] Values for Conservative T/O and Basic T/O	150
5.19	Total Delay for Both T/O Schemes	151

5.20	Effect of Delay on Probability of Rejection	153
5.21	TABLE: Total Delay (T) for Various Node Delays (d)	154
5.22	P[nonstar] for Various Service Distributions	156

CHAPTER 1

INTRODUCTION

In the past decade there has been much research in the area of distributed computing systems. The term has been used to describe anything from tightly coupled multiprocessor architectures for a single computer, to loosely coupled, geographically dispersed computers interconnected by a network.

A particular application of distributed computing that has received much attention in the literature is that of a distributed database (DDB) system. The usual architecture for a DDB involves a set of host computers, each supporting a conventional database (DB), interconnected by a communication network as illustrated in Figure 1.1. Each host supports a user interface to the DDB. A user query may be executed completely locally or it may be broken into subqueries where each subquery requires cooperation on the part of a local and/or remote DB host. Support for the distributed (that is, nonlocal) queries requires an exchange of messages by the cooperating hosts.

The evolution of database management systems (DBMS) has almost gone full cycle. Initially, integration of diverse file systems into a centralized DBMS solved problems such as data integrity and redundancy. Now that inexpensive terminals have given a wide range of remote users access to the database, communication costs and availability have been motivating design factors leading towards geographically dispersed but logically integrated DDB systems. For instance, the emergence of Telidon technology will allow many users to access data, stored in various

Figure 1.1 Logical Structure of the DDB

databases, from their homes. The design of the underlying DB structure (as either a centralized or distributed system) will be a major issue.

To date, DDB systems have been either research tools such as MUFFIN [STON79] (the distributed version of INGRES) and R* [LIND80] (the distributed version of IBM's System R) or prototypes such as SDD-1 [ROTH80] (developed by Computer Corporation of America). Before gaining wide acceptance commercially, the DDB technology must be demonstrated as being a viable alternative to relatively well understood centralized DBs. The particular technical problem in distributed (and centralized) databases addressed in this thesis is the concurrency control problem.

In multi-user computer systems where processes are competing for the use of conflicting resources, process synchronization involves some knowledge of the state of the processes and the resources so that no system constraints are violated. This problem is exacerbated in a distributed system since, given our current level of technology, the global state of all resources involved is not known. Some partial information is available, however, based on:

1. (normal) message exchange in performing the distributed task, and
2. special status messages such as ACKs to gather further system status information.

Concurrency control in databases is related to the process synchronization problem and deals with the control of conflicting concurrent transactions so as to maintain consistency assertions amongst the data entities. At a low level of interaction, we must maintain shared read/exclusive write (SR/EW) mutual exclusion. In a distributed, dynamic allocation situation, interleavings of read and

write operations on the database could lead to inconsistent data.

The characteristics of a DDB (with its interconnecting network), that makes the concurrency control problem difficult are:

1. message transit delays between hosts are variable and not negligible compared to the time between execution of events by any host;
2. ordering of messages may only be guaranteed between any two nodes; and
3. failures may occur due to loss of messages, and link and/or node failures.

A treatment of the general problem of ordering events for process synchronization is given in [LAMP78].

1.1 Thesis Objectives

The objective of this research is to study the basic timestamp ordering technique of the ADD concurrency control mechanism [MAHM79] for a distributed database environment.. In doing so, the thesis focuses on three areas. First, the problem must be defined and the relevant issues understood. Concurrency control is discussed in relation to the general process synchronization problems in order to understand their similarities and differences. The role of concurrency control in the distributed database environment is described. Second, as a protocol, concurrency control must be subjected to formal analysis. The ADD* protocol is presented as an extension of the ADD protocol to handle dynamic request generation. Formal specification and verification tools are used to analyze both protocols. Third, performance evaluation of the basic timestamp ordering is investigated. Two important parameters

for evaluating a concurrency control are the expected number of restarts and the expected delay before a transaction is committed. The remainder of this chapter will discuss in greater detail the above three aspects of concurrency control. An outline of the thesis organization is contained in Section 1.5.

1.2 The Process Synchronization Problem

The concurrency control problem in databases is closely related to the general problem of process synchronization in operating systems. Synchronization is "any constraint on the ordering [execution] of operations in time" [BRIN73]. An example of an ordering restriction posed by a precedence constraint is that operation A must be executed before operation B. In general, each operation can only be executed if it does not violate any constraints. In the example, the constraint is a condition that must be satisfied before operation B can be allowed to execute.

Some form of synchronization is required for computer systems which support:

1. multi tasking - more than one process executing in time (even if only logically as in a single CPU time-sharing system), and
2. resource sharing - a pool of common resources exists that more than one process can access.

Clearly if either factor is not present then synchronization is not required.

One of the operational objectives in a computer system is to achieve maximum parallelism or concurrency of execution in order to

maximize resource usage and system throughput. Unfortunately, physical and logical constraints on the processes and resources necessitate some form of process synchronization or resource scheduling policy. Some examples of system constraints are:

- (a) mutual exclusion of resource - there may exist only one instance of a special processor which does not allow concurrent usage (for example a disk controller);
- (b) mutual exclusion of operations - only one process may perform a write on a memory location at one time;
- (c) maximum resources - only n processes may use the n available CPUs in a multiprocessor system at any time;
- (d) precedence - operation A of process P cannot be executed before operation B; and
- (e) condition - operation A cannot be executed before condition X is satisfied (for example event primitives, security checks, semantic integrity constraints).

The idea of a constraint on the execution of actions within a process leads to the notion of conflict.

Definition. Two actions conflict if the concurrent execution of both actions would violate any system constraint.

In an operating systems context, resources such as CPU, memory and I/O devices, are managed using mechanisms such as monitors and semaphores [BRIN73].

Mutual exclusion constraints lead to the notion of scheduling processes based on resource allocation. The scheduler itself is a resource and so care should be taken that conflicting processes do not simultaneously access a common resource. This assumes that all request and release actions must be performed serially by the resource scheduler.

Each action $A(R)$ by some process P on resource R must be translated explicitly or implicitly to $\langle \text{request}(R), A(R), \text{release}(R) \rangle$. In the request phase, process P announces its intention to use resource R to the resource scheduler. Based on whether any system constraints will be violated, permission to proceed will either be granted or denied. If permission is granted, process P may execute $A(R)$ and once completed inform the resource scheduler via the release operation. If permission is denied, the process P must wait until the condition causing the conflict is no longer present. The waiting process may have to try again later or it may be placed in a waiting queue and informed by the resource scheduler as to when it may proceed. Due to the possibility of a conflict situation each initiated process may be in one of the following states:

active - the process has been granted access to a resource and is executing, or

blocked - the process cannot execute the next action because of a conflict and so is waiting until the conflict disappears.

Up to this point we have assumed the atomicness or indivisibility of the execution of each action. Once an action is allowed to start execution, it must be allowed to proceed to completion without interruption (at least logically). Even though time slicing may occur, as far as resource scheduling is concerned, the time sliced action still has control over its resource. Any other process wanting to access the same resource would be placed in a blocked state.

For certain processes to accomplish their tasks correctly we may require that two or more actions be considered atomic. Thus, we are defining a macro-action (composed of two or more atomic actions) which

may require more than one resource to execute. To maintain atomicness, a process might hold all resources it uses during the course of its execution. Use of dynamic request allocation can lead to a deadlock situation where a series of processes are each waiting on one of the other processes to effect a change in the constraint state. A new scheduling policy is needed in order to break deadlocks. Preemption involves restarting one of the processes in the deadlock cycle so that other processes may proceed. Bulk request or predeclaration of resources is sometimes considered a solution to the deadlock problem, but depending on how it is applied, deadlock and indefinite restart may still occur.

The issues we have mentioned concerning process synchronization will be discussed further in Chapters 2 and 3 in the context of concurrency control.

1.3 Formal Analysis of Protocols

A concurrency control mechanism (sometimes called a synchronization protocol) has the task of maintaining consistency amongst the data entities of a DDB in spite of concurrent accesses. A protocol is a set of rules governing the cooperation of independent, communicating processes in accomplishing a distributed task. Formal protocol analysis is important due to the increasing complexity of protocols being employed in distributed systems such as communication networks and DDBs. Analysis tools provide the means of precise description of a protocol via specification techniques and proof of logical correctness via

verification techniques. The following overview of formal analysis is based on an article by Bochmann [BOCH80].

Protocols can be subdivided into a hierarchy of functional layers (as in the Open Systems Interconnection recommendations) in order to reduce the overall complexity with respect to specification and verification. Each layer, n , is defined in terms of a protocol specification describing the way in which peer layers cooperate. A protocol specification should include:

1. a general description of the purpose and services provided by each layer;
2. an exact specification of the services provided;
3. the services assumed of the lower layer;
4. the internal structure of each layer in terms of modules and their relations; and
5. a description of the operation between peer layers using
 - an informal description,
 - the types and formats of all messages exchanged, and
 - the rules governing reactions to each message or event.

The tools for specification fall into two main categories: transition models and programming languages. "Transition models are motivated by the observation that protocols consist largely of simple processing in response to events such as commands [from the $n+1$ layer] and messages arrivals [from a peer n layer] ... Programming languages are motivated by the observation that protocols are simply one type of algorithm and that high-level languages provide a clear and relatively concise means of describing algorithms."

Verification involves demonstrating that a protocol satisfies design objectives (specifications) through use of formal reasoning.

"While testing and simulation allow validation under certain test conditions, verification, in principle, allows consideration of all possible situations that may be encountered during actual operation."

The properties of interest for verification include:

1. correctness with respect to specification requirements;
2. freedom from deadlock;
3. provision for all possible inputs; and
4. termination of operation.

The main tools for verification fall into two categories: reachability analysis and program (or assertion) proving. Reachability analysis is used, in conjunction with a state transition specification technique such as Petri-nets or finite state machines, to exhaustively analyze all possible global states of the system. The major difficulty with reachability analysis is "state space explosion" due to the number of states and the complexity of the protocol involved. Assertion proving involves the formulation of assertions or invariants which reflect correctness and the use of (sometimes automated, often manual) theorem-proving. Although the technique is powerful in that many possible situations can be verified by a few simple assertions, it is still necessary for the verifier to formulate the appropriate assertions. Due to the human factor, the assertion proved may not correctly verify the protocol.

These techniques are used in Chapter 4 in presenting the ADD and ADD* protocols. Both programming languages and finite state models will be used in protocol specification. Due to the possible participation of

many nodes in the protocol, assertion proving is adopted for verification rather than reachability analysis. A further discussion of the verification correctness criteria is presented in Section 2.7.

1.4 Performance Evaluation

Formal verification tools can provide proof of logical correctness of a protocol but, outside of proof of termination in a finite time, can say little with respect to the efficiency of a protocol. In Chapter 4 we shall see that ADD and ADD* are subject to multiple rejections which can impose a high overhead in terms of network message traffic and transaction delay. Thus it is important to have a grasp of the performance aspects of the protocol operation. The two main tools for performance analysis are simulation and analytic (or mathematical) modelling.

Simulation has the advantage that a complex system can be modelled and numeric results obtained for any parameters of interest. The disadvantage of simulation is the expense of the many runs needed to understand the behaviour of the system. Since many variables are being modelled simultaneously, it is difficult to derive the inter-relationship between any pair or group of variables.

Analytic modelling is powerful as the dynamics of a system can be explained concisely. Unfortunately, for a system of any complexity the analysis often becomes intractable. To alleviate this problem simplifying assumptions are made; the danger, however, is that the results may no longer accurately reflect the true behaviour of the

modelled system.

The approach in Chapter 5 is to use a combination of queueing analysis and simulation models. The study focuses on a few important parameters, particularly on the effects of variable network delay and node queueing delay on the probability of transaction acceptance. From this, the expected number of rejections and the expected total synchronization delay can be derived.

1.5 Outline of Thesis

Chapter 2 gives a presentation of the assumptions and environment leading into a transaction processing model for a DDB. The issues in correctness such as semantic integrity and serializability will be discussed.

A detailed survey of the literature on concurrency control is presented in Chapter 3. The major proposals are examined as variations of the main strategies: two-phase locking and timestamp ordering. Each proposal will be further examined in terms of how it employs the synchronization tactics of restart and delay. A discussion of efforts towards performance evaluation of concurrency control methods will also be presented.

In Chapter 4, the ADD protocol is described and verified using formal protocol analysis techniques. The ADD* protocol is developed to handle a dynamic request policy. The actions taken upon transaction restart are described and formal analysis techniques are applied to ADD*.

In Chapter 5 analytic results will be developed for the basic timestamp ordering algorithm. The primary results determined are the expected number of restarts and the expected delay per transaction for a given environment. Basic timestamp ordering will be compared with the conservative timestamp ordering scheme using simulation models.

Chapter 6 will conclude with a summary of contributions and suggestions for future research.

CHAPTER 2

DDB ENVIRONMENT

In this chapter we discuss the overall DDB environment and locate the concurrency control problem in that environment. The hierarchical ADD architecture for DDBs is reviewed and the Distributed Processor layer is examined in further detail with respect to functional modules and transaction flow. The ADD architecture is shown to maintain transparency of the underlying distribution of data.

Delayed update and atomic commitment are crucial to the concurrency control problem due to the possibility of node and link failures. A general transaction processing model is outlined which incorporates the above mechanisms, independent of the synchronization mechanism. A discussion of the issues in correctness is presented, with particular reference to semantic integrity and serializability as a criteria.

2.1 Database Definition and Environment

In Section 1.2, issues with respect to synchronization in a multi-tasking, resource-sharing environment were discussed. We now translate some of those concepts into the context of a relational distributed database system.

Definition. A logical entity is a unit of access in a database which is addressable by name. An entity corresponds to the notion of a resource in process synchronization.

The granularity [GRAY75] of an entity (whether the unit of access is a field, tuple, partition or relation) is application dependent and can

affect performance criteria such as overhead and throughput. Although it is possible [GRA75] (and perhaps desirable for performance reasons), to deal with variable levels of granularity, we will assume a fixed, unspecified granularity for ease of presentation.

Definition. A logical database $D_t = \{\dots, x, y, z, \dots\}$ is a collection of the logical data entities that exist in a DDB at time t . Although, with the possibility of creation/deletion operations, the set D_t could be variable, we assume that $D_t = D$ and is fixed.

Each entity X in the database may be replicated for availability reasons. The stored entity X_j is defined to be the j^{th} copy of entity X . Then the actual stored database

$$D_s = \{\dots, X_1, X_2, \dots, Y_1, Y_2, Y_3, \dots, Z_1, \dots\}$$

is a collection of all replications of each logical data item (that is, all stored data entities).

Since a database models some aspect of the real world we assume that there are assertions or logical constraints concerning a single entity or a group of entities which must be maintained in order to preserve the correctness of the database. For example,

Assume the database models a banking system with two entities X and Y which represent the savings account balance and the chequing account balance respectively. If customers have overdraft privileges on the chequing account but not on the savings account then the assertions to be maintained might be

$$A1: X + Y \geq 0 \quad A2: X \geq 0$$

One obvious assertion on a stored database is that for any stored entities X_i, X_j which are copies of logical entity X , we require $X_i = X_j$. From this assertion we can make the following observation.

Redundancy (having more than one stored entity for some logical entity)

does not, in itself complicate the mechanisms of maintaining assertions about the database. Rather it increases the number of assertions to be maintained. There is no inherent difference between maintaining the assertion $X + Y \geq 0$ and the assertion $X_1 = X_2$.

The replication of entities leads to many possibilities for the organization of stored entities. Let $N = \{N_1, N_2, \dots, N_k\}$ be the set of nodes in the DDB which maintain a local DB. Two special cases of organizing the stored database are:

1. centralized - where $|N| = 1$, and
2. distributed - where $|N| > 1$.

In a centralized system, all data entities are stored at the single database site and only one copy of each entity exists (although for reasons of efficiency and availability we may have replication even in a centralized scheme). In a distributed system we can identify three additional special cases:

1. fully replicated - each node N_i in N has a copy of every logical data entity;
2. fully partitioned - each node N_i in N has some subset of the entities in D but no two nodes have the same entities; and
3. partially replicated (general case) - each node N_i in N has some subset of the entities in D and each entity may be stored at more than one site. Some nodes may contain a copy of the complete database whereas some entities may not be replicated but only stored at one site.

The choosing of one type of data organization over another has been studied as the file allocation problem [MAHM76], [CHU73]. The DDB is modelled by access profiles for query and update at each node and by communication costs among each pair of nodes. At the two extremes we have the following organizations:

1. centralized. Updates are efficient since there exists only one copy of each entity to update. Since most users access the data remotely, queries may be inefficient due to the traffic involved in moving (perhaps large amounts of) data over communication links.
2. fully replicated. Queries are very efficient since all data is read locally. Updates are expensive since for each update every copy must be updated.

Without seeming to simplify the file allocation problem, which is often based on linear programming methods, the four cases may be organized as follows, based on access profiles:

centralized, fully partitioned, partially replicated, fully replicated

updates:100% <----- none
 queries:none -----> 100%

The concurrency control method should function independent of the organization scheme as far as correctness is concerned, however, performance characteristics will favour one organization over another for a specific application. The general partially replicated organization (of which each of the other organizations is a special case) is assumed.

2.2 Transactions

A transaction is the database equivalent of a macro-action and as such defines a level of correctness in terms of concurrent execution. Assuming that each transaction will maintain a consistent database state if executed alone on a centralized database, the concurrency control mechanism must ensure the same level of correctness in the concurrent environment. The only operations that affect the database correctness are read and write operations. Although reads do not immediately seem

to affect the database state, consider the following example:

Bank account with entities

X - savings balance Y - chequing balance Z - total balance,

the assertion A: $X+Y=Z$ and the following transactions

T1 : $X=X+10$; $Y=Y-10$	Transfer from account Y to account X
T2 : $Z=X+Y$	Compute total balance of accounts X and Y

which become

T1 : Read(X) ; Write(X gets $X+10$); Read(Y) ; Write(Y gets $Y-10$)
T2 : Read(X) ; Read(Y) ; Write(Z gets $X+Y$)

If either T1 or T2 is executed completely before the other, no assertion violation would occur, however, consider the following execution:

T2 : Read(X)
T1 : Read(X)
T1 : Read(Y)
T1 : Write(Y gets $Y-10$)
T2 : Read(Y)
T2 : Write(Z gets $X+Y$)

Clearly, after this concurrent execution, Z is not equal to the total of $X+Y$ either before or after T1. The reason is that the reads executed by T2 occurred both before and after some changes made by T1. So indeed inconsistent reads can affect database consistency. Although this example had T2 update the database, the result would be no better if T2 only reported the incorrect total to the user. Indeed, user output devices can be thought of as write-only entities [BERN78].

A transaction can be defined as a sequence of read and/or write operations on specific entities in the database. To better understand the functioning of a distributed transaction in terms of the individual operations we present an architecture for a local DB node in the DDB and then lead up to a specific transaction processing model.

2.3 Distributed DB Architecture

The DDB architecture within which we embed our local processing model is the ADD architecture [TOTH80] which comprises a four layer hierarchical structure at each node M_i in N (Figure 2.1). Requests for database access from the user progress through the various layers with processing occurring at each layer. Parts of the request may be routed to the various nodes involved to perform a distributed task by accessing local and/or remote stored entities. The four layers are listed below.

User View Processor (UVP) - provides user views tailored to the needs of particular applications since not all users will be allowed access or care to know about all the entities in D . Security and other access privilege checks are performed at this level.

Global Processor (GP) - presents the network-wide or global view of the database entities without the notion of distribution. This is what we have called our logical database D . This layer accepts transactions from the UVP and ensures that transactions are syntactically and semantically correct. Changes made at this level will generally necessitate change at other levels.

Distributed Processor (DP) - reflects the distribution and replication of the entities onto the various network nodes based on a file allocation analysis. User requests are mapped into atomic reads and writes on the stored entities, which can be executed at local and/or remote sites to achieve a distributed task. Query strategy coordination, concurrency control, and communication among DDB nodes is handled at this layer. A global directory of stored entities is maintained at this level and may be subject to change based on entity creation/deletion/migration.

Local Processor (LP) - implements a relational DBMS at the local nodes. Atomic operations from the DP are accepted and processed against the access structures built on the local file subsystem. Local consistency constraints are maintained at this level.

Each layer of the architecture will use well defined interfaces to cooperate with adjoining layers in processing user queries. We are concerned mostly with the functioning of the DP layer and its adjoining layers insofar as cooperation is required between them.

T₁ - a transaction which originates and is processed at the same site

T₂ - a transaction which originates at one site but is processed at a remote site

T₃ - a transaction which is processed both at a local site and a remote site

UVP - User View Processor

GP - Global Processor

DP - Distributed Processor

LP - Local Processor

— - transaction access path

- - - - transaction response path

CN - Communication Network

Figure 2.1 Processing Layers of the ADD Architecture [TOTH80]

2.4 Execution Flow

Requests are generated by a user process at the local site in the form of a query which may consist of one or more query statements expressed in the host data sublanguage. Each query statement may contain several database manipulation commands such as SELECT, UPDATE, JOIN, PROJECT (as in a relational algebra sublanguage) as well as some arithmetic operations. Examples of user processes which can generate queries are:

- a report-generating application program which consists of many query statements and perhaps executes in batch mode; and
- a user sitting at a terminal generating one query statement at a time and interactively receiving responses to each statement.

The query is validated by the GP for syntactic and semantic correctness against the logical data entities.

The three main functional modules of the DP layer are the Query Decomposition Process (QDP), the Transaction Control Process (TCP) and the Reliability Monitor (RM) (see Figure 2.2). The function of the QDP is to decompose queries from the GP into atomic operations on the logical entities. In the case of node and/or link failures, the RM handles restarts of transactions, atomic update and determines if continuation of a transaction is possible. The TCP manages overall distributed strategy and transaction synchronization. The combination of the above functions as part of a DP layer has been referred to as a Transaction Management Subsystem [ELMA81].

Each newly initiated transaction is placed under the control of a TCP which manages a private workspace for the transaction. The output of

Figure 2.2 Functional Modules of the DP Layer within ADD

the QDP in the form of reads and writes on the logical entities is transformed by the TCP into operations on the stored entities, reflecting the replication and distribution of the data. The sites at which the transaction will be executed are chosen such that communication costs and delay are minimized. Toth [TOTH80] presents strategies for efficient query processing. It is the function of the TCP to synchronize the reads and writes with those of other transactions and move intermediate results to and from the user workspace. The local DB in the LP layer executes the globally synchronized reads and writes using local locking procedures to maintain SR/EW mutual exclusion. On a read operation the TCP passes results to the user process via the GP and UVP layers.

Each transaction generated has a level of consistency (not be confused with degree of consistency [GRAY75]) which defines the range of read and write operations within the query which must be performed atomically or indivisibly as if the query were the only one executing. The user may explicitly define this level by bracketing the desired range of statements within a BEGIN-END or the QDP may enclose all query statements generated by a certain user within a single BEGIN-END. Therefore from the point of view of TCP a transaction is a BEGIN statement, followed by a sequence of read and/or write operations, followed by an END statement.

2.5 Transaction Transparency

One of the goals to be achieved in a DDB system is that of

transparency of the underlying architecture so that a user process can operate as if it were in a centralized environment (except for possibly slower response time). Transparency of this type has various aspects in a DDB [TRAI79]:

1. Location Transparency. Although data is geographically distributed and may move from place to place, the user process can act as though all data is located in one node.
2. Replication Transparency. Although the same entity may be replicated at several nodes in the network, the programmer may treat the entity as though it were stored as a single item at one node.
3. Concurrency Transparency. Although many transactions execute concurrently, it appears to each transaction as though it were the only activity in the system.
4. Failure Transparency. Either all or none of the effects of a transaction must be visible. Once a transaction is committed at any node it survives hardware and software failures.

In the ADD system, location and replication transparency are provided since the user deals with entities defined at the global or logical level (in the GP layer) just as in a centralized system. The TCP of the DP layer handles routing of atomic operations to the appropriate nodes by consulting the global directory. If an update is required on an entity, and more than one copy of the entity exists, one write operation on the entity is routed to each node involved. If a retrieval is requested on an entity, a read is sent to one site which holds a copy of that entity. The site is chosen based on communication and load considerations (a local copy will usually be chosen if one exists). Failure transparency will be discussed with respect to atomic commitment in Section 2.6. Concurrency transparency is, of course, the main topic of this thesis and methods implemented in the TCP will be discussed which assure correctness of operation in spite of concurrent

execution of transactions.

Before dealing with the transaction processing model we make one further observation concerning the architecture of a DDB. Until now we have been assuming that each DDB node implements all four layers of the architecture. From a functional point of view this is not necessary. We distinguish between the tasks of managing the user query (or transaction) and managing the accesses to the local DB. These two functions can be considered as a further refinement of the TCP module into:

1. a Master TCP for requests originating at the local QDP, which would manage transaction initiation, termination and restarts; and
2. a Slave TCP which cooperates with a (possibly remote) Master TCP by receiving atomic database operations and returning data and status information.

Now each DDB node can be redefined into two functionally distinct components [BER80a] (see Figure 2.3):

1. a Transaction Manager (TM) comprising the UVP and GP layers as well as the QDP, RM and Master TCP modules of the DP layer; and
2. a Data Manager (DM) comprising the LP layer and the Slave TCP module of the DP layer.

To manage a distributed query a local TM requires the cooperation of local and/or remote DMs. The DDB architecture is modified to require that each node consist of a TM, a DM or both a TM and a DM (Figure 2.4). Some nodes can handle user queries without requiring the existence of local data whereas some nodes can manage data, using specialized database machines, without provision for local access. The separation of functions allows flexibility in designing the organization of a DDB system with respect to user interface and DB processor locations.

Figure 2.3 Functional Separation of TM/DM within ADD

TM - Transaction Manager
 DM - Data Manager

Figure 2.4 Revised DDB Architecture

2.6 Transaction Processing Models

In this section a general model is developed by which the TM handles the atomic operations of a transaction. The input to the TCP consists of a BEGIN operation, followed by a sequence of logical READ and/or WRITE operations, followed by an END operation. We define a readset (writeset) for a transaction to be the set of logical entities which the transaction will access for a READ (WRITE) operation. The union of the readset and writeset is referred to as a read-writeset. If a transaction's read-writeset could be determined before any ~~execution~~ occurs it may be possible to optimize query execution. The user process may, however, be nondeterministic, making it difficult to predeclare resource requirements, for instance:

1. an application program issuing its requests based on the contents of past requests. For example,

```
READ(X)
if X = 1 then READ(Y) else READ(Z)
```

2. a user sitting at a terminal interactively issuing queries one at a time such that more than one query is part of the same logical transaction.

Even if predeclaration is possible, the estimated read-writeset [MILE80] would tend to be larger than the actual read-writeset required because the former must include all entities that might be accessed by a particular transaction.

Therefore, the transaction processing (TP) model and concurrency control methods are based on the general case of not being able to predeclare readsets and writesets. Thus, we assume an unspecified sequence of READ and WRITE operations as the output of the Decomposition

module. In the following presentation let TM_i (DM_i) represent the TM (DM) at node N_i . Let $WRITE(X, X')$ denote that the current value of X gets updated to X' . A general TP model follows:

TP Model 1

1. BEGIN indicates the start of the current transaction.
2. Whenever a $READ(X)$ is received, choose a site N_i in N at which a copy of X_i of X exists and retrieve the local value of X_i by sending DM_i a $dmread(X_i)$ message.
3. Whenever a $WRITE(X, X')$ is received, for every site N_i in N at which a copy of X_i of X exists send DM_i a $dmwrite(X_i, X')$ message.
4. END signifies that no further processing requests against the DB will be issued by the current transaction.

Notice that as far as concurrency control is concerned, we need to distinguish between logical operations, $READ(X)$ or $WRITE(X, X')$, and actual operations on the stored database, $dmread(X_i)$ or $dmwrite(X_i, X')$. It is the interleaving of the $dmreads$ and $dmwrites$ that can affect changes to the state of the database and must, therefore, be controlled by the concurrency control mechanism. The above TP model does not place any restrictions on the ordering of $dmread$ and $dmwrite$ operations within a transaction but the following discussions will point out the need to do so.

One issue of importance to the concurrency control problem is that of reliability. For various reasons transactions, nodes and communication links can fail causing inconsistency due to partially processed transactions. A full discussion of this issue is outside the scope of this thesis but some observations follow:

1. If a transaction which is performing an update fails after updating some but not all stored copies of all logical entities in

its writeset, the database will generally be left in an inconsistent state. A mechanism for atomic commitment or update is needed.

2. When a transaction is terminated (due to failure or restart) it is necessary to undo any changes the transaction may have made on the system as if it had never started processing. This requires a clean abort procedure. Once a transaction T1 has written new values into a database such that another transaction T2 can access the new data, T1 should not be "allowed" to fail or restart. If it does, both T1 and T2 must be aborted (since T2 read data which "never existed"). In fact, any transaction T3 that read any data written by T2 should also be aborted, etc. Rollback of many transactions is a complex problem in a distributed system because of incomplete system knowledge (about who read what from whom) and so we attempt to avoid it if possible. Rollback and abort can be differentiated in that ABORT is the termination of a transaction due to some failure situation whereas ROLLBACK is the process of ABORTing many transactions due to some data interdependence between them.

The general transaction processing model, TP1, does not handle the above-mentioned problems. To obtain a useful TP model we include mechanisms to deal with atomic commitment, and rollback. Recovery mechanisms to handle other types of failures have been discussed by many authors [HAMM80], [MENA79]. In particular, Laferriere [LAFE81] deals with recovery mechanisms for the Reliability Monitor in the ADD context.

The need for atomic commitment leads to the inclusion, in the TP model, of a technique called two-phase commit (not to be confused with two-phase locking). The technique ensures that in case of TM failure, either all or none of the updates are committed. Let TP2 be TP1 modified by replacing step 3 of TP1 by the following:

- 3.1a For each node N_i in N which contains a copy of X (call this set of DMS, DM^*), send DM_i a precommit(X_i, X') message.
- 3.1b As each DM_i in DM^* receives precommit(X_i, X'), the new value X' of X is written onto secure storage (a storage mechanism which can be made failsafe to a high degree) and an acknowledgement is sent to the originating TM.

3.2a Once an acknowledgement is received from each DMI in DM*, the TM sends a dmwrite(Xi) message to each DMI. If any DMI does not respond, the TM takes appropriate action based on its recovery policies, such as to abort the transaction or proceed with the reduced DM* set.

3.2b As each DMI receives the dmwrite message, it copies value X from secure storage onto the database copy Xi.

Upon TM failure all the DMs will choose a common action in dealing with the transaction in question (this assumes that each DMI knows who the other DMj in DM* are). If the TM fails, the DMI in DM* can confer and choose to either ABORT (if none has received a dmwrite) or COMMIT (if any dmwrite has been received) the transaction.

Note that two-phase commit does not fully solve the problem of atomic commitment. For example, if the only DM that received a dmwrite were to fail after the TM failed, the failing DM could COMMIT the updates whereas all the other DMs would ABORT the updates. Other mechanisms and recovery policies can be made to handle such situations to whatever degree of reliability (other than 100%) the system requires (depending on the overhead costs the system designer is willing to pay). The reason for considering 2PC in this discussion is because of its direct impact on the transaction processing model and the sequence of operations that directly access the stored database.

In TP2, each dmwrite is transformed in the way described above and in so doing each update is atomic being either globally committed or rejected. But because each WRITE operation in TP2 is handled independently, the effects of some write operations may be available to other transactions even though the former transaction can fail, leading to possibility of rollback. To deal with this problem, we use the

method of delaying the release of the effect of updates until the transaction terminates. If a failure occurs no other transaction will have accessed any of the delayed updates.

A delayed update policy can be implemented in various ways. Earlier we mentioned that one of the responsibilities of the master TCP module was the coordination of intermediate results in the workspace set aside for a particular user process. The private workspace can be employed to solve the rollback problem in the following way:

For each $WRITE(X, X')$ issued by a transaction T, rather than sending the precommits and dmwrite immediately to each DM in DM^* , the write can be performed on a copy of X in T's private workspace. After the END operation, a list of final updates could be compiled for each entity X updated by T and all precommit and dmwrites sent at that time. Now either all or none of the updates made by T would be committed using the 2PC mechanism.

The private workspace may be more accessible than any other stored copy for subsequent reads or writes on the same entity. The revised transaction processing model [BER80a] TP3 is as follows:

TP Model 3 (with two-phase commit and private workspace)

1. When transaction T issues a BEGIN, T's TM creates a private workspace (PW) for T. The location and organization of PW is a query processing issue and is not specified here.

2. When T issues a $READ(X)$ operation, TM checks

```
if a copy of X exists in PW
then READ(X) from PW
else select a stored copy Xi of X
      issue a dmread (Xi) to DMi
      DM copies value of Xi into T's PW
      READ(X) from PW
```

3. When T issues a $WRITE(X, X')$, TM checks

```
if no copy of X exists in PW
then create a copy of X in PW
X is updated to X'
```

4. When T issues an END, two-phase commit begins. For each X updated by T and for each stored copy X_i of X, TM issues a precommit (X_i, X') to DM1. Each DM responds by copying X' to secure storage and acknowledging T's TM. Once all acknowledgements are received, TM issues a dmwrite (X_i) for all copies of all logical entities updated by T. DM responds by copying X' from secure storage onto local stored copy X_i . Once all dmwrites are completed, the execution of T is complete.

In handling atomic commitment and rollback, TP3 has an interesting effect on the ordering of dmreads and dmwrites (with the accompanying precommits) on the stored database. Since all WRITE operations are initially only translated into writes on the private workspace, the effective ordering for concurrency control is that all dmreads are processed before any dmwrites. This may influence the functioning of a particular concurrency control mechanism. For example,

Under two-phase locking the precommit will serve to lock the entity in an exclusive mode. Since precommits are delayed the write locks are held for a shorter period of time than if they were claimed upon issuance of the WRITE(X, X') operation. A more subtle but less quantifiable effect is that delaying the write lock request may influence (increase or decrease?) the occurrence of deadlock.

Another effect of TP3 is that before any precommits are submitted, the writeset of each transaction is known. Writeset predeclaration is a requirement for the particular variation of timestamp ordering used in SDD-1.

The cost of restarts can be reduced by issuing resource requests as early as possible. Wasted processing is minimized since possible restarts are detected earlier. In terms of TP3, the precommits could be submitted upon issuance of the WRITE operation, whereas the updates are still delayed until after the END (dmwrite) operation. This variation has the restriction that each entity can only be updated once. Although

TP3 is used for presentation purposes in this thesis, precommits can be issued earlier if desired.

2.7 Issues In Correctness

The generally accepted correctness criteria of serializability [BER79a], [ESWA76], [ROSE78] states that a concurrent execution of a set of transactions preserves database consistency if the effect of the concurrent execution is equivalent to the effect of some serial execution of the same set of transactions. In this section we will examine the use of this correctness criteria and discuss the relationship of consistency in concurrency control to semantic integrity.

2.7.1 Semantic Integrity

Since the database models some aspect of the real world, we assume that explicitly or implicitly, there is a set of logical constraints or assertions which must be maintained to preserve correctness or integrity of the database. A correct database is usually thought of as one which accurately models the current state of the real world. The area of database research dealing with maintenance of database assertions is the study of semantic integrity (SI). An SI subsystem can validate and reject transactions which would violate the SI constraints. This validation can occur [BADA79] before (compile-time), during (run-time) or after (post-execution) transaction execution. Since SI validation involves partial execution of each transaction (in the sense of

computing update values) a run-time approach seems most suitable.

One run-time possibility might be to submit the update list from the private workspace to the SI subsystem for validation either before or after the first phase of the two-phase commit sequence. In order to make the two-phase commit an atomic procedure, the entities being updated would have to be locked after the precommit message is received at each DM.

An ideal SI constraint would only admit values that did not violate the integrity of the database. The SI constraint $X < 100$ defines a subset of allowable values when, in general, only one value of X is correct at any one time. A complete set of SI constraints would be large, dynamic and complex to specify so only a subset of the constraints are maintained explicitly. As such, SI constraints only provide a partial check against user transactions as an aid to eliminating some error and human dishonesty. For instance, the SI constraint $SALARY < \$40,000$ might prevent an incorrect assignment of say \$45,000 but it cannot prevent an employee from unintentionally or illegally raising his/her salary to \$35,000 from \$30,000. Prevention of this type of error (where a database value satisfies the SI constraints but does not reflect the intended value) involves other mechanisms such as security and authorization checks on the user as well as an assumption of an "honest and correct" user.

We define a database state that satisfies the explicit (and implicit) SI constraints to be consistent. From the point of view of concurrency control we make the following assumption concerning

transactions and consistency: if each transaction is given a consistent database as input and is run in an interference-free environment, then the transaction would produce a consistent database as output. In other words, in a nonconcurrent environment each transaction preserves the SI assertions. So the concurrency control mechanism then operates, independent of the semantic content of each transaction, to preserve consistency to the same degree as the transaction would have in a nonconcurrent environment.

Each transaction 'expects' that during its execution the data in its read-writeset will only be accessed by itself. Even though the database may be consistent before and after a transaction executes, there may be periods of time during the execution when the data being accessed is temporarily inconsistent and so no other transaction should attempt to read this data. This is why atomic actions are grouped into units of consistency called transactions. On the other hand, each transaction expects that at the time it tries to commit its updates, the read values on which the updates were based, should not have been changed (reproducibility of reads) [ESWA76].

One might ask whether the mechanism for atomic commitment of updates (so that either all or none of a transaction's effects on the database state are felt) is sufficient to guarantee consistency. To answer this we define two aspects of consistency [THOM78]. Mutual consistency means that all stored copies of each entity converge to the same state should all update activity cease in the system. Internal consistency is related to semantic integrity on a local copy of the database. Consider the following example:

Assertion: $X + Y + Z = 3$
Initial state: $X = Y = Z = 1$

T1: $X = X - 2$; $Y = Y + 2$

T2: $Y = Y - 2$; $Z = Z + 2$

Assume that both transactions T1 and T2 complete their read phase before either completes their write phase and both updates are applied. The assertion would then be violated and internal consistency lost even though atomic commitment would maintain mutual consistency by ensuring that both updates are applied in the same order at all nodes which contain entity Y. One of the updates must be rejected because it is based on information made obsolete by the update that gets accepted first. Therefore, atomic commitment ensures mutual consistency but does not guarantee internal consistency. A mechanism is needed which guarantees that the data being atomically committed is consistent.

Given our model of SI validation, one might also ask why we would not perform the read phase without concurrency control and then check if internal consistency would be violated before committing. The SI assertions are not usually explicit enough to ensure that the updates represent the intended values. The updates might leave an inconsistent database but not enough to call for a rejection on the grounds of explicit SI violation. As well, the approach depends solely on restarting transactions whereas some delaying of the transaction during the read phase may have lead to a consistent database. In general, we need to balance the amounts of blocking (delaying) and restarts for a given application in order to minimize wasted efforts and increase throughput.

2.7.2 Serializability

One approach to ensuring the consistency of a database is to remove the concurrency, and schedule all transactions serially. In other words, a new transaction does not begin until the previous transaction has completed execution. By invoking the consistency-preserving property of each transaction and applying some inductive reasoning we see that any serial execution of a set of transactions will produce a consistent database. We are, however, limiting throughput by not exploiting the fact that most transactions only use a small subset of the entities in the database. For example, in Figure 2.5(a), it is clear that transactions T1 and T2 could execute concurrently. The usual requirement is to control the concurrency so that any resulting schedule [ESWA76], log [BER79a] or execution is serializable.

Following a formalism similar to that used in [ESWA76] we assume a situation where all transaction atomic actions require exclusive access to each entity and each transaction only accesses each entity once.

Definition. For any set S and any irreflexive, transitive relation $*$ on S

- i) if for every S_i and S_j in S either $S_i * S_j$ or $S_j * S_i$ then $*$ defines a total order on S ; or
- ii) if there exists S_i and S_j in S such that neither $S_i * S_j$ nor $S_j * S_i$ is true then $*$ defines a partial order on S .

Definition. Given a set of transactions T , define T_e to be the subset of T containing transactions which accesses entity e .

Definition. For any T_i, T_j in T_e , $T_i <_e T_j$ iff T_i accesses entity e before T_j .

NOTE: Each block indicates the time during which a transaction has control over an entity.

Figure 2.5

Definition. For any T_i, T_j in T , $T_i < T_j$ iff $T_i <_e T_j$ for some entity e .

Examining the execution of a set of transactions T , the ordering produced by $<_e$ is a total ordering on T_e , whereas the ordering produced by $<$ is a partial ordering on T . In particular, if T_i and T_j do not access any common entities then neither $T_i < T_j$ nor $T_j < T_i$ is true.

Definition. The ordering $<$ produced by an execution of transactions in T is serializable iff $<$ is an acyclic ordering on T . In other words, there exists no $T_i, T_j, \dots, T_n$ such that $T_i < T_j < \dots < T_n < T_i$.

In an uncontrolled environment a nonserializable execution on T is possible. An acyclic $<$ relation can be seen as an ordering of transactions in T based on logical time, that is, $T_i < T_j$ if T_i executed before T_j at every node where they had an entity in common in their read-writesets. A serial execution produces an even stronger relation $<^*$ on T since a serial ordering is based on physical time. We define $T_i <^* T_j$ if T_j did not execute at any node until T_i completed execution. The goal of concurrency control mechanisms is to maximally exploit concurrency by producing an execution with a serializable logical ordering. The extension of $<_e$ to allow for shared-read mode of access is as follows: $T_i <_e T_j$ iff either T_i or T_j issues write on e and any other transaction T_k which issues a write on e does so before T_i or after T_j .

Figure 2.5 depicts some of these concepts. Figure 2.5(a) shows a serial ordering in physical time on transactions $T = \{T_1, T_2, \dots, T_6\}$. Figure 2.5(b) shows a possible serializable concurrent execution of T in physical time. Figure 2.5(c) shows the same concurrent execution of T in logical time.

Eswaran [ESWA76] defines the dependency relation, $DEP(S)$, as the ordering produced by an execution S of the transactions in T . For the example in Figure 2.5, $DEP(S) = \{T1 < cT3, T2 < dT3, T3 < dT6, T2 < eT4, T4 < eT6, T5 < fT6\}$.

Definition. Two executions $S1$ and $S2$ are equivalent if $DEP(S1) = DEP(S2)$.

Definition. A schedule $S1$ is consistent (or serializable) if it has an equivalent serial ordering.

The justification for this view of consistency is that the dependency relation defines the set of transactions from which each transaction reads its input values. In the serial execution, each transaction sees a consistent view of the database. If in the concurrent or interleaved execution, each transaction Ti reads its entities from updates produced by the same set of transactions as in the equivalent serial schedule, then Ti will perform the same computations and therefore produce the same output. So by induction, the interleaved execution will also produce a consistent database.

2.7.3 Two-phase locking and Timestamp Ordering

Most mechanisms for ensuring serializable executions are variations of two basic approaches, two-phase locking (2PL) and timestamp ordering (T/O). We will briefly consider both methods to see how they ensure serializability.

Two-phase locking is based on the explicit request, hold and release of locks on each entity. Eswaran [ESWA76] makes the following definitions:

1. A transaction T is well-formed if T locks each entity before using it and releases each entity before terminating.
2. A schedule S is legal if a transaction attempting to lock an already locked entity must wait.
3. A transaction T is two-phased if once the lock on any entity is released, T does not request new locks on any entities. Thus the transaction is divided into a growing phase during which new locks are requested and a shrinking phase during which locks are released.

The first two definitions require that transactions not violate the basic locking mechanism which can preserve shared read/exclusive write mutual exclusion.

Theorem. [ESWA76] If each transaction T_i in the set of transactions T is two-phased and well-formed then any legal schedule for T is consistent.

Two-phase locking ensures serializability since it requires that each transaction have a locked point [BER79a] at which the transaction holds locks on all entities it requires. This locked point guarantees the acyclicity of the $<$ relation on T as follows.

For any pair of two-phase transactions T_i and T_j , such that the writeset of T_i intersects the read-write set of T_j , both $T_i < T_j$ and $T_j < T_i$ cannot be true. $T_i < T_j$ implies that there exists an entity e such that $T_i <_e T_j$. Likewise $T_j < T_i$ implies that there exists an entity f such that $T_j <_f T_i$. If T_i is two-phased, then at its locked point T_i holds locks on both e and f . Therefore T_j is not two-phase since it released a lock on f and then acquired a lock on e . Similarly, if T_j is two-phase then T_i is not.

Many authors [BER80b], [THOM78], [MAHM79] have proposed concurrency

control methods based on timestamp ordering. Each transaction is assigned a system-wide unique timestamp, usually by appending a unique processor number to a local clock or counter. Serializability is maintained by guaranteeing that at every node where two transactions execute and conflict the transactions are executed in timestamp order. The timestamps are used to induce an a priori ordering on the set of transactions in T . Let $t(T_i)$ denote the timestamp for transaction T_i . Clearly $<$ is acyclic since $T_i < T_j$ implies that $t(T_i)$ is less than $t(T_j)$. We cannot have both $t(T_i)$ less than $t(T_j)$, and $t(T_j)$ less than $t(T_i)$. The timestamps themselves define the serial ordering to which the given execution will be equivalent. Concurrency can be maintained since the ordering restriction is performed on an entity by entity basis.

2.7.4 Increasing Concurrency

Eswaran proved that 2PL is a sufficient condition for consistency since it implies serializability. We can show that 2PL is not a necessary condition since some T/O-produced orderings are non-2PL and serializable. If serializability is not a necessary condition for consistency then perhaps other correctness criteria can provide greater concurrency of execution. Eswaran states that "it seems difficult to give nontrivial necessary conditions for all legal schedules for a set of transactions to be consistent."

It seems that nonserializable consistent schedules are dependent on the semantics of the transactions and the SI assertions of the system. For example, given the transactions

T1 : X = X + 2 ; Y = Y - 2
T2 : X = X + 5 ; Y = Y - 5

then the nonserializable schedule represented by

T1 < x T2 and T2 < y T1

is consistent due to the associativity of the operations in the transactions. Semantic based concurrency control is beyond the current state of the art.

If no SI assertions link the entities in the read-writeset of a transaction, then some actions could indeed be performed in a nonserializable execution. If this were the case we would not have one transaction but more than one. In discussing semantic integrity, a transaction was described as a unit of consistency such that if a second transaction were allowed to see or change intermediate results of the first transaction, then inconsistency would occur. This brings us to the question of how to define a transaction (that is, its range or level of consistency) in a DDB system. Do we allow the user to specify the BEGIN-END sequence or does the system automatically generate the BEGIN-END? Clearly if we could map the transaction into minimal transactions which would each maintain consistency we could increase concurrency. Researchers have either assumed that transactions would be specified by the user or that some automatic system of defining transactions would be used but no consideration has been given to guidelines as to how to do so. It seems that some research into this problem could provide benefits for increasing concurrency.

CHAPTER 3

REVIEW OF CURRENT PROPOSALS

The study of concurrency control for distributed database systems has been an active area of research since the mid-70s. Although a number of early survey papers described the general problems facing distributed database researchers [DEPP76],[ROTH77], the first survey paper to specifically deal with the concurrency control problem did not appear until 1979 [BER79b]. Many proposals have, however, been put forward. The recent work of Bernstein and Goodman [BER80a] at decomposing the concurrency control problem has done much to lead to a better understanding of the state of the field. The real deficiency in the research to date is a lack of conclusive, quantitative performance results comparing the various proposals.

In this chapter, we examine many of the proposals. Our approach is based on the classification found in [BER80a]. The various proposals are examined with respect to their use of restart and delay as synchronization tactics. Finally, an overview of the research in performance evaluation of concurrency control mechanisms is presented.

3.1 Basics of Concurrency Control Mechanisms

The many proposals in the literature are best understood in terms of their use of the basic tactics of delay and restart. When a transaction T is submitted in the form of dmreads, precommits and dmwrites, the local DM can respond in one of the following ways:

1. proceed - execution of the action is permitted,
2. wait - T is delayed (or blocked) until some condition is satisfied before being allowed to proceed, or
3. restart - execution of T would lead to a (possible) inconsistent database state or a deadlock has occurred and so T (or some other transaction with which T is in conflict) is forced to terminate and start over.

In the subsequent discussion we factor out the effect of waiting due to SR/EW locking since it is common to all concurrency control methods (see Figure 3.1(a)). Each method differs in how the DMS apply the above responses (to prevent deadlock and nonserializable executions), ranging from strictly delay (conservative T/O) to strictly restart (basic T/O) and many in-between options which use both (2PL for instance). An examination of the use of the above responses reveals characteristics of each protocol and can be used as part of a comparative examination of concurrency control schemes. Figure 3.1 gives a pseudocode sketch of each mechanism discussed in this chapter.

Consider a concurrency control mechanism which explicitly constructs the $<$ ordering relation or graph as discussed in Section 2.7.2. With only SR/EW locking, an acyclic $<$ ordering can occur. As each operation is granted access to an entity, perform the following test:

If executing an operation of transaction T_i would cause a cycle in the $<$ relation, then restart T_i .

For example, let $R_i(e)$ and $W_i(e)$ be a dmread and dmwrite operation, respectively, from transaction T_i on entity e . Let $R_1(e)$, $W_2(e)$, $W_2(f)$, $R_1(f)$ be an execution. The ordering graph $<$ would evolve as in Figure 3.2 after the execution of each of the above actions. According to the above test, $R_1(f)$ caused a cycle in the $<$ graph and so T_1 is restarted.

```

while entity locked
    wait
endwhile
lock(entity)
execute Ti
release(entity)

```

a) SR/EW locking

```

while entity locked
    wait
endwhile
lock(entity)
if <-graph is acyclic
then restart Ti
    release(entity)
else execute Ti
    add edge/node to <-graph
    release(entity)

```

b) Certifier

```

while entity locked
    wait
    if deadlock then restart
endwhile
lock(entity)
execute Ti
wait_for_locked_point(Ti)
release(entity)

```

c) 2PL/Deadlock Detection

```

if entity locked {by Tj}
then if t(Ti) < t(Tj)
    then restart Tj
    else while entity locked
        wait
    endwhile
    lock(entity)
    execute Ti
    wait_for_locked_point
        of(Ti)
    release(entity)

```

d) 2PL/Deadlock Prevention
(Wound-Wait)

```

if t(Ti) < t(entity)
then restart Ti
else while entity locked
    wait
    endwhile
    lock(entity)
    execute Ti
    t(entity) := t(Ti)
    release(entity)

```

e) Basic T/O

```

for each TM
    repeat
        wait
    until t(Ti) < t(TM)
    while entity locked
        wait
    endwhile
    lock(entity)
    execute Ti
    release(entity)

```

f) Conservative T/O

LEGEND: t(T_i) - timestamp of transaction T_i
 t(entity) - timestamp of the entity
 t(TM) - largest timestamp of any transaction from TM

NOTE: for further details consult text on each proposal

Figure 3.1 Pseudocode Sketches of the Proposals Discussed

Figure 3.2 Evolution of a $\prec$ graph for a Concurrent Execution

The basic approach of the above protocol underlies a class of algorithms called certification algorithms or certifiers (see Figure 3.1(b)) [BADA79], [BAYE80], [CASA79]. Except for delay due SR/EW locking, certifiers rely strictly on restarts as a synchronization tactic. Restart can be more expensive tactic than delay for the following reasons:

1. increased message traffic to resubmit a transaction and notify previously visited nodes,
2. wasted processing due to termination of current activity, and
3. restart involves an element of delay as transactions try to return to their state of execution just before restart.

The issue of balancing restarts and delay will be addressed further in Chapter 5 with respect to performance evaluation.

3.2 Two-Phase Locking

Two-phase locking (2PL) is a concurrency control based on waiting. As described in Section 2.7.3, each transaction acquires locks on its required entities and holds them through its locked point. The transaction does not release any entity until after it has acquired all entities it will eventually need. Thus the 2PL policy is managed cooperatively by the transaction (through its TM) following a two-phased locking policy and by the DM adhering to SR/EW locks. Since the 2PL condition ensures serializability, it is not necessary to explicitly maintain the $<$ graph.

The 2PL algorithm is subject to deadlock and thus requires a deadlock detection or deadlock prevention mechanism. Detection involves a method for detecting deadlocks when they occur and preempting one of

the transactions involved, in order to break the deadlock. Prevention imposes extra conditions on the acceptance of a transaction in order to guarantee that deadlock cannot occur.

3.2.1 Deadlock Detection (2PL/DD)

Deadlock detection has been studied by many researchers in the field of operating systems and databases [GRAY78], [MENA79], and involves constructing and checking for cycles (see Figure 3.1(c)) in a WAITS-FOR graph [HOLT72]. The WAITS-FOR graph is similar to our < graph except that the edge $T_i \leftarrow e T_j$ is added to the WAITS-FOR graph if T_j attempts to execute on entity e and is made to WAIT for T_i which currently has e locked. Edges and nodes are deleted from the graph once a transaction terminates, since the WAITS-FOR graph is not used to maintain serializability as in the certifiers approach. Two issues that arise in deadlock detection are location of the WAITS-FOR graph and the timeliness of the detection algorithm.

Detection schemes can be centralized such that one node is chosen to maintain a global WAITS-FOR graph [STON79], [GRAY78]. The approach entails additional communication costs (above the concurrency control protocol itself) and the centralized component subjects the distributed system to reliability and throughput (bottleneck) concerns. A distributed approach would allow each node to break local deadlocks based on local knowledge. A global detector is still needed for those deadlocks that cannot be detected locally. The distributed scheme can be extended to a hierarchical detection approach [MENA79]. The database

sites are organized into a hierarchy according to either physical proximity or logical access (those likely to be involved in the same deadlock should be in the same subtree) and any unresolved WAITS-FOR information is sent to the parent node for resolution.

As both schemes involve the periodic transmission of information to a global detector, the question of how often to send WAITS-FOR information arises. Sending the information immediately as each WAIT exists may be costly in terms of communication and unnecessary. For example, consider the following execution $W1(e)$, $W2(e)$, $W1(f)$, $W2(f)$. The nodes holding entities e and f will generate the information $T1 < e$ $T2$ and $T1 < f$ $T2$. Clearly no local deadlock occurs and after some time, $T1$ will release e and f to $T2$ so that any message sent to the global detector are wasted. In order to not send unnecessary messages the WAITS-FOR information can be sent after a time delay. If a deadlock does exist, however, the transactions involved are delayed and the entities being locked are blocked from active use for other transactions. Therefore, the detection algorithm must be tuned to trade off timely restart of deadlocks against communication costs.

3.2.2 Deadlock Prevention (2PL/DP)

2PL was described as a concurrency control method based strictly on waiting. The reason for this view is that in response to a transaction's request for access to an already locked entity, the DM always responds to the conflict by allowing the transaction which caused the conflict to WAIT. RESTART is only initiated as a remedy to an actual deadlock. Algorithms have been proposed which prevent deadlocks from occurring by

allowing each DM to choose either WAIT or RESTART in response to a conflict situation. Deadlock prevention schemes trade off the cost of deadlock detection overhead for increased restarts. Whereas detection schemes only preempt a transaction when an actual deadlock occurs, prevention schemes can restart a transaction whenever a deadlock might occur. The problem of how to regulate the use of WAIT and RESTART responses (based on timestamps) such that waiting forever (deadlock) and restart forever are prevented is examined in [ROSE78]. The following is a summary of their results:

Definition. A transaction T_i is said to cause a conflict with transaction T_j if T_i requests an entity already requested (even if not yet granted) by T_j .

Definition. A concurrency control is superstrict if a transaction T_i can only WAIT for transactions T_j that it caused a conflict with. Therefore, all transactions are processed strictly in received order.

Theorem. If a concurrency control is superstrict the only waiting forever is caused by deadlock.

Definition. If a transaction T_i is assigned a unique timestamp $t(T_i)$ which is retained even if the transaction is restarted, then the concurrency control is said to be based on a valid numbering scheme. A property of valid numbering schemes is that there are only a finite number of transactions which have a smaller (older) timestamp.

Definition. If a transaction is RESTARTED only when it is in conflict with an older process, then the concurrency control is a valid numbering method.

Theorem. If concurrency control is superstrict, is not subject to deadlock and has restart based on a valid numbering method, then every process will terminate.

Two algorithms were formulated which regulate the use of WAIT and

RESTART by the DM in a 2PL environment. Timestamps are used, only to regulate the ordering of transactions that are active and conflict. It is not necessary that transactions be processed in strictly timestamp order.

Assume that T_i issues a request causing a conflict with T_j . Let $t(T_i)$ and $t(T_j)$ be the unique timestamps assigned to T_i and T_j , respectively, by a valid numbering scheme. The algorithms followed by the DM are:

WAIT-DIE (a nonpreemptive algorithm)

if $t(T_i) < t(T_j)$ then WAIT else RESTART T_i

that is, if requestor is older then requestor waits else requestor dies.

WOUND-WAIT (a preemptive algorithm, see Figure 3.1(d))

if $t(T_i) < t(T_j)$ then RESTART T_j else WAIT

that is, if requestor is older then wound T_j else requestor waits.

Although both satisfy the conditions of the second theorem, their behaviour is quite different. WOUND-WAIT tends to serialize transactions in timestamp order whereas WAIT-DIE tends to serialize transactions in reverse timestamp order.

Consider a modified WOUND-WAIT algorithm where T_i causes a conflict with T_j only if T_j has already locked the entity in conflict. Although the modified algorithm is subject to waiting forever (but not due to deadlock) it will induce fewer restarts than the original WOUND-WAIT. The writer feels that events which could cause waiting forever (other than deadlock) are unlikely to occur.

3.2.3 Variations on 2PL

This section will briefly describe some of the major variations of 2PL. One variation is the basic 2PL algorithm in which a 2PL scheduler is distributed amongst every DM in the system. To perform a READ(e) the TM sends a dmread(e) to any DM holding entity e. The dmread implicitly acts as a read-lock (r-lock) request on e. To perform a WRITE(e) the TM sends a precommit(e) to every DM holding entity e. The precommit implicitly acts as a write-lock (w-lock) request on e. All locks are released during the second phase of the two-phase commit procedure. Thus, dmwrite(e) acts as the w-lock release for entity e. Explicit r-lock releases are needed but can be piggy-backed with w-lock releases at appropriate DM's. Basic 2PL is the approach proposed for SYSTEM R* [LIND80].

Centralized 2PL is a variation whereby the database is distributed but the lock controller is centralized at one DM, say DM*. To READ(e), the TM sends to DM* a dmread(e) which acts as an implicit r-lock request on e. The dmread(e) is forwarded to DMj once the lock is granted. DMj executes the request and responds to the TM. To WRITE(e), the TM sends a precommit(e) to DM* which acts as w-lock request on e. Once the lock request is granted, precommit(e) is forwarded to every DM containing e. A dmwrite(e) is sent by TM directly to each DM involved. DM* is notified in order to release all locks as soon as one of the DMs has correctly committed the transaction.

The centralized approach clearly requires more synchronization messages than the distributed approach, but it has advantages in terms

of deadlock occurrence and deadlock detection cost. A centralized lock controller implies that no WAITS-FOR messages are needed since DM* automatically has the WAITS-FOR information. Detection therefore becomes less costly, and timely detection of restarts will decrease detection delay over the distributed approach.

Another advantage of centralized 2PL is that a form of deadlock called multiple-copy write (MCW) deadlock cannot occur. Consider the following example W1(e1), W2(e1), W2(e2), W1(e2) where e1 and e2 are copies of the same entity. In basic 2PL a deadlock occurs because each DM maintains separate lock control on their copy of the entity. Centralized 2PL should incur fewer deadlocks since only one lock request is issued for each action on an entity and MCW deadlock cannot occur. Centralized 2PL has been suggested in [MENA78], [ALSB76], [GARC78].

Primary copy 2PL is a compromise between basic 2PL and centralized 2PL. Rather than having a single DM manage the global lock table, one DM is appointed the primary copy for each entity e and manages lock requests and releases on e. Let DM* be the primary copy node for entity e. To READ(e) from DMi the TM sends a dmread(e) to DM* which processes the lock request and sends the dmread(e) to DMi. To WRITE(e) a precommit(e) is sent to every DM holding e but only acts as a lock request at DM*. By the time all ACKs are received by the TM, the entity has been locked at DM*. The dmwrite(e) is sent to every DM but only acts as a lock release at DM*.

Compared with basic 2PL, reads incur more messages in primary copy 2PL but writes create fewer deadlocks (no MCW deadlock). Also, fewer

WAITS-FOR messages need be sent since only the primary copy of each entity reports to the deadlock detector. This is the approach proposed for the distributed INGRES system [STON79]. Stonebraker also presented some variations that allow for faster reads, by accessing the local rather than primary copy of an entity, but which might not preserve consistency in some cases.

The variations in 2PL techniques trade off:

1. the ability to access local copies for reads;
2. deadlock detection occurrence and overhead; and
3. throughput, bottleneck and reliability considerations due to increased dependence on certain copies such as centralized or primary copies and required backups.

3.3 Timestamp Ordering

Timestamps were used as a deadlock prevention scheme in a 2PL environment in Section 3.2.2. In this section the use of timestamp ordering (T/O) is presented as a deadlock-free concurrency control in a non-2PL environment. T/O schemes can be classified as basic T/O or conservative T/O depending on whether restart or delay is the dominant synchronization tactic. Some variations on the T/O schemes will also be discussed.

3.3.1 Basic Timestamp Ordering

Basic T/O is a concurrency control based on restarts. The difference between basic T/O and certifiers is analagous to the difference between deadlock prevention and deadlock detection. Whereas

a certification algorithm only restarts a transaction when an actual acyclic < ordering is produced, basic T/O can restart a transaction when a nonserializable ordering might be produced. This is due to the fact that certifiers maintain an explicit global < graph whereas basic T/O acts only on local knowledge by forcing transaction to execute in timestamp order to produce an acyclic < ordering.

Basic T/O is the mechanism used in the ADD protocol [MAHM79]. For each stored entity e , we maintain an entity timestamp $TS(e)$ which represents the timestamp of the last transaction to access e . Let $t(T_i)$ be the timestamp of T_i . The DM uses the following acceptance rule for each action of T_i accessing entity e (see also Figure 3.1(e)):

```

if  $t(T_i) \geq TS(e)$ 
  then process action on  $e$ ;  $TS(e) = t(T_i)$ 
  else RESTART  $T_i$ 

```

(that is, if younger then WAIT else RESTART)

When T_i is restarted a new (larger) timestamp needs to be assigned; therefore basic T/O is not a valid numbering scheme according to [ROSE78], and so is subject to cyclic-restart. The performance aspects of the basic T/O mechanism are discussed in Chapter 5.

As stated above, the algorithm disallows the following serializable execution of T_i and T_j where $t(T_i) < t(T_j)$: $R_j(e), R_i(e)$. It does not have enough resolution to detect harmless read-read conflicts. A modified basic T/O algorithm which maintains two timestamps for each entity is discussed in [BER80a]. Let $R-TS(e)$ and $W-TS(e)$ represent the largest timestamp of any transaction which reads or writes e respectively. The DM now uses the following acceptance rules

for processing dmread(e) and dmwrite(e) from Ti:

```
dmread(e)    if t(Ti) > W-TS(e)
               then dmread(e); R-TS(e)=MAX[R-TS(e), t(Ti)]
               else RESTART Ti

dmwrite(e)    if t(Ti) > R-TS(e) and t(Ti) > W-TS(e)
               then dmwrite(e); W-TS(e)=t(Ti)
               else RESTART Ti
```

At the cost of twice as many timestamps, out of order reads are allowed, thus reducing the number of restarts.

Another optimization, called the Thomas Write Rule (TWR), can now be incorporated into the algorithm. Thomas [THOM78] first observed that if two writes are received out of timestamp order, discarding the second write (by not processing it) has the same effect as executing them in timestamp order. The new dmwrite acceptance rule becomes:

```
if t(Ti) > R-TS(e)
then if t(Ti) > W-TS(e)
    then dmwrite(e); W-TS(e)=t(Ti)
else RESTART Ti
```

TWR may be desirable in a high conflict environment to improve the restart behaviour (by not rejecting out of order dmwrites) under the conditions that the timestamp granularity must be the same as the entity granularity. Until now this is assumed to have been the case. It is possible, however, to have one timestamp for a group of entities rather than a single entity. For example, the entity granularity may be a tuple in a relational DBMS whereas the timestamp granularity may be at the relation level. For performance reasons it may be desirable to trade off storage costs for increased blocking and restarts.

3.3.2 Conservative Timestamp Ordering

Conservative T/O schemes eliminate restart by delaying transactions until all transactions with earlier timestamps have been processed (Figure 3.1(f)). Thus in conservative T/O a transaction experiences two forms of waiting: waiting for acceptance due to global synchronization and waiting for processing due to local locking policy. The time a transaction waits before acceptance is defined in [KAM081] as its eligibility waiting time. In this section we discuss two conservative proposals: the ticketing algorithm [LELA77], [LELA78] and the SDD-1 algorithm [BER80b]. The two approaches differ in the way timestamps are generated and used in synchronization. Conservative T/O schemes assume predeclaration of read-writesets.

The ticketing algorithm assumes that all timestamps are generated as monotonically increasing, consecutive integers. This is enforced by connecting all the TM nodes on a virtual ring, independent of network topology and circulating a token or counter containing the next available timestamp among the TMs. As each TM receives the token, it removes a ticket and increments the counter for each pending transaction and then passes the token onto the next TM on the ring. Each transaction needs to be sent to every DM, either for actual processing or to transmit timestamp information. Thus each DM knows that given a transaction with timestamp $t(T_i)$ the set of transactions with earlier timestamps are precisely those transactions

$$\{T_j \text{ such that } t(T_j) = t(T_i) - k \text{ for } k = 1, 2, 3, 4, \dots\}.$$

Let TS be the timestamp of the last transaction to be accepted at the DM. The DM can use the following acceptance rule when it receives T_i :

if $t(T_i) = TS + 1$ then accept T_i else WAIT

Transactions which had been waiting may become eligible for processing due to the acceptance of T_i and so are removed from the eligibility waiting queue and are placed onto the local processing queue.

The SDD-1 approach does not require timestamps to be consecutive but, only monotonically increasing at each TM. A further assumption is that messages are sent by each TM and received by each DM in timestamp order (called pipelining in [BER80b]). Class pipelining enforces a serial ordering on the submission of transactions by a TM. At each DM define $t(TM_j)$ for each TM_j in the system as the largest timestamp received from TM_j . The DM acceptance rule becomes:

if $t(T_i) < t(TM_j)$ for each TM_j
then accept T_i
else T_i WAITS

In other words, T_i is made to wait until a transaction with a larger timestamp is received from each TM. Whereas in the ticketing algorithm the DM could decide whether a transaction could be accepted based on current knowledge, in the SDD-1 approach the DM requires "proof" from each TM that no further transaction with smaller timestamps than T_i will arrive.

Bernstein points out some drawbacks of the SDD-1 approach:

1. A transaction can be delayed indefinitely if a TM does not submit any new transactions.
2. The implementation requires all TMs to communicate with all DMs regularly since it serializes all transactions in timestamp order not just those that conflict.

To remedy the first problem, null transactions (containing only

timestamp information) can be sent from time to time by low traffic TMs or can be requested by impatient DMs. The second problem leads to the concept of transaction classes which assumes predeclaration of read-writesets. A class is defined by a read-writeset. A transaction T is a member of a class if T's read-writeset is a subset of the class' read-writeset. Class definitions are defined at database design time by the Database Administrator (DBA) and are not expected to change frequently.

Since each transaction is managed by the TM which manages the transaction's class, a transaction is forwarded to the appropriate TM by the originating TM. Now when a DM receives transaction T it need only wait for larger timestamps from those TMs whose classes conflict with T, thus reducing delay and communication costs. Conflict graph analysis is another method for improving performance of conservative T/O with classes. Through preanalysis of transaction classes, it can be determined which class conflicts need not be synchronized and what optimization is possible for conflicts that do need synchronization. Mohan [MOHA79] points out that not enough attention has been given to developing guidelines for class definitions by the DBA as a poor class design could severely reduce the system throughput.

The actual SDD-1 concurrency control uses a variation of conservative T/O with transaction classes and conflict graph analysis. The dmreads are forced to wait for dmwrites with larger timestamps from each conflicting class but the dmwrites need not wait for dmreads. Dmreads can thus be rejected if a dmwrite with a larger timestamp has already been processed; a timestamped write data field, W-TS(e), is needed as in basic T/O.

3.3.3 Variations on Timestamp Ordering

Various timestamp ordering proposals have been suggested for DDB with fully replicated DB nodes. Milenkovic [MILE80] uses full replication to ensure that a timestamp $t(T_i)$ is chosen such that no rejections can occur. Since each transaction must update at every node, T_j must wait for T_i if $t(T_j) > t(T_i)$ and so T_j cannot cause a rejection of T_i . Acceptance at each DM is based on explicit ACK messages rather than on subsequent transaction requests as in SDD-1.

Thomas [THOM78] uses full replication to perform the read phase locally (assuming predeclaration) and then submits the update list to all the DMs. Each data entity is timestamped by the last transaction to update that entity. A majority consensus voting procedure takes place to determine whether the information upon which the update list was based has been made obsolete by subsequent updates. Thomas is credited with introducing the concept of timestamping data entities and transactions for synchronization and the Thomas Write Rule mentioned in Section 3.3.1.

Cheng [CHEN81] also assumes that the read phase is performed locally. The need for timestamped data entities is eliminated by relying on explicit ACKs to determine when a transaction can be accepted. If any conflict exists only the transaction with the lowest timestamp is accepted and all others are rejected.

3.4 Performance Evaluation of Concurrency Control

The previous three sections familiarized the reader with the many proposals made to date in the literature. It will have been noted that many of the proposals overlap in the use of different tactics, making a qualitative comparison difficult. A further complication is the fact that specific details of implementation, which could affect the behaviour of each proposal, were not considered. In fact, many authors have analyzed their proposals qualitatively leaving the field of concurrency control with many contradictory claims.

Quantitative performance analysis can provide insight into the operation of the various concurrency control methods. Contradictory claims still remain, however, due to the differing assumptions in the analyses. In this section an overview of the research to date is presented.

Garcia-Molina [GARC78] used simulation to compare centralized 2PL and the distributed majority consensus algorithm [THOM78] assuming predeclaration, full redundancy and optimistic environments such that few run-time conflicts occurred. He concluded that the centralized 2PL algorithm outperformed the distributed algorithm under the tested environments.

Ries [RIES79] compares centralized 2PL, basic 2PL with deadlock prevention and basic 2PL with centralized deadlock detection under an optimistic environment using simulation. He concludes that all the tested algorithms exhibited similar behaviour.

Cheng [CHEN81] compares centralized 2PL and various distributed conservative T/O algorithms using simulation. His conclusions are that distributed schemes are a viable alternative to the centralized scheme and that, under a variety of operating conditions, different distributed algorithms performed better. The centralized scheme required a backup node for reliability reasons. It also suffered from saturation at the backup node since all messages sent to the central DM had to also be sent to the backup node. This caused response time for the centralized system to increase more rapidly under increasing load than for the distributed schemes.

Centralized 2PL, the majority consensus algorithm and a variation of conservative T/O based on logical clocks are compared in [KANE79]. Their simulation considered response time as dependent on communication delay and ignored CPU queues, I/O-queues and conflict among requests. The principal conclusion drawn was that the distributed schemes were better than centralized scheme. Conservative T/O was also verified as having higher relative costs for low arrival rates than for high arrival rates. (Remember that for low arrival rates conservative T/O depends on explicit acknowledgements whereas under higher traffic this function is performed by the requests implicitly.)

The effect of locking granularities under 2PL with predeclaration is examined in [POTI80] using a hierarchical multiprogramming analysis. They show that as the number of granules increases the throughput increases but through a minimum. The issue of granularity is also important for T/O schemes since a design can trade off storage requirements of queues/timestamps against restarts/delays.

A "straw man" analysis of the probabilities of waiting and deadlock in 2PL systems is presented in [GRAY81] using a simple mean-value analysis. Their objective is to try to develop a model to explain their observations that in several DBMSs, transaction waiting and deadlock are very rare. A simulation model was developed under the following parameters:

1. R entities in the system, where R is in the range one million to one billion;
2. N transactions, where N is between 1 and 100; and
3. read-writeset of size r is randomly chosen from R entities, where r is in the range of 1 to 100.

The simulation results supported their observations that transaction waits were rare, deadlock cycles were mostly of length two and both the probability of waits and deadlock increased proportional to N. The authors conclude that a WAITS-FOR graph would be very sparse and hence, argue against using a centralized deadlock detector.

A quite different approach has been adopted by the designers of DDTS [ELMA81] which is a testbed for the experimental measure of DDB systems. The researchers propose a modular implementation which will support and test a variety of concurrency control mechanisms (as well as data models, recovery mechanisms, file allocation schemes and user interfaces). Initially DDTS will employ a concurrency control method using basic 2PL with deadlock detection and basic 2PL with deadlock prevention. The second phase will employ T/O algorithms.

Queueing analysis of the disorderings that can occur due to variable network delays is presented in [KAMO81]. The ticketing implementation [LELA78] of conservative T/O is examined assuming an

M/M/ ∞ network. Results are developed for the probability of eligibility waiting and average waiting times from which can be derived the average time to acceptance for a transaction. In Chapter 5 we present a similar analysis for basic T/O and compare it to conservative T/O under a variety of conditions using simulation models.

CHAPTER 4

THE ADD AND ADD* PROTOCOLS

In this chapter the ADD implementation [MAHM79] of basic T/O will be investigated using the formal protocol analysis tools outlined in Section 1.3. Programming language and finite state machine techniques will be used to specify the protocol and assertion proving will be used to demonstrate correctness and examine its restart behaviour. In the ADD protocol each transaction is assumed to predeclare its read-writeset. An extension to handle the general (nonpredeclarative) case will be presented. The new protocol, ADD*, will also be analyzed using specification and verification techniques. The handling of restarts in ADD* is described and some suggestions for reducing the occurrence of restarts are presented. Hybrid basic T/O and conservative T/O schemes, which have characteristics more desirable for some environments than the pure T/O schemes, are discussed.

4.1 Specification of the ADD Algorithm

As outlined in Sections 2.4 and 2.5, the DM is composed of:

1. the slave Transaction Control Process of the DP layer; and
2. the LP layer which handles local SR/EW locking and execution of atomic operations on the entities of the local DB.

In the ADD protocol the above two functions are handled by a Global Coordinator (GC) and a Local Coordinator (LC), respectively, (Figure 4.1) which separate the synchronization and execution of a transaction into two distinct phases.

When a TM receives a transaction T_i , a master Transaction Control Process (TCP_i) is assigned to handle the synchronization of T_i . At each participating DM, a slave TCP_i is invoked which cooperates with the master TCP and performs the Global Coordinator function to help synchronize T_i . The slave TCP receives local and remote requests that require synchronization due to multi-node access and accepts or rejects each transaction based on the basic T/O criteria (Section 3.3). If a transaction is rejected, a message is sent to the master TCP which will resubmit the transaction with a new timestamp. If accepted, the transaction is placed into the Resource Access Table (RAT) which is a set of reservation queues for those transactions awaiting global acceptance (Figure 4.2). Transactions are only removed from the RAT in timestamp order so that a transaction that has been accepted cannot be subsequently rejected.

In the ADD protocol each transaction must be globally accepted, by receiving local acceptance from each participating DM, before being passed to the LC for execution. (Note that the expression "to process" a transaction T_i , is used to denote the action of removing T_i from the RAT and passing it to the LC once it has been globally accepted). The LC merges transactions that have been synchronized with those local, single-site-access transactions not requiring global synchronization. Since the GC processes transactions in timestamp order, local SR/EW locking mechanisms by the LC are sufficient to ensure internal consistency. In Section 4.2 we will prove that mutual consistency is preserved and that deadlocks cannot occur.

Figure 4.1 Structure of ADD Protocol

Figure 4.2 Structure of Global Coordinator

In presenting the algorithm in Figures 4.3, 4.4, 4.5, and in the subsequent discussion, the following notation is used:

Let $E = \{E_1, E_2, E_3, \dots\}$ be the set of entities maintained by the DDB;

$N = \{DM_1, DM_2, DM_3, \dots\}$ be the set of DM nodes;

$E(T_i)$ be the set of entities in the read-writeset of T_i ;

$DM(T_i)$ be the set of nodes which TM accesses to execute T_i ;

$N(T_i)$ be the cardinality of $DM(T_i)$;

$t(T_i)$ be the timestamp assigned to T_i by TM;

$t'(T_i)$ be the updated timestamp for rejected transaction T_i ;

$t(E_i)$ be the timestamp of the last transaction to be processed against E_i by the GC at some DM;

RAT_j be the resource access table at DM_j composed of a queue Q_k and an entity timestamp $t(E_k)$ for each entity E_k ;

$count(T_i)$ be the number of nodes in $N(T_i)$ which have locally accepted T_i ; and

$X_j(T_i)$ be the union of T_i and the set of transactions which conflict with T_i at DM_j . T_i and T_k conflict at DM_j if both are in RAT_j and the intersection of $E(T_i)$ and $E(T_j)$ is nonempty.

The following are the messages exchanged between the master TCP at TM_j and the slave TCP at DM_k to synchronize transaction T_i :

REQUEST $\{ \langle T_i, t(T_i) \rangle, from_j, \langle misc \rangle \}$...issued by TM to each DM in $DM(T_i)$,

ACCEPT $\{ \langle T_i, t(T_i) \rangle, from_k \}$...indicates local acceptance by DM_k ,

REJECT $\{ \langle T_i, t(T_i) \rangle, from_k \}$...indicates local rejection by DM_k ,

TSUPDATE $\{ \langle T_i, t(T_i) \rangle, from_j \}$...issued by TM, if any DM rejects T_i , to inform slave TCPs of new timestamp,

PROCESS $\{ \langle T_i, t(T_i) \rangle, from_j \}$...indicates global acceptance to all DM in $DM(T_i)$,

where

$\langle T_i, t(T_i) \rangle$ constitutes a unique identifier for each transaction (necessary to prevent sequencing anomalies due to different incarnations of rejected transactions);

from1 is the originating node of a message; and
<misc> contains information such as the read-writeset of T_i.

The algorithms will be presented using a Pascal-like pseudocode. Procedure MASTER (Figure 4.3) represents the master TCP for T_i at some TM. Procedure SLAVE (Figure 4.4) represents the slave TCP for T_i at some DM. The procedures which manipulate the data structure (for purposes of testing entity timestamps, updating transaction timestamps and entering or removing transactions) are presented in Figure 4.5. The operation of algorithm SLAVE is sensitive to the concurrent arrival and departure of transactions from the Resource Access Table. Therefore the RAT data structure is described as a monitor [BRIN73] such that only one of the critical monitor procedures (evaluate, update or process) can be manipulating the RAT at any time.

Procedure 'enter' places transactions into the appropriate queues in timestamp, rather than FIFO, order. In the monitor procedure 'process', the await operation causes the slave TCP to be placed in a blocked state until the transaction becomes the oldest in its conflict set X(T). This is required to prevent the rejection of transactions which have already been accepted. The wait_receive and wait_message procedures will cause the calling TCP to be blocked until the appropriate transaction or message has arrived.

If a transaction gets rejected at some DM, the master TCP must update the transaction timestamp and send the rejecting slave a new REQUEST message. All other slave TCPs will receive a timestamp update (TSUPDATE) message. The rejecting slave therefore performs 'exit' and

```

procedure MASTER {at TMi}
begin
    wait to receive(Ti)
    t(Ti) := assign_timestamp
    DM(Ti) := determine_node_set
    for each DMk in DM(Ti)
        send REQUEST{<Ti,t(Ti)>,fromi,<misc>}
    count(Ti) := 0
    repeat
        wait_message(reply)
        case reply of
            ACCEPT: count(Ti) := count(Ti) + 1
            REJECT: t'(Ti) := assign_timestamp
                    DM* := fromj
                    send DM* REQUEST{<Ti,t'(Ti)>,fromi,<misc>}
                    for each other DMk in DM(Ti)
                        send TSUPDATE{<Ti,t(Ti)>,t'(Ti),fromi}
                    t(Ti) := t'(Ti)
        end case
    until count(Ti) = N(Ti)
    for each DMk in DM(Ti)
        send PROCESS{<Ti,t(Ti)>,fromi}
    end MASTER.

```

Figure 4.3 The Master TCP Algorithm at T_{Mi}

```

procedure SLAVE {at DMk}
begin
    wait_message(REQUEST {<Ti,t(Ti)>,fromi,<misc>})
    E(Ti) := determine_read-writeset
    evaluate (request, rejected)
    if rejected
        then send TMi REJECT {<Ti,t(Ti)>,fromk}; exit
        else send TMi ACCEPT {<Ti,t(Ti)>,fromk}
    accepted := false
    repeat
        wait_message (reply)
        case reply of
            TSUPDATE: update (request, t'(Ti))
                      t(Ti) := t'(Ti)
            PROCESS:  accepted := true
        end case
    until accepted
    process (request)
end SLAVE.

```

Figure 4.4 The Slave TCP Algorithm at DM_k

```

monitor ACCESS
  var table: array [1..N] of
    record
      queue
      stamp
    end

  procedure evaluate (request:transaction, var rejected:boolean)
  begin
    rejected:=false
    for each Ej in E(Ti)
      if t(Ti) < table[j].stamp
        then rejected:=true
      if not (rejected) then
        for each Ej in E(Ti)
          enter (<Ti,t(Ti)>,table[j].queue)
        end evaluate.

  procedure update (request:transaction, ts:timestamp)
  begin
    for each Ej in E(Ti)
      remove (<Ti,t(Ti)>,table[j].queue)
      t(Ti):=ts
      enter (<Ti,t(Ti)>,table[j].queue)
    end update.

  procedure process (request:transaction)
  begin
    await (t(Ti) = min {t(Tj) | Tj is in Xk(Ti)})
    for each Ej in E(Ti)
      remove (<Ti,t(Ti)>,table[j].queue)
      t(Ej):=t(Ti)
      enter (<Ti,t(Ti)>,LC_queue)
    end process.

end ACCESS.

```

Figure 4.5 The RAT Monitor: Data Structure and Procedures

starts over by awaiting the new REQUEST message. The boolean variables rejected and accepted are used to represent local and global acceptance states respectively at each DM.

4.2 Protocol Verification

In Section 1.3 we noted that a protocol must satisfy certain verification criteria such as:

1. correctness with respect to the specified requirements,
2. freedom from deadlock,
3. freedom from indefinite blocking, and
4. freedom from indefinite restart.

In this section the ADD protocol will be verified according to the above criteria. Correctness will be demonstrated by showing that the ADD protocol produces an acyclic $<$ ordering of transactions. Deadlock-freeness will be seen to be a by-product of correct behaviour. Indefinite blocking is not possible due to the assumed finiteness of the transaction read-writeset and of the number of transactions. Indefinite restart, however, is shown to be a problem for basic T/O algorithms. Various mechanisms which affect the overall restart behaviour will be presented and the trade-offs involved will be discussed from a qualitative point of view.

4.2.1 Correctness

In the context of databases, correctness of a synchronization protocol implies a serializable execution or an acyclic $<$ ordering. In

the ADD protocol, the operation of removing T_i from the Resource Access Table and passing it to the LC is performed as an indivisible operation by the monitor procedures at each DMk. Therefore, if $T_i \leq_e T_k$ for some entity e , then $T_i \leq_e T_j$ for every e in the intersection of $E(T_i)$ and $E(T_j)$ at DMk and we write $T_i \leq_k T_j$. An execution is nonserializable if for some DMk and DMn, both $T_i \leq_k T_j$ and $T_j \leq_n T_i$ are true. We show that this nonserializable execution cannot occur for the ADD protocol. The proof of correctness is based on the following assertions:

1. Once a transaction is accepted at DMk, it cannot be rejected by DMk.
2. Once a transaction has been globally accepted, the transaction timestamp used to update each entity will be the same at all entities.
3. The basic T/O mechanism guarantees an acyclic ordering if each transaction has a unique timestamp.

Assertion 4.1 If transaction T_i is locally accepted at node DMk at time t_1 (and is possibly updated at time t_4), then it cannot be rejected by DMk at time $t_2 > t_1$.

Proof. Let $t(T_i)$ be the timestamp of transaction T_i . Let $t'(T_i) > t(T_i)$ be the updated timestamp for T_i at time $t_4 > t_1$. Let $t(T_i, t_1)$ and $t(E_k, t_1)$ be the transaction and entity timestamps at time t_1 . If T_i is accepted at time t_1 then $t(E_k, t_1) < t(T_i, t_1)$ for every E_k in $E(T_i)$. If T_i gets rejected at time $t_2 > t_1$ then there exists T_j such that (a) E_k is in the intersection of $E(T_i)$ and $E(T_j)$, (b) $t(T_j) > t(T_i)$ and (c) T_j gets passed to LC at time $t_3 < t_2$. Consider three cases:

1. $t_3 < t_1$; At time t_3 , $t(E_k, t_3) = t(T_j, t_3) = t(T_j)$. Since $t(E_k)$ is a monotonically increasing function of time we have $t(E_k, t_1) > t(E_k, t_3) = t(T_j) > t(T_i)$. Therefore, T_i is rejected at time t_1 (which is a contradiction).
2. $t_1 < t_3 < t_4$; At time $t_3 < t_4$, $t(T_i, t_3) = t(T_i) < t(T_j)$. If T_j

is passed to LC at time t_3 , then $t(T_j, t_3)$ must be $= \min\{t(T_n) \mid T_n \text{ is in } X_k(T_j)\}$. Since T_i is in $X_k(T_j)$, T_k cannot be passed to LC at time t_3 before T_i is processed.

3. $t_4 < t_3$; At time $t_3 > t_4$, $t(T_i, t_3) = t'(T_i) > t(T_i, t_1) = t(T_i)$.

a) If $t'(T_i) < t(T_j)$ then proceed as in 2 above.

b) If $t(T_i, t_3) = t'(T_i) > t(T_j)$ then T_i would not be rejected at time t_2 due to T_j since $t'(T_i) > t(T_j) = t(E_k, t_3)$.

Therefore, we conclude that a transaction T_i cannot be rejected by a DM after having been locally accepted, even if the timestamp of T_i is subsequently updated. Q.E.D.

Assertion 4.2. At the time that each DM receives a global acceptance (PROCESS) message from the TM for transaction T_i , each DM will update the entity timestamp based on the same timestamp value $t(T_i)$.

Proof. When the PROCESS message is sent by the TM, all DMs have locally accepted T_i ; that is, $\text{count}(T_i) = N(T_i)$. Therefore, according to Assertion 4.1, T_i cannot be rejected by any DM and the timestamp contained in the PROCESS message is the final timestamp for T_i . Due to variable delays experienced by each message sent in the network, it is possible for a DM to receive a PROCESS message before receiving a previous TSUPDATE message. Each message submitted, however, is identified by a transaction identifier pair $\langle T_i, t(T_i) \rangle$. The DM can either wait for the TSUPDATE to arrive before executing the 'process' procedure or it can use the timestamp supplied by the PROCESS message as the final $t(T_i)$ and disregard the subsequent TSUPDATE message. In any case, the PROCESS message ensures the uniqueness of $t(T_i)$ upon global acceptance. Q.E.D.

Theorem 4.1 The ADD protocol cannot cause a nonserializable execution such that, for some T_i and T_j and for some DM_k and DM_n , both $T_i <_k T_j$ and $T_j <_n T_i$ are true.

Proof. Assertion 4.2 claims that all transaction timestamps for T_i will have a common value at all DM in $DM(T_i)$ when T_i gets passed to the LC. Let t_1 be the time at which T_j gets processed at DM_n . If $T_j <_n T_i$, that is, T_j is processed before T_i at DM_n , then consider two cases:

1. If $t(T_i) < t(T_j)$, T_i was not in RAT_n at time t_1 since to get processed by the GC, $t(T_j) = \min \{t(T_k) \mid T_k \text{ is in } X_n(T_j)\}$. After T_j gets processed and for each E in the intersection of $E(T_i)$ and $E(T_j)$, we have $t(E) = t(T_j) > t(T_i)$. Therefore, T_i will get rejected at DM_n and must receive a new timestamp $t'(T_i) > t(T_j)$. Therefore, we cannot have $T_i <_k T_j$ for any DM_k .
2. If $t(T_j) < t(T_i)$, T_i can get accepted before T_j at DM_k , however $T_j <_n T_i$ implies that T_j is not rejected at any DM_k (since DM_n received the PROCESS message). Therefore, T_i has not been processed before T_j at any DM_k and $T_i <_k T_j$ cannot be true.

By similar arguments we can show the converse argument that $T_i <_k T_j$ implies $T_j <_n T_i$ is not true. Q.E.D.

We have shown that the ADD protocol correctly serializes transactions passed to the LC in a unique timestamp order. A by-product of timestamp ordering is that deadlocks cannot occur; a transaction can only be blocked from accessing resources by transactions whose timestamps are smaller so we cannot possibly have a cycle of transactions waiting for each other. We note at this point that the ADD protocol specified in [MAHM79] used a different but equivalent method of ensuring that all DMs accept T_i before being processed by any DM. Rather than having each DM notify the TM upon local acceptance, each DM sends the ACCEPT message to all other DMs in $DM(T_i)$. At each DM, T_i is globally accepted once the DM has received ACCEPTS from all other DMs in

DM(T_i). The advantages of both approaches are discussed in [LAFE81].

4.2.2 Transaction Termination

Although we have shown the ADD protocol to correctly serialize transactions, we have not guaranteed that transactions will in fact terminate. Concurrency controls using WAIT and RESTART can be subject to nontermination for the following two reasons [ROSE78]:

1. waiting forever (indefinite blocking) due to:
 - a. deadlock,
 - b. an infinite number of new transactions which cause the given transaction to wait, and
 - c. a transaction that runs forever; and
2. restarting forever (indefinite restarts) due to an infinite sequence of restarting the same transaction.

In the previous section, it was shown that deadlock cannot occur.

Although we assume that no transaction runs forever, the problems of indefinite delay and indefinite restarts must be addressed. The FSMs of the TM and DM process in Figure 4.6 identify five loops (marked [1], [2], ...) which can cause such problems. Actually, loop [4] cannot repeat forever, since the database, and thus the cardinality of the read-writeset of T_i , is assumed to be finite.

Loop [5] can only repeat forever if some globally accepted transaction T_i never becomes the oldest transaction in $X_k(T_i)$ at some DMk. This condition can only occur if an infinite sequence of T_j continues to arrive with $t(T_j) < t(T_i)$. The ADD protocol does not disallow transactions to "lie about their age" by being assigned

Figure 4.6 FSM for the TM and DM Transaction Control Processes

timestamps in nonincreasing order. The ability to assign an smaller timestamp may be desirable since it may reduce the amount of time a transaction has to wait before being processed by a GC. At the same time, however, a transaction with an older timestamp will also be more likely to be rejected so a TM must trade off the decreased delay with the increased probability of rejection due to assigning older timestamps. Thus it seems possible to get an infinite sequence of transactions with smaller timestamps generated by the TMs. The fact that transactions must be processed in timestamp order, however, will ensure that T_i eventually gets processed. Each rejected transaction must receive a larger timestamp and there are only a finite number of unique transactions with timestamps $t(T_j)$ such that $t(E) < t(T_j) < t(T_i)$ for each E in $E(T_i)$.

Updating forever (loops [1] and [2]) occurs as a result of restarting forever (loop [3]). Unfortunately, the ADD protocol, as stated, has no mechanism to handle this problem. Although Assertion 1 implies that the number of DMs which have not accepted T_i locally is a monotonically decreasing function, we have no guarantee that the number will not remain the same from one update to the next.

4.2.3 Reducing Restarts

Various factors influence the rate of rejection of transactions such as the relative synchrony of the clock generation mechanisms at the TMs, the disordering of messages and transactions in the network and the relative rates of transaction arrival at the TMs. These factors will be discussed in further detail when we develop some performance analytic

results for the basic T/O scheme in Chapter 5. Consider now two mechanisms that can reduce the rate of rejection: rejection timestamps and reservations.

A transaction T_i gets rejected at DM_k because $t(E) > t(T_i)$ for some E in $E(T_i)$. Although the TM will assign a new timestamp $t'(T_i) > t(T_i)$ we may have $t(E) > t'(T_i)$ and so T_i is rejected again. It seems advisable for the DM to include a rejection timestamp field in the REJECT message as

REJECT $\{ \langle T_i, t(T_i) \rangle, from_k, TS \}$

where TS is the timestamp of largest $t(E)$ for E in $E(T_i)$. Thus the TM will ensure that $t'(T_i) > t(E)$ and T_i will not get rejected at the same DM due to the same entity timestamps. It is still possible, however, for T_i to get rejected at the same DM due to the arrival and processing of a new T_j . For example, assume T_i with $t(T_i)$ gets rejected at a DM_k due to $t(E) > t(T_i)$ for E in $E(T_i)$. The DM sends a

REJECT $\{ \langle T_i, t(T_i) \rangle, from_k, TS=t(E) \}$

message and TM responds with a TSUPDATE or new REQUEST message with $t'(T_i) > t(E)$. Meanwhile, assume T_j arrives at DM_k such that E is in $E(T_j)$ and $t(T_j) > t'(T_i) > t(E)$. T_j may get globally accepted and processed before the DM receives the TSUPDATE for T_i . Now $t(E)=t(T_j) > t'(T_i)$ and T_i is rejected again. Clearly this can occur any number of times even though a rejection timestamp field was included in the REJECT message.

Reservation is a mechanism whereby a transaction holds its position in the RAT. The algorithm, as previously described, already

includes a limited reservation mechanism. When a transaction was rejected, a new REQUEST message was sent only to the rejecting DM rather than to all DMs in $DM(T_i)$. In effect, at the accepting DMs, T_i held a reservation in the RAT so that T_j with $t(T_j) > t(T_i)$ was blocked even though it may have been globally accepted. The blocked T_j could not get processed thus preventing the rejection of T_i . Without the reservation mechanism, the accepted transactions could have proceeded with less delay but at the expense of an increased overall rate of rejection.

It can be guaranteed that each T_i will only get rejected once at each DM in $DM(T_i)$ by allowing T_i to hold a reservation even at the rejecting DM. To illustrate, assume that $\langle T_i, t(T_i) \rangle$ gets rejected at DM_k due to $t(E) > t(T_i)$ for some E in $E(T_i)$. DM_k sends a

REJECT $\{\langle T_i, t(T_i) \rangle, from_k, TS=t(E)\}$

message to the TM but places T_i in the RAT. When T_j arrives at DM_k with $t(T_j) > t(E) > t(T_i)$, T_i is considered to be in $X_k(T_j)$. T_j will not get processed until TSUPDATE arrives with $t'(T_i) > t(E)$. If $t'(T_i) > t(T_j)$, then $T_j <_k T_i$, otherwise $T_i <_k T_j$.

By using the reservation and rejection timestamp mechanism the number of rejections for any transaction can be limited to $N(T_i)$, the number of DMs in $DM(T_i)$. Each transaction can be further limited to only one rejection by ensuring that the TM does not send the TSUPDATE message until it has received an ACCEPT or REJECT from every DM in $DM(T_i)$. Then at the expense of this extra delay, the TM can ensure that $t'(T_i) > \max \{TS\}$ over all TS in the REJECT messages. In chapter 5 the use of delay as a tactic to reduce restarts for basic T/O will be

discussed from a performance point of view.

4.3 Dynamic Request Generation

Up to this point, a static policy of resource requests has been assumed. Predeclaration could be used to reduce the rate of rejection since the read-writeset of each transaction was fixed. There may, however, be situations in which a dynamic request policy is more appropriate. In this section the basic T/O concurrency control mechanisms are extended to deal with the dynamic request case. The ADD* protocol which deals with reads and writes, rather than transactions, as atomic operations, is presented. The ADD* protocol will be specified and verified as in the previous sections. Since reservations cannot be used in ADD*, a procedure for handling restarts is presented. Some guidelines for improving restart behaviour in ADD* will also be considered.

4.3.1 The ADD* Protocol: Dynamic Requests

The ADD protocol separated the synchronization phase of transactions from the execution phase. This was possible due to the assumed predeclaration of the transaction read-writeset. A transaction's actual or estimated read-writeset is determined in order to package the subtransaction as an atomic unit to be executed at each DM. By correctly serializing the order of presentation of transactions to the LC, synchronization was performed with no rejections occurring at the execution phase.

22
OF / DE

The functioning of the new protocol ADD* is modified from ADD to reflect that each dmread and precommit/dmwrite is considered atomic and requires synchronization. In the dynamic case the synchronization and execution of transactions is not separated. Although it is possible for a transaction to be rejected after some processing has occurred, transaction processing model TP3 (Section 2.6) is used to provide atomic commitment and prevention of rollback. The two-phase commit must now be part of the synchronization phase. Any rejection of an operation will lead to a complete restart of the transaction; timestamp updates are not possible as some operations may already have been executed. The version of basic T/O described in ADD* incorporates separate read (R-TS) and write (W-TS) timestamps (Figure 4.7) in order to reduce restarts and allow for implementation of Thomas Write Rule optimization.

The messages exchanged between the master TCP at TMj and the slave TCP at DMk are;

REQUEST{<Ti,t(Ti)>,fromj, operation(entity)}	...sent by TM to each DM as needed,
ACCEPT{<Ti,t(Ti)>,fromk,value}	...sent by DMk for each precommit or dmread,
REJECT{<Ti,t(Ti)>,fromk}	...sent by DMk for each precommit or dmread,
COMMITTED{<Ti,t(Ti)>,fromk}	...sent by DMk after processing the dmwrite.

where

operation is either dmread, precommit or dmwrite; and

value is the result of the dmread. Value is null for a precommit operation.

Figure 4.7 Structure of ADD* Protocol

For each operation of transaction T_i, the master TCP issues a REQUEST message to the appropriate DM. The slave TCP oversees the acceptance and execution phases of each operation via the monitor procedures 'evaluate' and 'process'. If the operation is rejected, the slave TCP sends a REJECT message to the master TCP which then restarts the transaction by clearing the private workspace and aborting all operations in progress. If accepted, the operation is placed in the appropriate queue of the RAT and the master TCP is informed. For a read operation which is not aborted prior to execution, the master TCP receives an ACCEPT message with 'value' containing the result of the read. For a precommit, no results are being returned so the slave can issue an acceptance acknowledgement by sending the ACCEPT message with a null value field. Due to the two-phase commit procedure, dmwrites cannot be rejected. The master TCP is, however, informed by a COMMITTED message as each slave completes the second phase of the 2PC.

A read operation is processed once it becomes the oldest pending operation against an entity. Read results are passed to the private workspace by the slave TCP. When a DM receives and accepts a precommit the new value can be immediately copied to secure storage. The DM must guarantee that when the corresponding dmwrite arrives, no other operation will have been processed against the entity to cause the dmwrite to be rejected [BER80a]. The effect is similar to setting a write-lock against the entity except that:

1. dmreads with smaller timestamps than the precommit are allowed to proceed since the update has not yet been performed, and
2. precommits from transactions with smaller timestamps can be

processed and "steal" the initial precommit's lock on the entity. In this sense the lock acts as a reservation and so is not fixed. Dmreads, precommits and dmwrites from transactions with larger timestamps can be accepted but must be blocked from being processed until the pending dmwrite arrives and is processed. This mechanism ensures that the two phases (precommit and dmwrite) of the 2PC procedure are atomic.

Figures 4.8, 4.9, 4.10 and 4.11 present respectively the "master", "slave", "evaluate" and "process" procedures. The RAT is implemented as a physical locking mechanism using the timestamp ordered queues. A transaction operation is only eligible for processing if it is at the head of the queue and is either a dmread or a precommit whose pending dmwrite has arrived (see "process" module, Figure 4.11). We make the following explanatory notes with respect to the TCP and monitor routines:

1. PW denotes the transaction private workspace. Procedures "pwread" and "pwwrite" indicate read and write operations on PW.
2. R-TS(e)/W-TS(e) denotes the timestamp of the largest-timestamped dmread/dmwrite operation processed against entity e. Thus the RAT monitor data structure is:

```
var table: array[1..N] of
    record
        queue
        R-TS
        W-TS
    end
```

3. Procedures "enter" and "remove" manipulate the RAT queues in timestamp order. If the operation being entered is a dmwrite, it is placed in the RAT queue immediately following the corresponding precommit.
4. Procedures "first_request" and "second_request" only examine the appropriate requests on the queues without removing them.

5. Procedure "await_processing" suspends a slave TCP until the appropriate dmread or dmwrite operation is processed.

The algorithms can be extended to incorporate TWR optimization as follows:

1. In the evaluate module "case precommit" becomes

```
if t(Ti) < R-TS(entity)
then rejected:=true
else enter (request, queue(entity))
      if t(Ti) > W-TS(entity)
      then copy to secure storage
```

2. In the process module "case precommit" becomes

```
next:=second_request(queue(entity))
if next.operation=dmwrite
then remove (request, queue(entity))
      remove (request, queue(entity))
      if request.t(Ti) < W-TS(entity)
      then {discard request}
      else W-TS(entity):=request.t(Ti)
           write(entity, new value)
           {unblock TCP to signal committed}
```

In fact, in the evaluate module the precommit should be tagged for discard in order to not block other eligible operations at the process module.

For ease of presentation the master TCP is shown as submitting each operation serially, executing a "wait(response)" after each operation. In actual implementation it should be possible for the TCP to submit several dmreads (or precommits during the 2PC) and await responses on each. Clearly the responses must be received for all dmreads before executing the END operation and responses must be received from all precommits before submitting the dmwrites.

The database access procedures, "read and write", of the process module are subject to local locking, thus delaying the time at which the


```

procedure Master {at TMi}
  begin
    wait_receive (BEGIN)
    t(Ti):=assign_timestamp
    PW:=assign_private_workspace
    done:=false
    repeat
      wait_next(operation)
      case operation of
        READ: if entity E is in PW
              then pwrite(E)
              else select stored copy Ej of E
                    send DMj REQUEST{<Ti,t(Ti)>,fromi,dmread(E)}
                    wait(response)
                    if rejected then RESTART
                    else E:=value{Ej}
                          pwrite(E)
        WRITE: if entity E is in PW
              then pwrite (E,E')
              else create E in PW
                    pwrite (E,E')
        END: done:=true
      end case
    until done
    for each updated E in PW
      for each stored copy Ej of E
        send DMj REQUEST{<Ti,t(Ti)>,fromi,precommit(E,E')}
        wait(response)
        if rejected then RESTART
      for each updated E in PW
        for each stored copy Ej of E
          send DMj REQUEST{<Ti,t(Ti)>,fromi,dmwrite(E)}
    end master.

```

Figure 4.8 Master TCP at Tmi

```

procedure slave {at DMk}
  begin
    wait_receive (REQUEST{<Ti,t(Ti)>,fromi, operation, entity})
    evaluate(request,rejected)
    if rejected then send Tmi REJECT{<Ti,t(Ti)>,fromk}
    else case request.operation of
      dmread: await_processing
              send Tmi RESULT{<Ti,t(Ti)>,fromk,value}
      dmwrite: await_processing
               send Tmi COMMITTED{<Ti,t(Ti)>,fromk}
      precommit: send ACCEPTED{<Ti,t(Ti)>,fromk}
    end case
  end slave.

```

Figure 4.9 Slave TCP at DMk

```

procedure evaluate (request:transaction, var rejected:boolean)
begin
    rejected:=false
    with request do
        with table do
            case operation of
                dmread: if t(Ti) < W-TS(entity)
                        then rejected:=true
                        else enter(request,queue(entity))
                dmwrite: enter(request,queue(entity))
                precommit:if t(Ti) < R-TS(entity)
                        then rejected:=true
                        else if t(Ti) < W-TS(entity)
                        then rejected:=true
                        else enter(request,queue(entity))
                        {copy value to secure storage}
            end case
        end table
    end request
end evaluate.

```

Figure 4.10 Evaluate Monitor Procedure

```

procedure process
begin
    repeat
        with table do
            for each entity at DMk
                while not_empty(queue(entity))
                    request:=first_request(queue(entity))
                    case request.operation of
                        dmread: remove(request,queue(entity))
                                R-TS(entity):=max[request.t(Ti),R-TS(entity)]
                                read(entity)
                                {unlock TCP and pass entity value}
                        precommit: next:=second_request(queue(entity))
                                if next.operation=dmwrite
                                then remove(request,queue(entity))
                                    remove(request,queue(entity))
                                    W-TS(entity):=request.t(Ti)
                                    write(entity,new value)
                                    {unlock TCP to signal committed}
                                end if
                    end case
                end while
            end table
        forever
    end process.

```

Figure 4.11 Process Monitor Procedure

entity timestamp is updated. The updating of the entity timestamps can be further delayed by giving the evaluate module a higher priority than the process module. This policy may cause already accepted operations to wait, but on the whole, the rejection rate of transactions will be reduced.

Figures 4.12 and 4.13 represent the FSMs for the master and slave TCPs in ADD*. The FSMs will be discussed further in Section 4.3.3 with respect to termination and correctness of the ADD* protocol.

4.3.2 Handling Restarts in ADD*

The correctness of ADD* is based on each operation having a common timestamp. The basic T/O acceptance mechanism and the timestamp ordered RAT queues will ensure that $T_i < T_j$ iff $t(T_i) < t(T_j)$. Since transactions are assigned a new timestamp when a restart occurs, a proper restart procedure is needed to ensure that operations with the old timestamp are properly cleared out of the system before any new operations are issued.

According to transaction processing model TP3, a transaction consists of a sequence of dreads followed by an atomic 2PC procedure consisting of a sequence of precommits and a sequence of dwrites. Thus using TP3, a transaction can be in one of three phases: a read phase, a precommit phase or a write phase. TP3 ensures that a transaction does not proceed to the precommit phase until all reads have been accepted. The 2PC procedure ensures that a transaction does not proceed to the write phase until all precommits have been accepted. The restart

Figure 4.12 FSM for Slave TCP at DM_k

Figure 4.13 FSM for Master TCP

procedure entails clearing the private workspace, aborting any outstanding operations and starting transaction execution from the BEGIN command with a new timestamp. The procedure for aborting any outstanding operations will be described by outlining the possible cases for each of the above phases.

If the transaction is in the read phase, the master TCP sends an ABORT{<Ti,t(Ti)>,from} message to each slave TCP to which an outstanding dmread was sent. Each slave will respond with an acknowledgement ACK{<Ti,t(Ti)>,from} to indicate that the dmread was aborted. The outstanding dmread may be in one of the following phases when the slave receives the ABORT message:

1. If the dmread.REQUEST is still in transit, the slave waits for the dmread to arrive, discards it and then ACKs the master TCP.
2. If the dmread REQUEST has been received and accepted, the slave issues a monitor call to an abort procedure.
 - a. If the dmread is in the RAT, it is removed.
 - b. If the dmread is being executed, it can either be aborted or proceed to (c).
 - c. If the dmread has been executed, the result can be discarded by the slave.

In any case the slave sends the master TCP an ACK.

3. If the read result has already been returned, the slave disregards the ABORT message. The master TCP will discard the result when it is received.

Note that because the transaction identifier <Ti,t(Ti)> contains the timestamp, there can be no ambiguity as to the version of Ti intended for execution or aborting.

If the transaction is in the precommit phase, the master TCP sends an ABORT message to each slave TCP to which an outstanding precommit

has been sent. The outstanding precommit may be in one of the following phases when the slave receives the ABORT message:

1. If the precommit is in transit, the slave waits for the precommit to arrive, discards it and ACKs the master TCP.
2. If the precommit has been received and accepted, the slave will issue a monitor 'abort' call which removes the precommit from the RAT and clears the private workspace. The master TCP will disregard the received ACCEPT message.

If the transaction is in the write phase, no further rejections can occur and thus the restart procedure does not include this case. This implies that once a write timestamp is updated, it cannot be undone (in order to avoid rollback). Note, however, that read timestamps may have been updated and other writes have proceeded based on the updated R-TS value. This does not cause any inconsistencies since the dmread did not affect the entity value read by any subsequent dmwrites.

The updated read timestamp, based on the rejected transaction, may cause some writes to be rejected unnecessarily. For example, the execution $\langle R1(e), 5 \rangle$, $\langle R2(e), 10 \rangle$, $\langle W3(e), 7 \rangle$ would cause $W3(e)$ to be rejected. If $R2(e)$ has been restarted, then $W3(e)$ should be acceptable. Unfortunately, it is a difficult matter to restore the old timestamp value, which could in any case lead to inconsistencies since the correct R-TS value is usually unknown.

4.3.3 Verification of ADD*

The proof of correctness for ADD* is similar to that used for ADD. The protocol is correct if:

1. the transaction timestamps used to update entity timestamps are

common for all operations within a transaction; and

2. the basic T/O mechanisms guarantee an acyclic ordering of transactions even though operations are synchronized separately.

The restart procedure outlined ensures that the first point holds true. When a transaction is rejected its private workspace is cleared, all operations in progress are aborted and the transaction is restarted with a new timestamp. Thus when the second phase of the 2PC procedure starts, no further rejections can occur and all DMs will update the entities based on a common timestamp.

Theorem 4.2 The ADD* protocol cannot cause a nonserializable execution such that for some transactions T_i and T_j and some entities e and f , both $T_i \prec_e T_j$ and $T_j \prec_f T_i$ are true.

Proof. Assume that $t(T_i) < t(T_j)$. Let t_1 be the time of processing T_j and let t_2 be time of arrival of T_i at some DM. Consider the two cases:

1. $t_2 < t_1$. T_i will not get rejected due to T_j . If T_i is not rejected due to some other transaction, then $T_i \prec_e T_j$ since T_i is at the head of the queue (T_i is in $X_k(T_j)$ and $t(T_i) < t(T_j)$).
2. $t_1 < t_2$.
 - (a) If either T_i or T_j is a write/precommit operation then T_i will get rejected since $t(e, t_2) > t(T_j) > t(T_i)$. The TM must restart T_i and issue a new timestamp such that $t'(T_i) > t(T_j)$ which contradicts the assumption of $t(T_i) < t(T_j)$.
 - (b) If both operations are reads, T_i may get accepted. According to our definition of the $\prec_e$ relation (Section 3.1) we do not say $T_j \prec_e T_i$ if both operations are reads.

Therefore, if $t(T_i) < t(T_j)$ we have $T_i \prec_e T_j$. Since the timestamp of each transaction is unique both $T_i \prec_e T_j$ and $T_j \prec_f T_i$ cannot be true.

Q.E.D.

Since ADD* uses the same basic T/O mechanism as ADD, similar

observations can be made concerning transaction termination. Thus ADD* cannot wait forever but can restart indefinitely. Rejection timestamps can be used to reduce the rejection rate. Reservations, however, cannot be used since a dynamic request scheme requires that a rejection of a transaction at one node requires a restart of the transaction at all nodes at which it executed. In the next section the particular transaction interleavings that can cause restarts are examined in order to develop policies to reduce restarts.

4.3.4 Examination of Restarts in ADD*

In using timestamp ordering to serialize transactions we have imposed a fixed ordering that the transactions' execution must produce. This is in contrast to two-phase locking where the serialization order is imposed by the relative order in which each transaction achieves its locked point.

The following qualitative observations can be made concerning use of basic T/O in ADD*. Let R_i and P_i denote the read and precommit operations of transaction T_i .

1. Assume $t(T_i) < t(T_j)$ and R_j is executed before R_i . R_i will be accepted since out of order reads are allowed, but any precommit P_i that follows will get rejected. Thus if an out of order read knows about a subsequent write on the same entity, the rejection may as well occur on the read before any processing takes place.
2. Assume $t(T_i) < t(T_j)$ and R_i and R_j get executed in timestamp order. A precommit P_i that follows will get rejected because R_j was allowed to read. Thus, if read operation knows about a

subsequent write on the same entity, it should block reads with larger timestamps until after the write has been processed. Thus a 'read with intent to update' operation, similar to setting a write lock, may be useful for reducing restarts.

The above mechanisms assume that TCP has some knowledge of the transaction read-writeset. Although nonpredeclaration is assumed, some limited knowledge may become available during the course of a transaction execution. This information can be used to reduce the cost of restarts (as in 1 above) or reduce the occurrence of restarts (as in 2 above).

4.4 Hybrid T/O Schemes

We now consider some hybrid T/O schemes which may have more desirable characteristics than either basic T/O or conservative T/O in some applications. The concurrency control problem can be decomposed into the synchronization of conflicting reads and writes and the synchronization of conflicting writes in such a way that different concurrency control schemes can be used for each subproblem [BER80a]. By further decomposing the synchronization of reads and writes we can identify three conflict pairs: read/write (R/W), write/read (W/R) and write/write (W/W). Thus each read operation must be synchronized against all conflicting writes and each write must be synchronized against all conflicting reads and writes.

When using basic T/O as a synchronization technique an R-TS entity timestamp is needed to regulate writes against reads; a W-TS entity

timestamp is needed to regulate reads against writes and writes against other writes. The use of entity timestamps has been criticized as impractical because of the storage requirements. Analysis of the granularity of entity timestamps can determine the optimum number of timestamps required. For low conflict (optimistic) environments the granularity can be coarser whereas for high conflict (pessimistic) environments a finer granularity is desired to increase concurrency and reduce rejections. Since pure conservative T/O did not require entity timestamps, it may be possible to reduce timestamp requirements by considering hybrid schemes.

Figure * 4.14 presents eight hybrid schemes based on the synchronization mechanism used for each conflict pair and describes the entity timestamp requirements of each scheme. For example, Scheme 6 indicates that each write must wait for all reads with smaller timestamps before being processed (conservative T/O) whereas each read and write need not wait for older writes but must be checked against the W-TS entity timestamp before being accepted (basic T/O). Scheme 1 is pure conservative T/O and Scheme 8 is pure basic T/O as used in ADD*. Note that for each scheme which requires a W-TS timestamp (all except 1, 3 and 5) we can substitute a TWR policy for W/W synchronization to reduce the number of restarts.

For schemes using basic T/O we prove the following assertions with respect to the need for entity timestamps:

Assertion 4.4 If conservative T/O is used to synchronize reads against write conflicts (R/W) then write operations cannot be rejected due to a

Scheme	Synchronization Techniques Used For Each Conflict Pair			Timestamp Requirements	
	R/W	W/R	W/W	R-TS	W-TS
1	conservative	conservative	conservative	no	no
2	conservative	conservative	basic	no	yes
3	conservative	basic	conservative	no [1]	no
4	conservative	basic	basic	no [1]	yes
5	basic	conservative	conservative	no	no [2]
6	basic	conservative	basic	no	yes
7	basic	basic	conservative	yes	yes
8	basic	basic	basic	yes	yes

NOTE: [1] see Assertion 4.4 in the text
[2] see Assertion 4.5 in the text

Figure 4.14 Hybrid T/O Schemes

read conflict (W/R) even if basic T/O is used to synchronize the W/R conflict.

Proof. If some W_i arrives and gets rejected then $R-TS(e) > t(W_i)$ for some e in $E(W_i)$. Let R_j be the read operation with timestamp $t(R_j) = R-TS(e)$. Then R_j did not obey conservative T/O with respect to R/W conflicts since there exists W_i with $t(W_i) < t(R_j)$ for which R_j did not wait.

Assertion 4.5 If conservative T/O is used to synchronize writes against read conflicts (W/R) then read operations cannot be rejected due to a write conflict (R/W) even if basic T/O is used to synchronize R/W conflict.

Proof. If some R_i arrives and gets rejected then $W-TS(e) > t(R_i)$ for some e in $E(R_i)$. Let W_j be the write operation with timestamp $t(W_j) = W-TS(e)$. Then W_j did not obey conservative T/O with respect to W/R conflicts since there exists R_i with $t(R_i) < t(W_j)$ for which W_j did not wait.

Assertion 4 implies that no R-TS is needed for Schemes 3 and 4 (note [1] of Figure 4.14). Assertion 5 implies that Scheme 5 does not require W-TS (note [2] of Figure 4.14). Scheme 6, however, requires W-TS since basic T/O is being used for W/W synchronization.

Scheme 4, with TWR optimization for W/W conflicts, is used in the SDD-1 concurrency control. In addition, transaction classes and conflict graph analysis are used to reduce the number of TM/DMs which must communicate regularly. Conservative T/O requires regular communication between each TM and DM and is thus slow and message-inefficient for

optimistic (low conflict) environments. The SDD-1 mechanism allows writes to proceed virtually unsynchronized (no write rejections can occur) by placing the burden for synchronization on the reads which must wait for all writes (or NULLWRITES) with smaller timestamps. To operate efficiently, the SDD-1 approach (Scheme 4) requires high write traffic and is thus most suitable for update-oriented systems.

Schemes 6 (and 5) on the other hand, place the burden of synchronization on the writes and is thus most suitable for a retrieval-oriented system. Each read operation can proceed unsynchronized whereas each write must wait for all older reads (and all older writes for Scheme 5). Although a lot of reads are expected in a retrieval-oriented system, NULLREAD messages are required from TMs having low transaction rates in order for Scheme 6 to operate correctly. Further, each TM can only send one dmread to each DM thus requiring predeclaration of readsets. Scheme 5 is similar to the ADD* protocol (Scheme 8) but is advantageous in that retrievals are faster and reads never get rejected (although writes can get rejected). Performance evaluation of the various hybrid schemes presented would provide a good area for future research.

CHAPTER 5

PERFORMANCE ANALYSIS OF BASIC T/O

Basic T/O as a concurrency control was shown to lead to the problem of multiple rejections in Chapter 4. Delay, reservations and rejection timestamps were suggested as mechanisms for reducing the probability of rejections. In this chapter, performance of the basic T/O scheme is evaluated using queueing analysis and simulation modelling.

The effect of network disordering on the LeLann implementation of conservative T/O has been analyzed using an $M/M/\infty$ model [KAM081]. For basic T/O a similar model is introduced. This model assumes: (a) a synchronized TM timestamp generation mechanism, (b) a single DM node at which all transactions conflict, and (c) instantaneous execution of each transaction at the DM. The primary result is the derivation of $P[\text{acc}]$, the probability of a transaction being accepted on any attempt. $P[\text{acc}]$ is used to calculate $E[T]$, the expected time before a transaction gets accepted. Both $P[\text{acc}]$ and $E[T]$ are seen to be functions of the network load parameter, ρ . The analysis is extended to consider transactions which access more than one DM. Simulation is used to model the effects of DM delay caused by processing and synchronization. The results are compared to those derived in [KAM081] for conservative T/O.

5.1 Factors Affecting Disorder

Different sources of delay can cause transactions to arrive at a DM out of timestamp order. The factors affecting transaction disordering in a distributed system are a function of the phases of a transaction

lifetime:

1. the transaction issues a BEGIN and is assigned a timestamp;
2. subtransactions (operations) are submitted to the DMS via the network;
3. each subtransaction experiences delay due to synchronization and contention for resources;
4. if a dynamic request policy is used, steps 2) and 3) may be repeated, and
5. the transaction is committed using 2PC.

During the above procedure the transaction is subjected to the basic T/O acceptance criteria, based on the timestamp generated in step 1. The events suggest the following quantifiable factors which can be used in an analytic or simulation model:

1. the rate at which transactions are generated;
2. the relative synchrony of distributed timestamp generation clocks and counters;
3. the variable delay experienced by messages that traverse the network;
4. the number of nodes that a transaction accesses (since for ADD* a rejection at any node causes a restart of the whole transaction); and
5. the delay experienced by each transaction waiting for global acceptance and for locked resources.

In the performance model developed in this chapter, the effects of factors 1, 3, 4 and 5 will be evaluated. Factor 2 will be discussed briefly in Chapter 6. The model assumes predeclaration (all subtransactions are submitted in parallel) and therefore applies directly to the ADD protocol. The results obtained can, however, provide insight into the operation of ADD* since both protocols share the same basic T/O acceptance mechanism.

The complexity of the concurrency control environment often leads to simulation as a tool for performance evaluation ([GARC78], [RIES79], [KANE79] and [CHEN81]). Simulation of the complete system is time consuming and requires many runs to understand the effect and interrelationship among all the variables. On the other hand, an analytic model that captures the dynamic behaviour of all the variables would be intractable.

Kamoun et al. [KAM081] use a queueing analytic approach to derive preliminary results, noting that their purpose "is not to model the complete system" - that remains the ultimate goal of the research. Similarly, the goal here is to develop an understanding of the underlying mechanisms at work in a concurrency control environment, especially for basic T/O, using both analytic modelling and simulation. The preliminary results derived could be used towards developing a comprehensive model for basic T/O concurrency control.

5.2 An M/M/ ∞ Model for Conservative T/O

Kamoun studies the effects of message disorder produced by the network using an M/M/ ∞ queueing model (Figure 5.1). The analysis is based on the LeLann conservative T/O algorithm as well as the following assumptions:

1. consecutive, monotonically increasing transaction timestamps are generated by circulating a token amongst the TMs on a virtual ring;
2. the database is fully replicated;
3. predeclaration of read-writesets implies that all REQUESTs are submitted at once;

Figure 5.1 Queueing Model of [KAM081]

4. every transaction is submitted to every DM; and

5. every transaction is assumed to conflict with every other transaction so that at each DM all transactions are processed in timestamp order.

By the token and ring mechanism, transactions can logically be thought of as entering the network from a single TM which issues consecutive timestamps. The communication network is modelled as a single queueing system with an infinite number of servers. Full replication of data entities allows the behaviour of transaction disordering to be studied by examining the order of arrivals at any DM.

Each transaction is referred to as a customer. Denote the customer with timestamp n as C_n . Let D_n denote the departure time of customer C_n from the network (that is, the arrival time of C_n to the DM). Due to the stochastic nature of the network, a customer may depart the network before other customers with smaller timestamps.

Definition. C_n is a star or in-sequence customer iff for every $k < n$, $D_k < D_n$.

Definition. C_n is a nonstar or out-of-sequence customer iff there exists $k < n$ such that $D_k > D_n$.

In other words, C_n is a star customer if it departs the network after all lower numbered customers. As an example, consider the initial input sequence:

1 2 3 4 5 6 7 8 9 10 11 12

A possible output sequence (which reflects from left to right the departure times) may be:

4 2 3 1 6 5 7 10 9 12 8 11
 * * * * *

According to the definition, customers 1, 5, 7, 8 and 11 are star customers whereas the others are nonstar customers. Star customers are only delayed due to SR/EW locking constraints whereas nonstar customers are subject to eligibility waiting time for synchronization reasons (see Figure 5.1). A waiting nonstar customer C_n becomes eligible for service only upon the arrival of all star customers C_k with $k < n$. For example,

<u>Upon the arrival of:</u>	<u>Customers that become eligible are:</u>
1	2, 3, 4
5	6
7	-
8	9, 10
11	12

When C_8 arrives, C_9 and C_{10} become eligible whereas C_{12} does not become eligible until C_{11} arrives.

5.2.1 Results of Analysis

The objective in reviewing the work in [KAM081] is to use certain derived results to compare the performance of conservative T/O and basic T/O. For conservative T/O, $E[T_1]$, the expected time until a transaction becomes eligible for DM service equals the expected network service time plus the expected eligibility waiting time as follows:

$$E[T_1] = E[X] + (1-P^*)E[W] \quad (5.1)$$

where $E[X]$ is the expected network service time for a customer,

P^* is the probability that an arbitrary customer is a star customer, and

$E[W]$ is the expected eligibility waiting time for a nonstar customer.

The interarrival time, Y , and the network service time, X , of a customer are assumed to be independent and exponentially distributed random variables with probability density functions (pdf)

$$f(y) = \lambda e^{-\lambda y}, \quad y \geq 0 \quad (5.2)$$

$$f(x) = \mu e^{-\mu x}, \quad x \geq 0 \quad (5.3)$$

Define a network traffic parameter $\rho = \lambda/\mu$. Clearly, we have $E[X] = 1/\mu$.

We state without proof [KAM081]:

Theorem. The steady state probability that an arbitrary customer is a star customer is given by

$$p^* = \frac{1 - e^{-\rho}}{\rho} \quad (5.4)$$

Theorem. The pdf of the waiting time for a nonstar customer is

$$f(w) = \frac{\mu e^{\mu w} - (\rho + e^{\mu w}) (\mu e^{-\rho e^{-\mu w}})}{\rho (1 - p^*)} \quad (5.5)$$

Due to the complexity of the above expression, $E[W]$ and $E[T]$ are not calculated algebraically. Instead, $E[T]$ is computed numerically and by simulation of the above system. The results will be presented later as appropriate to our comparative discussion with basic T/O.

5.3 An M/M/ ∞ Model for Basic T/O

An M/M/ ∞ model, similar to that presented in [KAM081], is used to analyze basic T/O. In order to make the analysis of $P[\text{acc}]$ and $E[T]$ tractable, certain simplifying assumptions are made:

1. The TM clock or timestamp generation mechanism is assumed to be

synchronized such that transactions enter the network in increasing (but not necessarily consecutive) timestamp order. Thus in considering the transactions which can cause an arbitrary transaction, TC, to be rejected, we need only consider those transactions which enter the network after TC. This is an optimistic assumption in that unsynchronized clocks will tend to cause more rejections than the model predicts.

2. Initially, the model assumes that only one DM node exists and that all transactions conflict at the single DM. This will be extended to consider the effects of accessing q out of n DM nodes such that a transaction is globally accepted only if it is accepted at every DM (as in ADD*).
3. Once accepted at the DM, each transaction is initially assumed to be executed instantaneously with no processing or synchronization delay. Therefore, the probability of acceptance is strictly a function of network disordering. In reality, the probability of acceptance is also a function of DM delay since out-of-order transactions can be accepted as long as the entity timestamp has not yet been updated. Simulation is used to evaluate the effect of DM delay by considering a two-stage queueing system composed of a network server and a node server.

Under these assumptions, the effects of network disordering on transactions that enter the network in sequence (factors 1 and 3 of Section 5.1) are examined.

Let C_j be the j^{th} customer to enter the network with timestamp $t(C_j)$ such that for $i < j$, $t(C_i) < t(C_j)$. Let D_j be the departure time of C_j from the network.

Definition. C_j is a star customer iff for every $k > j$, $D_k > D_j$.

Definition. C_j is a nonstar customer iff there exists $k > j$ such that $D_k < D_j$.

In other words, for basic T/O, C_j is a star customer if C_j leaves the network before all customers with larger timestamps, whereas in conservative T/O, C_j is a star customer if C_j leaves the network after all customers with smaller timestamps. Consider the example in Section

5.1 with customers 1 to 12 entering the network and the same departure sequence:

4 2 3 1 6 5 7 10 9 12 8 11
 * * * *

Under the basic T/O definition, customers 4, 6, 7, 10 and 12 are star customers. When referring to a star or a nonstar customer; the basic T/O definition is assumed unless specifically stated otherwise.

Definition. Define the following events:

- accepted (acc) - a customer is accepted by a DM,
- rejected (rej) - a customer is rejected by a DM,
- star - a customer is a star customer, and
- nonstar - a customer is a nonstar customer.

Notation. Given an event E, $P[E]$ is defined as the probability of event E occurring.

In general, the unconditional probabilities of acceptance and rejection can be defined in terms of conditional probabilities as

$$P[acc] = P[acc|star]P[star] + P[acc|nonstar]P[nonstar]$$

$$P[rej] = P[rej|star]P[star] + P[rej|nonstar]P[nonstar].$$

Under our initial assumption, each transaction that is accepted at a DM is processed and the entity timestamp updated instantaneously. Thus

$$P[acc|star] = P[rej|nonstar] = 1$$

$$P[acc|nonstar] = P[rej|star] = 0$$

which gives

$$P[acc] = P[star] \text{ and } P[rej] = P[nonstar].$$

The interarrival times, Y , and the network service times, X , for each customer are independent and exponentially distributed random

variables with parameters λ and μ . The respective pdfs are:

$$f(y) = \lambda e^{-\lambda y} \quad , y \geq 0 \quad (5.6)$$

$$f(x) = \mu e^{-\mu x} \quad , x \geq 0 \quad (5.7)$$

We define a network parameter $\rho = \lambda/\mu$ which is used as an indication of the load on the network.

5.3.1 Towards the Derivation of P[star]

The operation of basic T/O can be considered as a queueing system with feedback. Transactions are submitted by the TM to the DM in the forward path. Those transactions rejected by the DM are returned, along the feedback path, to the TM which resubmits them with a new timestamp value. Due to rejections, the actual rate at which transactions enter the network is greater than λ , the rate due to new arrivals.

Under our present assumptions, the probability that a transaction is accepted is simply P[star]. In solving for P[star], we initially derive P[A], the probability of acceptance, disregarding the feedback path. In the next section an iterative approach is used which accounts for the increased traffic due to rejected transactions.

We proceed by considering a discrete-state model which represents an arbitrary tagged customer, TC, entering the network (Figure 5.2). Let C^* be the set of customers with timestamps larger than TC. Due to the timestamp generation assumption, C^* is the set of all customers entering the network after TC. Define two absorbing states R and A, as follows:

States

- R : rejection absorbing state.
- A : acceptance absorbing state
- n : transient states representing the number of customers from C^* in the network before TC or any customer in C^* leaves
- $n\mu, \lambda, \mu$: transition rates

Figure 5.2 State-Transition-Rate Diagram for Tagged Customer

state R - the event that a customer from C^* leaves the network before TC, and

state A - the event that TC leaves the network before any customer in C^* .

Let state n (for $n = 0, 1, 2, 3, \dots$) represent the event that n customers from C^* have entered the network before event A or event R has occurred. Once the tagged customer enters the network, it is represented as being in one of the above states. When the system (that is, the tagged customer) is in state n , exactly one of the following must occur:

- a) a new customer enters the network (system moves to state $n+1$),
- b) TC leaves the network (system moves to state A), or
- c) a customer from C^* leaves the network (system moves to state R).

Since transitions can occur at any time and both the interarrival times and network service times are exponential, the model defines a continuous-time, discrete-state Markov chain. Due to the assumptions of an infinite server and the memoryless property of Markov processes, the rate of occurrence of each of the above events is λ , μ and $n\mu$ respectively. Figure 5.2 represents the state-transition-rate diagram [KLEI75] of the tagged customer. The transition probabilities for each state n are defined as:

$P[n, n+1] \equiv P[\text{the system is in state } n \text{ and a new customer arrives}],$

$P[n, A] \equiv P[\text{the system is in state } n \text{ and event A occurs}], \text{ and}$

$P[n, R] \equiv P[\text{the system is in state } n \text{ and event R occurs}].$

Let $P[n] \equiv P[\text{the system "passes through" state } n]$. Based on the assumption of synchronized timestamps, the system is initially in state 0. Thus, $P[0] = 1$.

If we denote $P[A, t]$ and $P[R, t]$ to be the probability of the system being in state A and state R, respectively, at time t then

$$P[A] + P[R] = \lim_{t \rightarrow \infty} \{P[A, t] + P[R, t]\} = 1.$$

If not, then for all customers in $C^* \cup \{TC\}$, no network service takes place in $(0; \infty)$. This can only occur for $\mu = 0$. The following lemma shows that the transition probabilities in Figure 5.2 are proportional to the transition rates. $P[A]$ is then derived in Theorem 5.1.

The expression for $P[A]$ (5.9) can be rewritten such that it does not involve both summation and product series. The gamma function is a generalized factorial function such that $\Gamma(n+1) = n\Gamma(n)$. For n an integer, $\Gamma(n) = (n-1)!$. For noninteger n , $\Gamma(n)$ can be computed as the definite integral

$$\Gamma(n) = \int_0^{\infty} e^{-x} x^{n-1} dx$$

The beta function can be defined in terms of the gamma function as

$$\beta(m, n) = \frac{\Gamma(m)\Gamma(n)}{\Gamma(m+n)}$$

or can be defined as the definite integral

$$\beta(m, n) = \int_0^1 x^{m-1} (1-x)^{n-1} dx$$

In Theorem 5.2, an expression for $P[A]$ equivalent to Equation 5.9 is derived.

Lemma. Given a Markovian state transition diagram, for each state i with transition rates μ_j to state j (where $j=1,2,\dots,n$), the respective transition probabilities are

$$P[i,j] = \frac{\mu_j}{\alpha} \quad \text{where} \quad \alpha = \sum_{k=1}^n \mu_k.$$

Proof. Consider $j=1$. Assume that at time 0 the system is in state i . Let E_1 denote an event causing a transition to state 1. Let E_2 denote the set of events causing a transition to any state $j=2,3,\dots,n$. Let

$$f_j(t) = \mu_j e^{-\mu_j t}, \quad t \geq 0, \quad j=1,2,3,\dots,n$$

be the pdf for the time to the next event causing a transition to state j . For a continuous random variable X with pdf $f(x)$, denote the probability that X occurs at time x as $P[x < X \leq x+dx] = f(x)dx$. Define the random variable T_1 as the time to the next event in E_1 . Define the random variable T_2 as the time to the next event in E_2 .

The probability of a transition from state i to state 1, $P[i,1]$, can be evaluated by considering the conditional probability of the transition occurring at time t and then integrating over all values of t . A transition from i to 1 at time t implies that no event in E_2 occurred in the interval $(0,t)$. Therefore,

$$\begin{aligned} P[i,1] &= \int_0^\infty P[\text{transition from } i \text{ to } 1 \text{ occurs at time } t] \\ &= \int_0^\infty P[\text{no event } E_2 \text{ in } (0,t)] P[E_1 \text{ occurs at time } t] \\ &= \int_0^\infty P[T_2 > t] f_1(t) dt \\ &= \int_0^\infty e^{-\mu_2 t} e^{-\mu_3 t} \dots e^{-\mu_n t} \mu_1 e^{-\mu_1 t} dt \\ &= \mu_1 \int_0^\infty e^{-\alpha t} dt = \frac{\mu_1}{\alpha} \int_0^\infty \alpha e^{-\alpha t} dt = \frac{\mu_1}{\alpha}. \end{aligned}$$

By symmetry, $P[i,j] = \mu_j/\alpha$ for $j=2,3,\dots,n$.

Theorem 5.1

$$P[A] = \sum_{n=0}^{\infty} \frac{\rho^n}{\prod_{i=1}^{n+1} (\rho+i)}$$

Proof

For each state n

$n = 0, 1, 2, 3, \dots$

$$1 = P[n, n+1] + P[n, A] + P[n, R]$$

$$= \lambda\alpha + \mu\alpha + n\mu\alpha$$

$$= \alpha(\lambda + (n+1)\mu) \quad \dots \text{from the lemma.}$$

Therefore $\alpha = 1/(\lambda + (n+1)\mu)$ and

$$P[n, n+1] = \lambda\alpha = \lambda/(\lambda + (n+1)\mu) = \rho/(\rho + n+1)$$

$$P[n, A] = \mu\alpha = \mu/(\lambda + (n+1)\mu) = 1/(\rho + n+1)$$

$$P[n, R] = n\mu\alpha = n\mu/(\lambda + (n+1)\mu) = n/(\rho + n+1).$$

Thus $P[n] = P[n-1, n] \cdot \dots \cdot P[n-1]$ for $n = 1, 2, 3, \dots$

$$= P[0] \cdot P[0, 1] \cdot P[1, 2] \cdot \dots \cdot P[n-1, n]$$

$$= 1 \cdot \frac{\rho}{\rho+1} \cdot \frac{\rho}{\rho+2} \cdot \frac{\rho}{\rho+3} \cdot \dots \cdot \frac{\rho}{\rho+n}$$

$$= \prod_{i=0}^n \frac{\rho}{\rho+i} = \frac{\rho^{n+1}}{\prod_{i=0}^n (\rho+i)} = \frac{\rho^n}{\prod_{i=1}^{n+1} (\rho+i)} \quad (5.8)$$

and $P[A] = \sum_{n=0}^{\infty} P[n, A] \cdot P[n]$

$$= \sum_{n=0}^{\infty} \frac{1}{\rho+n+1} \cdot \prod_{i=0}^n \frac{\rho}{\rho+i} = \sum_{n=0}^{\infty} \frac{\rho^{n+1}}{\prod_{i=0}^{n+1} (\rho+i)}$$

$$= \sum_{n=0}^{\infty} \frac{\rho^n}{\prod_{i=1}^{n+1} (\rho+i)} \quad (5.9)$$

$$\text{Likewise } P[R] = \sum_{n=0}^{\infty} P[n, R] \cdot P[n] = \sum_{n=0}^{\infty} \frac{n\rho^n}{\prod_{i=1}^{n+1} (\rho+i)} \quad (5.10)$$

Q.E.D.

Theorem 5.2

$$P[A] = e^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho)^i}{i!(\rho+i+1)}.$$

Proof

From Theorem 5.1

$$P[A] = \sum_{i=0}^{\infty} \frac{\rho^i}{i+1 \prod_{j=1}^{i+1} (\rho+j)}$$

Now

$$\begin{aligned} \prod_{j=1}^{i+1} (\rho+j) &= (\rho+i+1) \cdot (\rho+i) \cdot \dots \cdot (\rho+2) \cdot (\rho+1) \\ &= \frac{\Gamma(\rho+i+2)}{\Gamma(\rho+1)}. \end{aligned}$$

$$\text{Thus } P[A] = \sum_{i=0}^{\infty} \frac{\rho^i \Gamma(\rho+1)}{\Gamma(\rho+i+2)} = \sum_{i=0}^{\infty} \frac{\rho^i}{i!} \frac{i! \Gamma(\rho+1)}{\Gamma(\rho+i+2)}$$

$$= \sum_{i=0}^{\infty} \frac{\rho^i}{i!} \frac{\Gamma(i+1) \Gamma(\rho+1)}{\Gamma(\rho+i+2)} = \sum_{i=0}^{\infty} \frac{\rho^i}{i!} \beta(i+1, \rho+1)$$

$$= \sum_{i=0}^{\infty} \frac{\rho^i}{i!} \int_0^1 x^{\rho} (1-x)^i dx = \int_0^1 x^{\rho} \sum_{i=0}^{\infty} \frac{\rho^i (1-x)^i}{i!} dx$$

$$= \int_0^1 x^{\rho} e^{\rho(1-x)} dx = e^{\rho} \int_0^1 x^{\rho} e^{-\rho x} dx$$

$$= e^{\rho} \int_0^1 x^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho x)^i}{i!} dx = e^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho)^i}{i!} \int_0^1 x^{\rho+i} dx$$

$$= e^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho)^i}{i!} \int_0^1 x^{\rho+i} (1-x)^0 dx = e^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho)^i}{i!} \beta(\rho+i+1, 1)$$

$$= e^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho)^i}{i!} \frac{\Gamma(\rho+i+1) \Gamma(1)}{\Gamma(\rho+i+2)}$$

$$= e^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho)^i}{i!} \frac{1}{(\rho+i+1)}$$

(5.11)

Q.E.D.

5.3.2 Accounting for the Feedback Loop

The derivation in the previous section assumed that when a transaction enters the network, it sees an average network load of $\rho = \lambda / \mu$, which is the expected number of customers in the system due to new arrivals. The expression $P[A]$ ignored the effects of rejected transactions which were 'lost' to the system (Figure 5.3(a)). In a basic T/O system, however, transaction rejections increase the traffic in the network and must be considered (Figure 5.3(b)). An iterative approach is used in solving for $P[\text{star}]$, the actual probability of acceptance in the feedback system. In Section 5.3 the validity of assuming a Poisson input stream at each iteration is discussed. Let

$$P[A|k] = e^{-k} \sum_{i=0}^{\infty} \frac{(-k)^i}{i! (k+i+1)} \quad \text{be the probability of acceptance given a network load } k,$$

r_i be the probability of rejection on iteration i ,

λ_i be the rate at which transactions enter the network on iteration i ,

ρ_i be the network load on iteration i , and

λ_{∞} and ρ_{∞} denote $\lim_{i \rightarrow \infty} \lambda_i$ and $\lim_{i \rightarrow \infty} \rho_i$, respectively, if they exist.

Now $P[\text{star}]$ can be evaluated as $P[A|\rho_{\infty}]$ using the following algorithm:

```

function star ( $\lambda$ )
  begin
     $\lambda_0 := \lambda$ 
    for  $i := 0$  to  $\infty$ 
       $\rho_i := \lambda_i / \mu$ 
       $r_i := 1 - P[A|\rho_i]$ 
       $\lambda_{i+1} := \lambda + \lambda_i r_i$ 
    next  $i$ 
    star :=  $P[A|\rho_{\infty}]$ 
  end star.
```

(a) Open System

(b) Closed Feedback System

Figure 5.3 Logical Description of Transaction Flow

Initially, $\lambda_0 = \lambda$, the rate due to new arrivals. At each iteration $\lambda_i r_i$ is the rate of transaction rejection and must be added to the fixed input rate to give the new rate of transactions entering the network. Let $\{\lambda_i\}$ represent the sequence formed by the λ_i values on each iteration. For $i > 0$

$$\begin{aligned}\lambda_{i+1} - \lambda_i &= (\lambda + \lambda_i r_i) - (\lambda + \lambda_{i-1} r_{i-1}) \\ &= \lambda_i r_i - \lambda_{i-1} r_{i-1} \\ &= \lambda_i r_i - (\lambda_i - \lambda) \\ &= \lambda_i (r_i - 1) + \lambda \\ &= \lambda - \lambda_i (1 - r_i) \\ &= \lambda - d_i\end{aligned}$$

The difference in the rate of arrivals to the network at each iteration is equal to the difference between the initial arrival rate λ and the departure rate $d_i = \lambda_i (1 - r_i)$. Since the rate of departure must be less than λ , we have

$$0 < d_i \leq \lambda$$

and therefore

$$\lambda_{i+1} \geq \lambda_i$$

Thus $\{\lambda_i\}$ is a monotonically increasing (or technically, non-decreasing) sequence. As λ_i increases, ρ_i , the average number of customers in the network, increases. Therefore, the rejection probability $r_i = P[A|\rho_i]$ will increase monotonically. Since r_i is bounded by 1, the sequence $\{r_i\}$ must converge.

Note that $P[\text{star}]$ cannot equal 0. For the set of transactions with timestamps greater than the current entity timestamp, at least one

transaction of the set must get accepted. Although this implies that r_i can never equal 1, it may be that some values of λ , the limiting or steady state probability of rejection

$$r_{\infty} = \lim_{i \rightarrow \infty} r_i = 1$$

Therefore λ_i will increase without bound. It is important to know for which values of λ , $r_{\infty} < 1$.

Definition. The basic T/O concurrency control environment is stable iff $r_{\infty} < 1$.

Note that stability as defined here does not imply practicality; even if $r_{\infty} < 1$, λ_{∞} may be too large for practical operation of a real system. Due to the complexity of $P[A]$, an intuitive condition for stability based on conservation of flow is derived using an informal argument.

Since $\{r_i\}$ is monotonically increasing and converges for all λ , we have

$$r_0 < r_{\infty} \leq 1$$

Therefore either $r_{\infty} = 1$ or $r_{\infty} < 1$. From the algorithm, it can be shown that for $n > 0$,

$$\lambda_n = \lambda \left(1 + \sum_{i=0}^{n-1} \prod_{j=i}^{n-1} r_j \right)$$

If $r_{\infty} < 1$, then assume that $r_i = r$ is a fixed rejection probability at each iteration. Then

$$\begin{aligned} \lambda_n &= \lambda \left(1 + \sum_{i=0}^{n-1} \prod_{j=i}^{n-1} r \right) \\ &= \lambda \left(1 + \sum_{i=0}^{n-1} r^{n-i} \right) \end{aligned}$$

$$= \lambda(1 + r^n + r^{n-1} + \dots + r^2 + r)$$

$$= \lambda \left(\sum_{i=0}^{n-1} r^i \right)$$

Therefore $\lambda_{\infty} = \lambda \sum_{i=0}^{\infty} r^i = \frac{\lambda}{1-r}$ since $r < 1$

and $\lambda_{\infty} = \lambda + r\lambda_{\infty}$

When the system is stable, the actual rate of transactions entering the network is just equal to the rate due to new arrivals plus the rate due to rejected transactions. Continuing, we get

$$\lambda = \lambda_{\infty}(1-r) = \lambda_{\infty}(1-r_{\infty}) = d_{\infty} \quad (5.12)$$

which says that in the steady state, if the system is stable, the rate of departure must equal the rate of arrival of new transactions. In other words, conservation of flow is satisfied. Conversely, if conservation of flow is not satisfied, the λ_i will increase without bound and therefore $r_{\infty}=1$.

The system is therefore stable if given λ , we can find λ_{∞} such that

$$d_{\infty} = \lambda_{\infty}(1-r_{\infty}) = \lambda_{\infty} e^{\rho_{\infty}} \sum_{i=0}^{\infty} \frac{(-\rho_{\infty})^i}{i!(\rho_{\infty}+i+1)} = \lambda \quad (5.13)$$

is true (where $\rho_{\infty} = \lambda_{\infty}/\mu$). To derive numeric solutions of $P[\text{star}]$ and test the above condition, algorithm $\text{star}(\lambda)$ is rewritten to account for a stopping condition based on $d_{\infty} = \lambda$.

function $\text{star}(\lambda)$

begin

$r_0 := 0$

$i := -1$

```

repeat
    i := i + 1
     $\lambda_i := \lambda + \lambda_i r_i$ 
     $\rho_i := \lambda_i / \mu$ 
     $r_i := 1 - P[A|\rho_i]$ 
     $d_i := \lambda_i (1 - r_i)$ 
    until  $d_i = \lambda$  or  $d_i = 0$ 
    star :=  $P[A|\rho_i]$ 
end star.

```

Although the above discussion was based on λ , the stopping condition in the following example is in terms of ρ . The following example demonstrates the iteration values for the parameters of interest given an initial value of $\rho = 1.0$. The stopping condition is $d_i/\mu = 1.0$.

i	ρ_i	$P[A \rho_i]$	$r_i = 1 - P[A \rho_i]$	d_i/μ
1	1.00000	.71828	.28172	.71828
2	1.39211	.66175	.33825	.92130
3	1.51114	.64731	.35269	.97818
4	1.54485	.64341	.35659	.99396
5	1.55423	.64233	.35767	.99833
6	1.55682	.64204	.35796	.99954
7	1.55754	.64196	.35804	.99987
8	1.55774	.64193	.35807	.99996
9	1.55780	.64193	.35807	.99999
10	1.55781	.64192	.35808	1.00000
11	1.55782	.64192	.35808	1.00000
12	1.55782	.64192	.35808	1.00000

Therefore for $\rho = 1.0$, the actual probability of being a star customer is .64192. Figure 5.4 compares the open system and feedback values of $P[\text{star}]$, as a function of ρ , the load due to new arrivals. The probability of acceptance is decreased in the feedback model due to an increase in the number of customers which can cause TC to be a nonstar customer.

Prob[customer is accepted] versus network load ρ (due to arrivals)

Figure 5.4 Probability of Acceptance for Open and Feedback Systems

It is interesting to note the similarity of the function $P[\text{star}]$ (5.11) to P^* (5.4) for conservative T/O.

$$\begin{aligned}
 P^* &= \frac{1 - e^{-\rho}}{\rho} \\
 &= \frac{e^{-\rho}}{\rho} (e^{\rho} - 1) = \frac{e^{-\rho}}{\rho} \sum_{i=1}^{\infty} \frac{\rho^i}{i!} \\
 &= e^{-\rho} \sum_{i=0}^{\infty} \frac{\rho^i}{(i+1)!} = e^{-\rho} \sum_{i=0}^{\infty} \frac{\rho^i}{i!} \frac{1}{(i+1)} \\
 P[\text{star}] &= e^{\rho} \sum_{i=0}^{\infty} \frac{(-\rho)^i}{i!} \frac{1}{(\rho+i+1)}
 \end{aligned}$$

Figure 5.5 shows a graph of both curves for various values of ρ . Both expressions are products of an exponential function and a damped exponential function whose exponent is of the opposite sign. The expression for P^* , however, reduces to a simpler form than the expression for $P[\text{star}]$ due to the absence of ρ in the denominator of the summation term.

In conservative T/O the state of the system upon the arrival of a tagged customer (TC) is defined in terms of n , the number of customers with smaller timestamps which are still in the network. The two possible state transitions of TC are to a star or nonstar state. In basic T/O when TC is in state n there are three possible transitions: to a star state (A), to a nonstar state (R) or to state $n+1$. The possible transition to the $n+1$ state (rather than having a fixed n as in conservative T/O) accounts for the $\rho+i+1$, rather than $i+1$, as the damping factor in the denominator of the summation term.

Equation 5.11 belongs to a class of functions called incomplete gamma functions which can only be expressed as infinite sums or

Figure 5.5 Probability of a Star Customer for Conservative and Basic T/O

integrals and are difficult to manipulate in further derivations. (The chi-squared function used in statistics is an example of an incomplete gamma function.)

5.3.3 Concerning the Feedback Process

In the evaluation of $P[\text{star}]$ using the feedback model, the input stream of transactions entering the network was assumed to be a Poisson process at each iteration. Disney [DISN80] presents an analysis of five random flow processes (Figure 5.6) in an M/G/1 server model with instantaneous Bernoulli feedback. After each service period, a Bernoulli decision is made which causes each customer to either depart the system (with probability q) or be fed back into the input stream (with probability $1-q$). The derivation of $P[\text{star}]$ was based on the assumption that, at each iteration, the feedback process is Poisson and independent of the Poisson arrival process, thus making the input process Poisson as well. Disney shows that for an M/M/1 queue with instantaneous feedback, either one or both of the following is true:

1. the feedback process is not a Poisson process, and/or
2. the arrival process and the feedback process are not independent.

Although our analysis uses an infinite (rather than single) server model, the implication is that the assumption of a Poisson input process is possibly incorrect. To test the accuracy of the model, a simulation was conducted based on the feedback model of Figure 5.3(b) using SIMPAC [SIMP78] subroutines embedded in FORTRAN code. Figure 5.7 compares the theoretical and simulation values of $P[\text{star}]$ for various values of ρ . In each case the simulation was run for $5000/\lambda$ time units (or about

Figure 5.6 Random Processes of a Queueing System with Feedback [DISN80]

E[Y]	$\rho = \frac{E[X]}{E[Y]}$	P[star] predicted	P[star] simulation value		
			Run 1	Run 2	confidence limits on Run 2 values
10000	.01	.9950	.9964	.9956	(.9944, .9969)
5000	.02	.9901	.9892	.9905	(.9875, .9934)
2000	.05	.9754	.9765	.9761	(.9733, .9789)
1000	.10	.9517	.9503	.9501	(.9451, .9552)
500	.20	.9065	.9005	.9095	(.9033, .9156)
200	.50	.7890	.8008	.7846	(.7781, .7911)
100	1.00	.6419	.6571	.6469	(.6367, .6571)
50	2.00	.4614	.4547	.4619	(.4511, .4728)
20	5.00	.2467	.2546	.2491	(.2423, .2559)

NOTES:

1. $E[X] = 100$.

2. Run 1 consisted of approximately 5000 transaction arrivals (after an initial 1000 arrivals to allow the system to stabilize) for each value of E[Y].

3. Run 2 consisted of ten independent runs of 1000 transaction arrivals (after an initial 1000 arrivals to eliminate transient effects) for each value of E[Y]. Because each run is independent the sample variance can be calculated without an adjustment for correlation among the samples. The confidence limits make use of the sample variance; therefore the Student's-t distribution with 9 degrees of freedom is used in calculating the confidence intervals.

Figure 5.7 Theoretical and Simulation Results for P[star]

5000 transaction arrivals) after an extra $1000/\lambda$ time units to allow the system to stabilize. The simulation results compare favourably with the theoretical results (satisfying the 95% confidence limits for each case in Run 2). The closeness of the theoretical and simulation results validate the use of the Poisson assumption in the iterative model. The derived expression for $P[\text{star}]$ shall be used as a good approximation upon which to base further results.

5.4 Results Based on $P[\text{star}]$

We are now in a position to evaluate:

- a) the expected number of retries (and thus messages needed) before a transaction gets accepted, and
- b) the expected delay before a transaction gets accepted.

In determining a steady state probability of acceptance it was implicitly assumed that the acceptance decision was independent of any past attempts by the same transaction. This assumption is reasonable since each rejected transaction gets a new timestamp based on the same mechanism as that for new arrivals. Thus, adaptive mechanisms which favour previously rejected transactions are not being considered. The decision mechanism can therefore be approximated as a Bernoulli switch with $P[\text{success}] = P[\text{star}] = 1-r$ and $P[\text{failure}] = r$. For each transaction, let R represent the number of rejections before being accepted. R is a geometric random variable with discrete pdf

$$P[R=n] = (1-r)r^n \quad \text{for } n = 0, 1, 2, 3, \dots$$

and the expected number of rejections for each transaction is

$$E[R] = \sum_{n=0}^{\infty} n(1-r)r^n = \frac{r}{1-r} \quad \text{for } r < 1 \quad (5.14)$$

Define T_1 to be the expected delay before a transaction gets accepted. Since a transaction must traverse the network twice for each rejection (TM-to-DM and DM-to-TM), the expected delay, $E[T_1 | R=n]$, for a transaction with n rejections is $(2n+1)E[X]$ where $E[X]$ is the average network delay. Therefore, $E[T_1]$, the expected delay before a transaction is accepted is given by

$$\begin{aligned}
 E[T_1] &= \sum_{n=0}^{\infty} E[T_1 | R=n] P\{R=n\} \\
 &= \sum_{n=0}^{\infty} (2n+1)E[X]r^n(1-r) \\
 &= E[X](1-r) \left\{ \sum_{n=0}^{\infty} r^n + 2 \sum_{n=0}^{\infty} nr^n \right\} \\
 &= E[X](1-r) \left\{ \frac{1}{1-r} + \frac{2r}{(1-r)^2} \right\} \\
 &= E[X] \left\{ 1 + \frac{2r}{1-r} \right\} \\
 &= E[X] \left\{ \frac{1+r}{1-r} \right\} \quad \text{for } r < 1
 \end{aligned}$$

Figure 5.8 plots $E[R]$ against the load parameter ρ . $E[R]$ is a measure of the number of overhead messages needed due to rejections in basic T/O. (Remember, conservative T/O has no rejections.) Figure 5.9 compares the normalized delay $E[T_1]/E[X]$ for conservative T/O against basic T/O. Except for values of $\rho < 1$, the basic T/O scheme seems to impose a large synchronization penalty due to its geometric behaviour. Figure 5.10 compares the predicted and simulation values of $E[R]$ and $E[T_1]/E[X]$. The assumption of the independence of each attempt is validated by the simulation results.

Expected Number of Retries, $E[R]$ versus ρ
for Basic T/O

Figure 5.8 Expected Retries for Basic T/O

Normalized Network Delay versus ρ

Figure 5.9 Network Delay for T/O Schemes

	E[R]			E[T ₁]/E[X]		
	(1)	(2)	(3)	(1)	(2)	(3)
.01	.005	.003	.003	1.010	1.001	1.007
.02	.010	.011	.011	1.020	1.015	1.022
.05	.025	.024	.024	1.050	1.043	1.048
.10	.051	.052	.052	1.102	1.109	1.105
.20	.103	.111	.110	1.206	1.232	1.221
.50	.267	.249	.249	1.535	1.500	1.498
1.00	.558	.521	.522	2.116	2.018	2.044
2.00	1.17	1.20	1.20	3.335	3.410	3.399
5.00	3.05	2.92	2.93	7.106	6.765	6.850

NOTES:

- (1) the predicted value based on the theoretical value of P{star}
- (2) the recorded simulation average value
- (3) the predicted value based on mean simulation value of P{star}

Figure 5.10 Simulation Results for Number of Retries (E[R])
and Normalized Network Delay (E[T₁]/E[X])

The basic T/O and the LeLann scheme as currently modelled cannot be directly compared. The LeLann conservative T/O scheme imposes an extra delay which has not been considered in [KAM081]. Due to the virtual ring and token mechanism needed for consecutive timestamp generation, a transaction cannot be submitted until the TM receives the token. As well, in a nonconsecutive timestamp generation scheme (as used in SDD-1), transaction eligibility is not just based on being a (LeLann) star customer. Instead, each transaction has to wait for a transaction with a larger timestamp from each TM, thus increasing the expected delay. In Section 5.6 the node delay experienced by each customer is shown to strongly affect the expected delay for basic T/O.

5.5 Multiple DM Requests

In this section the analysis of $P[\text{acc}]$ is extended to include the effect of multiple DM nodes in the DDB. Assume that the DDB is composed of n DMs and that each transaction accesses an average of α DMs where $1 \leq \alpha \leq n$. The system is now as depicted in Figure 5.11. All transactions sent to a DM are assumed to conflict. This is a conservative assumption since, in fact, each DM manages many stored entities and transactions conflict only if their read-writesets conflict. For each transaction which accesses α nodes, assume a uniform probability of access across the n nodes such that

$$P[T \text{ accesses } DM_i] = \alpha/n \quad \text{for } i=1 \text{ to } n.$$

Then the rate of transaction arrival (and thus conflict) at each DM is $\lambda\alpha/n$. Since each DM bases the acceptance decision on the relative arrival times of transactions it sees, the probability of local

Figure 5.11 Multiple DM Accesses

acceptance at each DM is

$$P[\text{local acceptance}] = P[A | \frac{\alpha \rho}{n}]$$

Since $\alpha/n \leq 1$, $P[A | \alpha \rho / n] \geq P[A | \rho]$. The infinite server network guarantees that the acceptance/rejection of each transaction is performed independently at each DM. Since a transaction is accepted globally only if it is accepted at all DMs, the probability of (global) acceptance is

$$P[\text{acc}] = P[A | \frac{\alpha \rho}{n}]^\alpha \quad (5.15)$$

For $n = 1$ this reduces to $P[\text{acc}] = P[\text{star}]$.

Figure 5.12 plots $P[\text{acc}]$ against ρ for $\alpha = 1$ and $n = 1, 2, 3, 5$ and 10 . Figure 5.13 plots $P[\text{acc}]$ against ρ for $n = 10$ and $\alpha = 1, 2, 3, 4.5$ and 5.5 . As expected, $P[\text{acc}]$ increases as n gets larger and decreases as α gets larger. For a given DDB, concurrency can be increased and the rejection rate decreased by using a finer level of granularity to increase n . At the same time α , the size of the read-writeset, is increased. Therefore, an optimal granularity for the DDB must be determined.

In Figure 5.13, the rapid drop in $P[\text{acc}]$ as α increases indicates a possible instability in the basic T/O concurrency control. The iterative algorithm of Section 5.3.2 is again used to calculate $P[\text{acc}]$. For $n = 1$ it was found numerically that on each iteration the rate of departure increased until $\lambda_\infty(1-r_\infty) = \lambda$ and conservation of flow was satisfied. For $n > 1$ we find that for some α and ρ the rate of departure may start to decrease until $r_\infty = 1$, thus violating the conservation of flow condition. The following example shows the decreasing departure

Probability of Global Acceptance for Multi-DM Accesses

$$P[\text{acc}] = P[A | \frac{\alpha \alpha}{n}] \alpha \quad \text{for } \alpha = 1$$

Figure 5.12 $P[\text{acc}]$ for Multiple DM System; $\alpha = 1$

Probability of Global Acceptance ($P[\text{acc}]$)

$$P[\text{acc}] = P\left[\Lambda \left| \frac{\alpha \alpha}{n} \right| \right] \quad \text{for } n = 10$$

Figure 5.13 $P[\text{acc}]$ for Multiple DM System; $n = 10$

rate after 4 iterations for $n=10$, $\alpha=3$ and initial load $\rho = 2.0$. As previously the conservation of flow (stopping) condition used is in terms of the load ρ , that is, $d_1/\mu = 2.0$.

i	P[acc]	P[rej]	d_1/μ	$\frac{\alpha p_i}{n}$
1	.50583	.49417	1.01166	.60000
2	.32783	.67217	1.29621	1.18617
3	.23055	.76945	1.40649	1.83020
4	.16644	.83356	1.44388	2.60251
5	.11966	.88034	1.43791	3.60487
6	.08361	.91639	1.39743	5.01405
7	.05530	.94470	1.32281	7.17609
8	.03346	.96654	1.21006	10.84975
9	.01763	.98237	1.05374	17.93255
10	.00751	.99249	.85154	34.03614
11	.00230	.99770	.61271	79.93979
12	.00042	.99958	.36779	260.93991
13	.00004	.99996	.16696	1418.95782
14	.00000	1.00000	.04972	16997.17421
15	.00000	1.00000	.00367	683686.37197
16	.00000	1.00000	.00000	372272165.21000
17	.00000	1.00000	.00000	3.0904151941E+16

The possible instability points out that basic T/O is not suitable for pessimistic or high conflict environments, in which case conservative T/O (delay) is perhaps a better strategy than basic T/O (restarts).

5.6 Effects of DM Delay in Basic T/O

In this section the effect of DM delay on the probability of acceptance and the expected delay is investigated for the single DM case. Until now, the execution of transactions at the DM has been assumed to happen instantaneously. In actual fact, transactions experience delay at the DM (in the RAT) queueing for resources and (in the multiple DM case) awaiting global synchronization. As a result, transactions which are nonstar can be accepted as long as no transaction

with a later timestamp has yet been processed by the DM. An out-of-sequence transaction is re-ordered by the DM which maintains the RAT queues in timestamp (rather than FIFO) order. Figure 5.14 depicts this new structure as two queueing systems in series: a network portion and a node portion.

In Section 5.3 the probabilities of acceptance and rejection of each transaction were defined in terms of conditional probabilities as

$$P[\text{acc}] = P[\text{acc}|\text{star}]P[\text{star}] + P[\text{acc}|\text{nonstar}]P[\text{nonstar}] \quad (5.16)$$

$$\text{and } P[\text{rej}] = P[\text{rej}|\text{star}]P[\text{star}] + P[\text{rej}|\text{nonstar}]P[\text{nonstar}]. \quad (5.17)$$

Whereas previously it was assumed that

$$P[\text{acc}|\text{nonstar}] = 0, P[\text{rej}|\text{nonstar}] = 1 \text{ and thus } P[\text{acc}] = P[\text{star}]$$

we now have

$$P[\text{acc}|\text{star}] = 1 \text{ and } P[\text{acc}|\text{nonstar}] \geq 0$$

$$\text{and } P[\text{rej}|\text{star}] = 0 \text{ and } P[\text{rej}|\text{nonstar}] \leq 1.$$

Thus the probabilities of acceptance and rejection become

$$P[\text{acc}] = P[\text{star}] + P[\text{acc}|\text{nonstar}]P[\text{nonstar}] \quad (5.18)$$

$$\text{and } P[\text{rej}] = P[\text{rej}|\text{nonstar}]P[\text{nonstar}]. \quad (5.19)$$

In order to evaluate $P[\text{acc}]$, we need to evaluate $r' = P[\text{rej}|\text{nonstar}]$ whose complement represents the probability that an out-of-sequence transaction will arrive before processing has occurred on any transaction with a larger timestamp. Partial results towards a derivation of r' are reported in the Appendix. The analytic difficulties encountered prompted the use of simulation as a tool to investigate the effects of DM delay.

Under the present assumption all transactions conflict at the DM node which can only service one transaction at a time. If we assume

Figure 5.14 Network and Node Queueing Systems in Series

exponential DM service time Z , the node can be modelled as a G/M/1 server with

$$f(z) = \alpha e^{-\alpha z}, \quad z \geq 0.$$

Define the network load parameter $\rho_1 = \lambda / \mu$ and the DM load parameter $\rho_2 = \lambda / \alpha$. A set of thirty-three simulation runs were conducted with $E[X] = 1/\mu = 100$. The simulation values chosen represent an assumed fixed network server rate $E[X] = 100$ msec. The average interarrival time $E[Y]$ and the average node server time $E[Z]$ were varied as indicated to model the range of ρ_1 and ρ_2 shown in Figure 5.15. As before, each simulation generated approximately 5000 transaction arrivals. Figure 5.16 shows the values of $r = P[\text{rej}|\text{nonstar}]$ over the complete set of simulations. For large values of ρ_2 , as the DM queue size and expected DM delay increases, the probability of a nonstar customer being accepted increases and thus r decreases from 1.

Let T_1 be the time a transaction is delayed before being accepted and let T_2 be the time a transaction spends at the node before completing DM service. Figure 5.17 graphs the normalized network delay, $E[T_1]/E[X] = (1+r)/(1-r)$, for various values of $E[Z]$, where $r = P[\text{rej}] = P[\text{rej}|\text{nonstar}]P[\text{nonstar}]$. For conservative T/O schemes, the network delay is not affected by node delay. For basic T/O schemes, however, the network delay is a function of $P[\text{rej}|\text{nonstar}]$ and therefore of the node delay. Thus the geometric behaviour of expected delay for basic T/O is counteracted by a reduction in $P[\text{rej}|\text{nonstar}]$ for large values of ρ_2 .

For conservative T/O, nonstar customers become eligible for service only upon the arrival of a star customer. The star customer, plus any nonstar customers that become eligible, constitute a bulk arrival.

$E[X]=100$

$$\rho_2 = \frac{E[Z]}{E[Y]}$$

$$\rho_1 = \frac{E[X]}{E[Y]}$$

$E[Z] \backslash E[Y]$	800	400	160	80	40	16	
10000	.08	.04	.016	-	-	-	.01
5000	.16	.08	.032	.016	-	-	.02
2000	.4	.2	.08	.04	.02	-	.05
1000	.8	.4	.16	.08	.04	.016	.10
500	-	.8	.32	.16	.08	.032	.20
200	-	-	.8	.4	.2	.08	.50
100	-	-	-	.8	.4	.16	1.00
50	-	-	-	-	.8	.32	2.00
20	-	-	-	-	-	.8	5.00

Figure 5.15 Values of ρ_1 and ρ_2 Used in Simulations

$$E[X] = 100$$

E[Y] \ E[Z]	E[Z]						$\rho = \frac{E[X]}{E[Y]}$
	800	400	160	80	40	16	
10000	.952	1.0	1.0	-	-	-	.01
5000	.909	.931	.976	1.0	-	-	.02
2000	.619	.831	.962	1.0	.992	-	.05
1000	.215	.667	.914	1.0	.992	1.0	.10
500	-	.267	.797	.958	.981	1.0	.20
200	-	-	.380	.854	.965	.999	.50
100	-	-	-	.439	.882	.990	1.0
50	-	-	-	-	.562	.972	2.0
20	-	-	-	-	-	.650	5.0

Figure 5.16 $P[\text{rej}|\text{nonstar}]$ for various $E[Y]$ and $E[Z]$

Normalized Network Delay (WT_1): Conservative T/O versus Basic T/O

Figure 5.17 Effects of Node Delay for Basic T/O

process to the DM server where the bulk size is $G = 1/P^*$ and the rate of arrival of bulks is λP^* [KAM081]. Figure 5.18 tabulates the ratio of $E[W]$ for conservative T/O to that of basic T/O for the simulation runs. In general, the ratios are greater than 1 because of the bulk arrival process for conservative T/O. Therefore, $E[T_2] = E[W] + E[Z]$ will be greater for conservative T/O than for basic T/O due to a greater expected waiting time, $E[W]$.

Figure 5.19 presents the total delay $T = T_1 + T_2$ for both T/O schemes. Contrary to what was suggested by Figure 5.9, basic T/O compares favourably with conservative T/O. Although the intention in Figure 5.19 is not to portray an exact comparison of the T/O schemes, it does illustrate how the mechanisms in each scheme affect the overall delay.

The reduction in restarts due to inherent DM delay suggests that further 'intentional' delay may be useful as an heuristic to reduce $E[T]$. As the delay experienced by each transaction increases, the probability of rejection will decrease. We are thus trading off an intentional increase in T_2 for a resultant decrease in T_1 hoping to reduce the overall value of T . The mechanism of reservations used in Chapter 4 to reduce restarts is an example of intentional delay.

To illustrate the effect of delay, a simulation was conducted in which the DM server was replaced by a delay process. Each transaction experienced a minimum delay of d time units (plus some queueing delay in waiting for transactions with smaller timestamps to become eligible) after which processing occurred instantaneously. Figure 5.20 plots

$$E[X] = 100$$

E[Y] \ E[Z]							$\frac{E[X]}{E[Y]}$
	800	400	160	80	40	16	
10000	1.1	.9	1.7	-	-	-	.01
5000	1.0	1.0	1.2	2.1	-	-	.02
2000	1.1	1.0	1.1	1.9	2.2	-	.05
1000*	1.2	1.1	1.1	1.6	2.4	32.9	.10
500	-	1.1	1.2	1.4	2.7	35.5	.20
200	-	-	1.1	1.3	2.6	9.5	.50
100	-	-	-	1.1	2.0	3.7	1.0
50	-	-	-	-	1.4	4.6	2.0
20	-	-	-	-	-	1.4	5.0

Figure 5.18 Ratio of E[W] Values for Conservative T/O and Basic T/O

Total Delay (T) for Conservative T/O and Basic T/O

Figure 5.19 Total Delay for Both T/O Schemes

$P[\text{rej}]$ and Figure 5.21 tabulates T for various ρ_1 and d . Although $P[\text{rej}]$ decreases as d increases, $T(\rho_1, d) < T(\rho_1, d=0)$ only for large values of ρ_1 which is precisely when we expect inherent delay due to queueing at the DM. The minimization of T by imposing delay as an heuristic is an area that warrants further research since delay mechanisms could be an important tuning parameter for basic T/O concurrently control.

5.7 Effects of Non-Exponential Server

The probability of rejection is a function of the amount of disorder caused by the network, which is itself related to the network service density function. An exponential distribution has coefficient of variation $C = \sigma/E[X] = 1$ where σ and $E[X]$ are the standard deviation and mean of the service distribution respectively. In commercial networks an exponential service time distribution is unacceptable due to the end user's desire for predictability of delay. We investigate two network service distributions which may more realistically represent actual commercial networks:

1. a uniform pdf

$$f(x) = \frac{1}{B-A}, \quad A < x < B$$

$$= 0, \quad \text{otherwise}$$

2. a truncated exponential pdf

$$= \alpha e^{-\mu x}, \quad A < x < B$$

$$= 0, \quad \text{otherwise}$$

$$\text{where } \alpha = 1/(e^{-\mu A} - e^{-\mu B})$$

Figure 5.20 Effect of Delay on Probability of Rejection

ρ_1	Delay, d.					
	0	10	50	100	500	1000
.01	100.7	114.8	154.9	201.2	602.7	1101.8
.02	106.6	112.7	150.9	206.6	600.4	1100.6
.05	108.3	113.1	150.5	196.2	603.8	1099.5
.10	101.3	120.4	152.5	207.4	602.9	1106.8
.20	114.4	127.3	158.4	202.3	609.9	1107
.50	150.0	155.6	175.1	217.1	627.4	1121.8
1.0	209.4	201.1*	195.6*	233.8	634.8	1130.9
2.0	304.3	302.1*	223.2*	252.7*	679.9	1179.5
5.0	624.3	476.3*	251.6*	269.2*	752.7	1224.2

NOTE: * signifies those ρ_1 and d such that $T(\rho_1, d) < T(\rho_1, d=0)$

Figure 5.21 Total Delay (T) for Various Node Delays (d)

In both cases we wish to find A and B such that $E[X] = 100$ in order to compare results with our previous simulations.

$$\text{In the uniform case } E[X] = \frac{B+A}{2} \text{ and } \text{var}(X) = \frac{(B-A)^2}{12}$$

$$\text{Therefore the coefficient of variation is } C = \frac{\sigma}{E[X]} = \frac{B-A}{(B+A)\sqrt{12}}$$

For $A = 50$ and $E[X] = 100$ we have $B = 150$, $\sigma = 28.87$ and $C = 0.2887$.

For the truncated exponential we can show that

$$E[X] = \alpha(Ae^{-\mu A} - Be^{-\mu B}) + 1/\mu$$

$$\text{and } \text{var}(X) = \alpha(A^2e^{-\mu A} - B^2e^{-\mu B}) + (1/\mu)^2.$$

$$\text{Now, } E[X] = 1/\mu \text{ implies that } Ae^{-\mu A} = Be^{-\mu B} \quad (5.20)$$

where $xe^{-\mu x}$ is a skew bell-shaped curve with maximum value at $x = 1/\mu$.

We wish to find $A < 1/\mu < B$ to satisfy Eq. 5.20. Choosing $A = 50$ and $1/\mu = 100$ we find (numerically) that $B = 175.64$, $\text{var}[X] = 1186.12$ and thus $C = 0.3458$.

Figure 5.22 shows the result of a simulation assuming no DM delay ($E[Z]=0$) for the exponential, uniform and truncated exponential network service pdfs. As expected, the probability of rejection is lower for the service distributions with smaller variances. In fact, at the extreme ($C=0$ for a deterministic service distribution) no disorder would occur.

Figure 5.22 P[nonstar] for Various Service Distributions

CHAPTER 6

SUMMARY AND SUGGESTIONS FOR FURTHER RESEARCH

The major contributions of this research can be summarized as follows:

1. the development of ADD*, an extension of the ADD protocol to handle dynamic request generation;
2. a formal analysis of the ADD and ADD* protocols;
3. the development of an M/M/ ∞ model with feedback to investigate transaction disordering in a network; and
4. an investigation of factors contributing to transaction rejection for basic timestamp ordering concurrency control.

Throughout the thesis, questions were raised relating to unanswered issues in the literature. Rather than repeat those concerns here, we present possible extensions to the performance work in Chapter 5 as suggestions for further research.

6.1 Summary of Research

6.1.1 The ADD* Protocol

The need for a dynamic (nonpredeclarative) request capability motivated an extension of the ADD basic T/O concurrency control mechanisms. The functioning of the new protocol ADD* was modified from ADD to reflect that each dmread and precommit/dmwrite is considered atomic and requires synchronization. In the dynamic case, the synchronization and execution of transactions is not separated. Transaction processing model TP3, with two-phase commit and delayed updates, was integrated into synchronization procedure to provide atomic

commitment and prevention of rollback. Since a rejection of any operation led to a complete restart of the transaction, timestamp updates were not possible. Thus a procedure for aborting all in progress operations was presented for handling restarts. The version of basic T/O in ADD* incorporated separate read (R-TS) and write (W-TS) timestamps in order to reduce restarts and allow for implementation of Thomas Write Rule optimization. Suggestions for reducing restarts in ADD* based on partial read-writeset knowledge were presented.

6.1.2 Protocol Analysis

The ADD and ADD* protocols were described in detail using pseudocode and finite state machine specification techniques. The protocols were defined by describing a master TCP at the initiating TM node and a cooperating slave TCP at the DM nodes. Each TCP was described in terms of:

1. a general description of the protocol;
2. the internal structure of each cooperating module;
3. the types and formats of messages exchanged; and
4. the rules governing the reactions to each message or event.

The finite state machine models were used in determining possible causes of nontermination of the protocol.

To verify the protocols formal arguments were used to show that:

1. each transaction is committed based on the same timestamp values; and
2. given the unique timestamp, the basic T/O mechanism correctly serializes transactions at all DMs to produce an acyclic ordering.

Inspection of the FSM representation, showed that the ADD and ADD* protocols are subject to multiple rejections. Rejection timestamps and reservations were suggested to reduce the probability of rejections in ADD. The dynamic request scheme, however, prevented the use of these mechanisms in ADD*. Instead, the rejection rate in ADD* could be reduced by:

1. using separate read and write timestamps;
2. implementing the Thomas Write Rule;
3. allowing read with intent to update; and
4. imposing intentional delays before processing transactions.

6.1.3 The M/M/∞ Feedback Model

The effect of transaction disordering due to the stochastic nature of the network was studied using an M/M/∞ model with feedback. A star customer was defined as a one which left the network before all customers with larger timestamps. In considering the transactions which can cause an arbitrary customer TC to be a nonstar customer, the assumption of synchronized TM clocks, allowed us to consider only those customers entering the network after TC. Initially a single DM node was assumed. $P[A]$, the probability of a star customer ignoring the feedback loop, was derived, and was seen to be an incomplete gamma function.

An iterative approach was used to consider the effects of the feedback loop. A recursive solution to the iterative equation

$$\lambda_{i+1} = \lambda + \lambda_i r_i = \lambda + \lambda_i (1 - P[A | \lambda_i / \mu])$$

in terms of λ_i was abandoned due to the inherent difficulty in

manipulation of incomplete gamma functions. The basic T/O concurrency control was defined as stable if

$$r_{\infty} = \lim_{i \rightarrow \infty} r_i < 1$$

A condition for stability based on conservation of flow was derived: Given an input rate of arrival λ , the system is stable if there exists λ_{∞} such that

$$d_{\infty} = \lambda_{\infty} P[A | \lambda_{\infty} / \mu] = \lambda$$

The probability of a star customer in the feedback model was defined as

$$P[\text{star}] = \lim_{i \rightarrow \infty} P[A | \rho_1]$$

The model was extended to consider the effects of accessing α out of n DM nodes. Numeric examples were presented for a stable and unstable environment (for some values of λ , α , and n).

The feedback model relied on the assumption of a Poisson input stream at each iteration. Although Disney showed this assumption to be incorrect for an M/M/1 server with feedback, simulation indicated that the expression derived for $P[\text{star}]$ was indeed a good approximation.

Due to the independence of transaction attempts, a geometric model was used to derive the expected number of rejections and the expected delay before being accepted for each transaction.

6.1.4 Effects of DM Node Delay

Rejections in basic T/O caused an undesirable behaviour in terms of greater expected network ~~delay~~ when compared with conservative T/O. Simulation was used to show that while the node delay had no effect on conservative T/O, it did serve to reduce rejections for basic T/O. The

initial model assumed that transaction acceptance was based strictly on the node arrival order. In reality, queueing due to processing and synchronization at the DM allows out of order transactions to be accepted as long as the entity timestamp has not been updated. Thus we have

$$P[\text{acc}] = P[\text{star}] + P[\text{acc}|\text{nonstar}]P[\text{nonstar}].$$

Simulation was used to show that for large values of the geometric behaviour of network delay was counteracted by an increase in the probability of acceptance for nonstar customers. In fact, in terms of the overall delay, neither basic T/O nor conservative T/O seemed to outperform the other. A more detailed analysis of both schemes is required before any firm conclusions can be drawn.

6.2 Suggestions for Further Research

The work of Bernstein and Goodman [BER80a] in classifying the concurrency control problem has been valuable towards analyzing the performance behaviour of the many schemes in the literature. In categorizing the proposals, the possible space of concurrency controls to be analyzed has been made more manageable. Attempts at performance analysis will tend to be inconclusive until a common base of assumptions is established to analyze the various concurrency control techniques. The analysis used for ADD and ADD* used assumptions whose full effects need to be investigated. These are presented below as areas for further research.

1. The assumption that all transactions entered the network in timestamp order implies that:

- a) the timestamp generation mechanisms at all TM nodes are perfectly synchronized; and
- b) all transactions with a given timestamp are submitted in parallel.

The first assumption is satisfied by a physical scheme such as the LeLann token and ring mechanism. General timestamp generation schemes, however, cannot maintain the TM clocks in perfect synchrony. The second assumption is satisfied by predeclaration of read-writesets as in the ADD protocol. In ADD* many subtransactions may be submitted over a period of time for a given transaction, each using the timestamp assigned at the initiation phase. If one or both of the above assumptions is not satisfied the model predictions will be optimistic. An investigation into the effects of relaxing either assumption is needed before conclusive performance results can be stated for basic T/O.

2. Preliminary attempts at solving $P[\text{rej}|\text{nonstar}]$ led to an intractable analysis. An investigation leading to a solution might consider using approximations to reduce the analysis to manageable proportions. Approximate solutions to $P[A|\rho_\infty]$ might also be investigated.

3. Many commercial networks offer virtual circuits or guaranteed message ordering between any pair of nodes. Use of the infinite server model assumes that any two transactions can depart the network out of relative order. This assumption may only be valid for networks with high connectivity and no virtual circuits. Further analysis might

investigate the effects of other network models.

4. The variance in the network service distribution was shown to have an effect on $P[\text{star}]$ in Section 5.7. The analysis can be generalized to consider nonexponential arrival and network service distributions.

APPENDIX

In Section 5.6 the effect of DM delay on

$$P[\text{acc}] = P[\text{star}] + P[\text{acc}|\text{nonstar}]P[\text{nonstar}]$$

was shown through simulation. To evaluate the above expression analytically

$$r' = P[\text{rej}|\text{nonstar}] = 1 - P[\text{acc}|\text{nonstar}]$$

must be evaluated. Some preliminary results are reported here.

To evaluate r' , we define the pdfs of two random variables: X_r , the residual network service time of a nonstar customer, and W , the DM waiting time of each customer. Specifically,

$$\begin{aligned} r' &= P[\text{rej}|\text{nonstar}] \\ &= P[\text{residual service time of nonstar customer} > \text{waiting time of} \\ &\quad \text{any customer leaving the network before the nonstar customer}] \\ &= P[X_r > W]. \end{aligned}$$

From [KLEI75], the pdf of W can be determined for an G/M/1 node model once the pdf of $A(x)$, the arrival process at the DM, is known. The analysis in [DISN80] suggests that $A(x)$ may be Poisson for our feedback system if the acceptance mechanism is a Bernoulli switch.

The X_r process is the residual network time of X_n , the a posteriori conditional service distribution given a nonstar customer. X_r can then be derived from X_n as [KLEI75]

$$f(X_r) = \frac{1 - F(X_n)}{E[X_n]}$$

From [PAPO65] the pdf of X_n is

$$f(x|\text{nonstar}) = \frac{f(x)P[\text{nonstar} | x = \tau]}{P[\text{nonstar}]}$$

Since $f(x)$ and $P[\text{nonstar}]$ are known, it remains to evaluate $P[\text{nonstar} | x = \tau]$, the probability that customer is nonstar given its network service time τ . Let

TC be a tagged customer entering the network at time 0,

C_i be the i^{th} customer to enter the network after TC,

t_i be the time of arrival of C_i , where $t_i > 0$,

s_i be the network service time of C_i , and

$$X_i = t_i + s_i.$$

Now s_i is an exponential random variable (r.v.) with p.d.f.

$$f_{s_i}(t) = f_s(t) = \mu e^{-\mu t}, \quad t > 0$$

The r.v. t_i is the sum of i independent exponential interarrival r.v.'s and is thus a gamma r.v. with p.d.f.

$$f_{t_i}(t) = \frac{\lambda e^{-\lambda t} (\lambda t)^{i-1}}{(i-1)!} \quad t > 0$$

Then X_i is the sum of $i+1$ independent r.v.'s (i of which have mean $1/\lambda$ and one having mean $1/\mu$). It can be shown that X_i is a gamma r.v. with p.d.f.

$$\begin{aligned} f_{X_i}(t) &= \left(\frac{\lambda}{\lambda - \mu} \right)^i \left[f_s(t) - \frac{\mu}{\lambda} \sum_{j=1}^i \left(\frac{\lambda - \mu}{\lambda} \right)^{j-1} f_{t_j}(t) \right], \quad \text{for } \mu \neq \lambda \\ &= \frac{\lambda e^{-\lambda t} (\lambda t)^i}{i!}, \quad \text{for } \mu = \lambda \end{aligned}$$

and Laplace transform

$$X_i^* = \left(\frac{\mu}{s+\mu} \right) \left(\frac{\lambda}{s+\lambda} \right)^i$$

with $E[X_i] = n/\lambda + 1/\mu$.

Given the service time of TC, $X = \tau$, we can calculate

$$P[\text{nonstar} | x=\tau] = \prod_{i=1}^{\infty} P[X_i > \tau]$$

where we can show that

$$P[X_i > \tau] = \int_{\tau}^{\infty} f_{X_i}(t) dt$$

$$= \left(\frac{\lambda}{\lambda-\mu} \right)^i \left[e^{-\mu\tau} - \frac{\mu}{\lambda} e^{-\lambda\tau} \sum_{k=0}^{i-1} \left(\frac{\lambda-\mu}{\lambda} \right)^k \sum_{r=0}^k \frac{(\lambda\tau)^r}{r!} \right], \text{ for } \mu \neq \lambda$$

$$= e^{-\lambda\tau} \sum_{r=0}^i \frac{(\lambda\tau)^r}{r!}, \text{ for } \mu = \lambda$$

The difficulty in deriving analytic results, due to the incomplete gamma functions that are constantly encountered (as above), leads us to the use of simulation to evaluate the effects of delay.

GLOSSARY OF TERMS

ADD	Architecture for Distributed Database [TOTH80]
DB	database
DBA	Database Administrator
DBMS	database management system
DDB	distributed database
DM	Data Manager
DP	Distributed Processor
FSM	finite state machine
GC	Global Coordinator
GP	Global Processor
LC	Local Coordinator
LP	Local Processor
MCW	multiple-copy write (deadlock)
pdf	probability density function
QDP	Query Decomposition Process
RAT	Resource Access Table
r-lock	read lock
RM	Reliability Monitor
R-TS	read timestamp
SI	semantic integrity
SR/EW	shared read/exclusive write (locking)
TC	tagged customer
TCP	Transaction Control Process
TM	Transaction Manager
T/O	timestamp ordering

TP transaction processing (model)

2PC two-phase commit

2PL two-phase locking

2PL/DD 2PL/deadlock detection

2PL/DP 2PL/deadlock prevention

TWR Thomas Write Rule

UVP User View Processor

w-lock write lock

W-TS write timestamp

REFERENCES

- ALSB76 Alsberg, P.A.; and Day, J.D. "A principle for resilient sharing of distributed resources," in Proceedings of the International Conference on Software Engineering, San Francisco, California, October 1976, pp. 562-570.
- BADA79 Badal, D.Z. "Concurrency control and semantic integrity enforcement in distributed databases," Infotech Report on Distributed Databases, 1979, pp. 17-27.
- BAYE80 Bayer, R.; Heller, H; and Reiser, A. "Parallelism and recovery in database systems," ACM Transactions on Database Systems 5, 2(June 1980), 139-156.
- BERN78 Bernstein, P.A.; Rothni, J.B.; Goodman, N.; and Papadimitriou, C.A. "The concurrency control mechanism of SDD-1: a system for distributed databases (the fully redundant case)," IEEE Transactions on Software Engineering SE-4, 3(May 1978), 154-168.
- BER79a Bernstein, P.A.; Shipman, D.W.; and Wong, W.S. "Formal aspects of serializability in database concurrency control," IEEE Transactions on Software Engineering SE-5, 3(May 1979), 203-216.
- BER79b Bernstein, P.A.; and Goodman, N. "Approaches to concurrency control in distributed database systems," in Proceedings AFIPS National Computer Conference, AFIPS Press, Vol. 48, 1979, 813-820.
- BER80a Bernstein, P.A.; and Goodman, N. Fundamental algorithms for concurrency control in distributed database systems, Technical Report CCA-80-05, Computer Corporation of America, Feb. 1980.
- BER80b Bernstein, P.A.; Shipman, D.W.; and Rothnie, J.B. "Concurrency control in a system for distributed databases (SDD-1)," ACM Transactions on Database Systems 5, 1(March 1980), 18-51.
- BOCH80 Bochmann, G.V.; and Sunshine, C.A. "Formal methods in communication protocol design," IEEE Transactions on Communications COM-28, 4(April 1980), 624-631.
- BRIN73 Brinch Hansen, P. Operating System Principles, Prentice-Hall, Englewood Cliffs, N.J., 1973.
- CASA79 Casanova, M.A. "The concurrency control problem for database systems," Ph.D. dissertation, Harvard University; Technical Report TR-17-79, Centre for Research in Computing Technology, 1979.
- CHEN81 Cheng, W.K. "Performance analysis of update synchronization algorithms in distributed databases," Ph.D. thesis, Dep. Com-

8

puter Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1981.

CHU75 Chu, W.W. "Optimal file allocation in a computer network," in Computer Communications Networks, F.F. Kuo (Ed.), Prentice-Hall, 1973.

CRC68 CRC Standard Mathematical Tables, S.M. Selby (Ed.), Chemical Rubber Company, Sixteenth Edition, 1968, p. 422.

DEPP76 Deppe, M.E.; and Fry, J.P. "Distributed data bases: a summary of research," Computer Networks 1 (1976), 130-138.

DISN80 Disney, R.L. "The M/G/1 queue with instantaneous Bernoulli feedback," Naval Research Logistics Quarterly 27, 4(Dec. 1980), 635-644.

ELMA81 Elmasri, R.; Devor, C.; and Rahimi, S. "Notes on DDTs: an apparatus for experimental research in distributed database management systems," ACM SIGMOD Record 11, 4(July 1981), 32-49.

ESWA76 Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; and Traiger, I.L. "The notions of consistency and predicate locks in a database system," Communications of the ACM 19, 11(Nov. 1976), 624-633.

GARC78 Garcia-Molina, H. "Performance comparisons of two update algorithms for distributed databases," in Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, August 1978, pp. 108-119.

GRAY75 Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; and Traiger, I.L. Granularity of locks and degrees of consistency in a shared data base, Report RJ1654, IBM Research Laboratory, San Jose, California, September 1975.

GRAY78 Gray, J.N. "Notes on database operating systems," in Operating Systems: An Advanced Course, Vol. 60, Lecture Notes in Computer Science, Springer-Verlag, New York, 1978, pp. 393-481.

GRAY81 Gray, J.; Homan, P.; Obermarck, R.; and Korth, H. A straw man analysis of probability of waiting and deadlock, Report RJ3066, IBM Research Laboratory, San Jose, California, February 1981.

HAMM80 Hammer, M.M.; and Shipman, D.W. "Reliability mechanisms for SDD-1: A system for distributed databases," ACM Transactions on Database Systems 5, 4(Dec. 1980), 431-466.

HOLT72 Holt, R.C. "Some deadlock properties of computer systems," ACM Computing Surveys 4, 3(Dec. 1972), 179-195.

KAM081 Kamoun, F.; Kleinrock, L.; and Muntz, R. "Queueing analysis of the ordering issue in a distributed database concurrency control mechanism," in Proceedings of the Second International

Conference on Distributed Computing Systems, IEEE, New York, 1981, pp. 13-23.

- KANE79 Kaneko, A.; Nishihara, Y.; Tsuruoka, K.; and Hattori, M. "Logical clock synchronization method for duplicated database control," in Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Alabama, October 1979, pp. 1-11.
- KLEI75 Kleinrock, L. Queueing Systems, Volume I: Theory, John Wiley & Sons, Inc., 1975.
- LAFE81 Laferriere, C. "Reliable protocols for synchronization and updating in distributed databases," Ph.D. thesis, Dep. Systems Engineering and Computer Science, Carleton University, Ottawa, Ontario, Sept., 1981.
- LAMP78 Lampert, L. "Time, clocks, and the ordering of events in a distributed system," Communications of the ACM 21, 7(July 1978), 558-565.
- LAMP76 Lampson, B.; and Sturgis, H. Crash recovery in a distributed data storage system, Xerox Palo Alto Research Centre, California, unpublished paper, 1976.
- LELA77 LeLann, G. "Distributed systems - towards a formal approach," in Proceedings IFIP 1977, IFIP Congress Series Vol. 7, North-Holland Publishing Co., Amsterdam, The Netherlands, 1977, 155-160.
- LELA78 LeLann, G. "Algorithms for distributed data-sharing systems which use tickets," in Proceedings of the Third Berkeley Conference on Distributed Data Management and Computer Networks, Berkeley, California, August 1978, pp. 259-272.
- LIND80 Lindsay, B.G. Sixth International Conference on Very Large Data Bases. Panel Discussion (Oct. 2, 1980).
- MAHM76 Mahmoud, S.A.; Riordon, J.S. "Optimal allocation of resources in distributed information networks," ACM Transactions on Database Systems 1, 1(March 1976), 66-78.
- MAHM79 Mahmoud, S.A.; Riordon, J.S.; and Toth, K.C. "Distributed database partitioning and query processing," in Proceedings of the IFIP Working Conference on Distributed Database Architecture, North-Holland Publishing Co., Amsterdam, The Netherlands, 1979, pp. 32-51.
- MENA78 Menasce, D.A.; Popek, G.J.; and Müntz, R.R. "A locking protocol for resource coordination in distributed databases," in Proceedings of the 1978 ACM SIGMOD Conference, ACM, New York, 1978, pp. 1-14.

- MENA79 Menasce, D.A.; and Muntz, R.R. "Locking and deadlock detection in distributed databases," IEEE Transactions on Software Engineering SE-5, 3(May 1979), 195-202.
- MILE80 Milenkovic, M. "Synchronization of concurrent updates in redundant distributed databases," North-Holland Publishing Co., Amsterdam, The Netherlands, 1980, pp. 48-65.
- MOHA79 Mohan, C. An analysis of the design of SDD-1: a system for distributed data bases, Report from the Software and Data Base Engineering Group, University of Texas, Austin, Texas, 16 April 1979.
- PAP065 Papoulis, A. Probability, Random Variables, and Stochastic Processes, McGraw-Hill, 1965.
- POTI80 Potier, D.; and Leblanc, Ph. "Analysis of locking policies in database management systems," Communications of the ACM 23, 10(Oct. 1980), 584-593.
- RIES79 Ries, D. "The effects of concurrency control on the performance of a distributed database management system," in Proceedings of the Fourth Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, Aug. 1979.
- ROSE78 Rosenkrantz, D.J.; Stearns, R.E.; and Lewis II, P.M. "System level concurrency control for distributed database systems," ACM Transactions on Database Systems 3, 2(June 1978), 178-198.
- ROTH77 Rothnie, J.B.; and Goodman, N. "A survey of research and development in distributed database management," in Proceedings of the Third International Conference for Very Large Databases, IEEE, 1977, pp. 48-62.
- ROTH80 Rothnie, J.B.; Bernstein, P.A.; Fox, S.; Goodman, N.; Hammer, M.; Landers, T.A.; Reeve, C.; Shipman, D.W.; and Wong, E. "Introduction to a system for distributed databases (SDD-1)," ACM Transactions on Database Systems 5, 1(March 1980), 1-17.
- SCHL78 Schlageter, G. "Process synchronization in database systems," ACM Transactions on Database Systems 3, 3(Sept. 1978), 248-271.
- SCHL79 Schlageter, G. "Deadlock-free synchronization in distributed databases," in Proceedings EURO IFIP 79, North-Holland Publishing Co., Amsterdam, The Netherlands, 1979, pp. 123-128.
- SHAP77 Shapiro, R.M.; and Millstein, R.E. "Reliability and fault recovery in distributed processing," in Oceans 77 Conference Record, Vol. 1, IEEE, New York, 1977, 31D-1 - 31D-5.
- SIMP78 SIMPAC Discrete Simulation Package, Carleton University,

Ottawa, Ontario, March 1978.

- STON79 Stonebraker, M. "Concurrency control and consistency of multiple copies of data in distributed INGRES," IEEE Transactions on Software Engineering SE-5, 3(May 1979), 188-194.
- THOM78 Thomas, R.B. "A solution to the concurrency control problem for multiple copy data bases," in Proceedings of COMPCON 78 Spring, IEEE, New York, 1978, pp. 509-515.
- TOTH80 Toth, K.C. "Distributed database architecture and query processing strategies," Ph.D. thesis, Dep. Systems Engineering and Computing Science, Carleton University, Ottawa, Ontario, May 1980.
- TRAI79 Traiger, I.L.; Gray, J.N.; Galtieri, C.A.; and Lindsay, B.G. Transactions and Consistency in distributed database systems, Report RJ2555, IBM Research Laboratory, San Jose, California, June 1979.

END

0 610 618 3

FIN