INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A methodology for designing concurrency control schemes in distributed databases

Chiu, Lin, Ph.D.
The Ohio State University, 1987

Copyright ©1987 by Chiu, Lin. All rights reserved.
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs or pages
2. Colored illustrations, paper or print
3. Photographs with dark background
4. Illustrations are poor copy
5. Pages with black marks, not original copy ✓
6. Print shows through as there is text on both sides of page
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements
9. Tightly bound copy with print lost in spine
10. Computer printout pages with indistinct print
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) seem to be missing in numbering only as text follows.
13. Two pages numbered. Text follows.
14. Curling and wrinkled pages
15. Dissertation contains pages with print at a slant, filmed as received ✓
16. Other

______________________________________________________________

______________________________________________________________

UMI
A METHODOLOGY FOR DESIGNING CONCURRENCY CONTROL
SCHEMES IN DISTRIBUTED DATABASES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Lin Chiu, B.S., M.S.

* * * * *

The Ohio State University

1987

Reading Committee:

Dr. Ming T. Liu
Dr. Mukesh Singhal
Dr. Neelam Soundararajan

Approved By

Dr. Ming T. Liu
Advisor
Department of Computer
and Information Science
© Copyright by
Lin Chiu
1987
Dedicated To My Parents

and My Husband
ACKNOWLEDGMENTS

I am very grateful to my advisor, Prof. Ming T. Liu, who guided me through these years of research, and continuously encouraged me to carry on my work. I have learned from him not only the research direction, but also the ways to organize and present ideas. I especially appreciate the way he cares about our progress by constantly motivating us, which I believe is the most important thing in research. I also wish to thank Mrs. Liu, who helped me through my problems in real life, for being such a nice person.

I would like to express my appreciation to Prof. Neelam Sondararajan and Prof. Mukesh Singhal for serving on my reading committee. In addition to their assistance in this thesis, the discussions with them were always full of inspiration and benefit.

A lot of people at OSU have affected me in many different ways. Discussions with Prof. Steve Lai were always inspiring. Prof. Yao-Nan Lien was always there whenever I ran into problems with my programs. I also wish to thank a special friend, Nien-Chen Liu, who has given me constant support all the time. His advice and comfort helped me through some difficult stages. Thanks also go to all members in the Reliable Distributed Systems group, especially Ming-Jye Sheu; together we worked out many interesting ideas.

Finally, I wish to thank my husband, Hong-Chei, for being so patient and understanding. Together we have shared both the tears and joy of life. I also wish to thank my parents for giving me the opportunity to pursue my goal. There is nothing comparable to their love and support for me.
VITA

August 20, 1959  Born, Taipei, Taiwan, Republic of China

1981  B.S., Dept. of Computer and Information Engineering, The National Taiwan University, Taipei, Taiwan, Republic of China

Summer 1982  System Programmer, The Wei-Shih Company, Taipei, Taiwan

1983  M.S., Dept. of Computer and Information Science, The Ohio State University, Columbus, Ohio

Summer 1983  Graduate Research Associate, Dept. of Accounting, The Ohio State University, Columbus, Ohio

1981—present  Graduate Teaching Associate, Dept. of Computer and Information Science, The Ohio State University, Columbus, Ohio
PUBLICATIONS

  Co-author: Ming T. Liu

  Co-author: Ming T. Liu

  Co-author: Ming T. Liu

FIELDS OF STUDY


• Programming Languages: Concurrent (Distributed) Programming Languages, Formal Specification and Analysis Techniques, Program Validation and Verification.

• Computer Architectures: Computer Organizations, Network Architectures.
Table of Contents

Dedication ......................................................................................................... ii
Acknowledgments ........................................................................................... iii
Vita ............................................................................................................... iv
Table of Contents ......................................................................................... vi
List of Tables ............................................................................................ lx
List of Figures .............................................................................................. x

1. Introduction ................................................................................................. 1
   1.1 Motivations ........................................................................................ 1
   1.2 Research Objectives and Contributions .................................... 5
   1.3 Organization of the Thesis ........................................................... 8

2. Preliminaries ............................................................................................... 11
   2.1 Concurrency Control Problem .................................................... 11
   2.2 Concurrency Control Schemes .................................................... 16
      2.2.1 Two-phase Locking ............................................................ 16
      2.2.2 Timestamping ........................................................................ 18
      2.2.3 Commit-time Validation .................................................... 19
      2.2.4 Summary ................................................................................ 20
   2.3 An Object-oriented Design Model ............................................ 21
   2.4 A CSP-based Specification ............................................................ 22

3. High Level Specification Method ......................................................... 25
   3.1 Introduction ........................................................................................ 25
   3.2 The Model ........................................................................................ 31
   3.3 High Level Specification ............................................................... 38
      3.3.1 The Concurrency Control Datablock .................................... 38
      3.3.2 The Concurrency ControlInvariant ..................................... 43
      3.3.3 Declaration of Message Types ........................................... 43
      3.3.4 The Processing .................................................................... 44
      3.3.5 Related Issues ...................................................................... 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conclusions</td>
<td>140</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>140</td>
</tr>
<tr>
<td>6.2 Concluding Remarks and Future Work</td>
<td>143</td>
</tr>
<tr>
<td>Appendix A. Simulation Program for the Variable Timestamping Scheme</td>
<td>146</td>
</tr>
<tr>
<td>Appendix B. Simulation Program for the Timestamping with Locks Scheme</td>
<td>181</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>213</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1.</td>
<td>Example of an AccessTable</td>
<td>40</td>
</tr>
<tr>
<td>Table 4.1.</td>
<td>Precedence Relationship by X and Y</td>
<td>70</td>
</tr>
<tr>
<td>Table 4.2.</td>
<td>Precedence Relationship by X and Y</td>
<td>74</td>
</tr>
<tr>
<td>Table 5.1.</td>
<td>Variable Timestamping(VT)</td>
<td>121</td>
</tr>
<tr>
<td>Table 5.2.</td>
<td>Timestamping with Locks(TSLK)</td>
<td>121</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1. Example of Two Interfering Transactions ............... 13
Figure 3.1. Organization of A Distributed Database System .......... 31
Figure 3.2. Abstract View of A Distributed Database System ......... 33
Figure 3.3. Structure of A Transaction Handler ......................... 35
Figure 3.4. Structure of A Data Object .................................. 37
Figure 3.5. Specification for Two-phase Locking Scheme ............... 50
Figure 3.6. Specification of Timestamping ................................. 53
Figure 3.7. Specification of Commit-time Validation ..................... 56
Figure 4.1. Precedence Relationship Represented by Channels ....... 71
Figure 4.2. Precedence Relationship after T2 is Aborted ............... 71
Figure 4.3. Precedence Relationship Represented by Channels ....... 75
Figure 4.4. Message Arrivals ............................................... 76
Figure 4.5. Message Arrivals when T2 is Aborted ......................... 77
Figure 4.6. Cycles in Precedence Relationships ......................... 78
Figure 4.7. Specification of the Transaction Handler Object .......... 87
Figure 4.8. Specification of the Data Object .............................. 91
Figure 5.1. Communication between Objects ............................... 100
Figure 5.2. Example I .......................................................... 106
Figure 5.3. Example II .......................................................... 109
Figure 5.4. Example III .......................................................... 113
Figure 5.5. System Throughput vs REQPER With Message Delay = 0.1 123
Figure 5.6. System Throughput vs REQPER With Message Delay = 0.05 124
Figure 5.7. System Throughput vs REQPER With Message Delay = 0.5 125
Figure 5.8. Average Response Time vs REQPER With Message Delay = 0.1 127
Figure 5.9. Average Response Time vs REQPER under VT With Various Message Delay 128
| Figure 5.10. | Response Time vs REQPER under TSLK With Various Message Delay | 129 |
| Figure 5.11. | Average Response Time vs Message Delay | 130 |
| Figure 5.12. | Specification of the Transaction Handler Object | 134 |
| Figure 5.13. | Specification of the Data Object | 136 |
CHAPTER I
Introduction

1.1 Motivations

Because of the rapid growth in the volume of information needed in both research and applications, database technology has been one of the most rapidly growing areas in computer and information science. In the early days, a database system was usually no more than a complicated file management system; however, it has now evolved into a system more sophisticated than an operating system, involving newer technology, such as artificial intelligence. To maintain the enormous amount of information and provide services to users in an efficient way, data must be integrated and shared. 'Integrated' means the database system must unify several otherwise distinct data files, with redundancy within files removed; and 'shared' means the database system must provide services to many users.

The performance of a database system will be increased if data can be shared among users: that is, several users can actually access the
database -- possibly even the same piece of data -- at the same time. However, this kind of sharing leads to possible interference among concurrent accesses to the same part of the database at the same time. To resolve the problem without eliminating the 'sharing', the behavior of concurrent accesses and possible interference must be studied.

Research in 'concurrency control' in database systems not only studies the types of interference between concurrent accesses, but also resolves the kind of interference that leads to incorrectness of the database. In general, the interference between concurrent accesses can be categorized as **read-write**, **write-read** and **write-write** dependency. The types of interference existing between concurrent accesses are important to the method for resolving them because the three kinds of dependency mentioned above may cause **unreliable-read**, **dirty-read**, and **lost-update** problems respectively. The simple method used in current operating systems for mutual exclusion is no longer feasible due to the fact that users concurrently access the shared data in an interleaved way. More complicated schemes are needed not only to prevent interference among concurrent accesses on one instance but also to guarantee that the same kind of policy enforcement is consistent on other similar instances.

Different approaches with varying prospects, have been proposed to resolve interference among concurrent accesses. An optimistic method
might be too optimistic on some occasions while a pessimistic method might be overconservative on other occasions. Therefore, a lot of research has been devoted to evaluating existing concurrency control methods using either analytical models or simulations. Their common goal is to find a better scheme.

Despite the work that has been done in developing concurrency control schemes and evaluating them, much work is still needed. For one thing, the semantic knowledge of transactions must be obtained to maximize the sharing among users without sacrificing system consistency. Distributed database systems call for newer schemes, since concurrency control schemes suitable for centralized database systems cannot maintain the same level of performance when they are extended to distributed database systems. This is because the separation of the database and the need for communication mechanisms implicit in a distributed database greatly affect the centralized concurrency control schemes.

In a distributed database system, the global state of the system is the collection of all local states. Due to communication delays, finding the global state becomes non-trivial and uneconomical. Without a global state, many of the centralized schemes have to rely on ad hoc routines in order to cope with the distributed environment. Therefore, understanding the nature of a distributed computing environment is crucial
the design of a distributed concurrency control scheme. Although distributed systems have come into practice for a while now, we believe that the power of parallel processing in many areas has not been fully utilized.

Concurrency control schemes can be regarded as a kind of protocol governing the interaction among concurrent transactions. Therefore, they should be well specified. Concurrency control schemes should also be able to be validated and verified. Unfortunately, there is lack of a specification model for specifying these schemes and for verification as well as validation. These unsolved problems have stimulated our interest in working on the concurrency control problem, especially in distributed database systems. Therefore, an object-oriented model that suits our needs is presented. Based on the model, the specification is constructed. The concept of distributed control has been incorporated into the development of new concurrency control schemes. The proposed schemes are described based on the object-oriented model and specified using the specification to eliminate any ambiguities.
1.2 Research Objectives and Contributions

The objectives of this research are to provide a high level specification for describing concurrency control schemes and to incorporate the idea of distributed control into the design of concurrency control schemes in a distributed database environment.

Since distributed database systems have different prospects from centralized database systems, concurrency control schemes for distributed database systems should be based on the concept of distributed control instead of centralized control. From the study of many well known concurrency control schemes, it seems clear that the idea of distributed control has not been used to design most of these schemes. This is true for both pessimistic and optimistic schemes. In a pessimistic scheme, the ability of data items and transactions to negotiate with each other has often been ignored. Hence, the resulting scheme is unavoidably conservative and less flexible. In an optimistic scheme, the computing power of a distributed system has not been fully used; therefore, the maximum performance of the scheme cannot be obtained.

We first present an object-oriented model as the foundation for our work. The object-oriented model has the advantages of data abstraction, which encapsulates the data and the operations performed on them in an object. One result of this is that the semantic meaning of an object can
be clearly defined and separated from the interface to these objects. Our model extends this idea by encapsulating the concurrency control data within the transaction process. As a result, the distributed concurrency control can be better described. Both transaction processes and data objects are objects; therefore, their roles in the concurrency control can be described more uniformly.

In order to describe concurrency control schemes in a formal way, a high level specification of concurrency control schemes is proposed which incorporates software engineering into the design of concurrency control schemes. Basing the specification on our object-oriented design model, it is easy for us to describe the behavior of transaction handlers, data managers and so on. Since transaction handlers, data managers, and processes are all regarded as objects, the specification of a concurrency control scheme can be constructed as the integration of the specifications of various objects and the cooperation among these objects.

Since 'objects' originated from associating operations with data, the specification of an object in a concurrency control scheme includes the definition of the data needed for exercising concurrency control and the internal operations performed on these data. For each object, concurrency control invariants are clearly defined to describe the behavior of that object as seen by the rest of the system. We also show the usage
of the communication mechanisms, included in the high level specification, for describing interactions between objects. The invariants of individual objects and the communications between them indicate how the concurrency control is distributed and how these objects cooperate in maintaining the system consistency. Our specification method is at high level thus providing a high degree abstraction; moreover, it has the capability of verifying and validating the specified scheme.

We then incorporate the idea of distributed control into the design of concurrency control schemes. Due to the different characteristics of database systems, which are attributed by transaction patterns and data management constraints, the concurrency control scheme suitable for various systems might be very different. Therefore, two general categories of concurrency control schemes, i.e., pessimistic and optimistic, have been studied. Pessimistic schemes rely on rules to prevent any inconsistency from occurring while optimistic schemes rely on methods to resolve inconsistency caused by interference.

To demonstrate how to incorporate distributed control into both types of schemes and show the improvement in the performance of the resulting schemes, we develop two schemes. In the case of pessimistic concurrency control schemes, we propose a scheme based on negotiation among the transactions and data objects, resulting in a great improve-
ment in system throughput. In the case of optimistic concurrency control schemes, we propose a scheme based on fully distributing the concurrency control to both transaction and data objects. As a result, the entire scheme becomes simple to describe, and the system performance has increased due to the distribution of control.

1.3 Organization of the Thesis

This chapter has explained the motivation of the work, research objectives and major contributions. The rest of the dissertation is organized as follows.

Chapter 2 introduces the preliminaries in concurrency control. Many important works have been included not only because they reflect the current trend in this area, but also because our research is based on many of them. Therefore, terminology and the notion of consistency are explained in this chapter in order to establish clarity for the following chapters. The main concept of object-oriented models and a CSP-based specification method are also introduced.

Chapter 3 introduces the high level specification for describing concurrency control schemes. The reason for such work is detailed in this chapter, followed by the presentation of the specification. Examples of
well known concurrency control schemes have been chosen to provide a
good understanding of the specification and to demonstrate both the
use and the power of the high level specification.

Chapter 4 discusses the optimistic concurrency control scheme
developed as one of the results of incorporating distributed control into
designing better schemes. The scheme proposed integrates the use of
communication networks and abstract data types. This chapter explains
how the model is applied, the assumptions made, the concurrency control
scheme, and state restoration. The proposed scheme is described first in
the usual informal way and then specified using the high level specification
for more clarity and to allow comparison between the informal
description and the formal specification.

Chapter 5 presents the pessimistic concurrency control scheme as
another result of incorporating distributed control into various parts of a
system. The interaction among objects is demonstrated in our object-
oriented model. The interaction among objects results from negotiation
when potential concurrency control problems occur. Simulation work has
been included to show the expected performance, under various system
conditions, as compared to a similar scheme without negotiations among
objects. The results obtained from the simulation are shown and the
complete simulation program for the two compared schemes are included
in Appendices A and B. The formal specification of this scheme is also included towards the end of this chapter.

Finally, Chapter 6 concludes the study with a brief review of our work and some final remarks. We have set a new direction for future work in the subject of concurrency control, especially in proving formal specifications to enable verification and validation.

Together we have presented a methodology for designing concurrency control schemes in distributed databases. Our goal is to set up the design philosophy for future concurrency control schemes. Since new technologies come out very rapidly, the need for concurrency control schemes may be changing. Therefore, our methodology is based on the design principles and a formal specification for describing concurrency control schemes.
CHAPTER II
Preliminaries

An overview of the concurrency control problem and a discussion of how the problem is resolved by concurrency control schemes are given in this chapter. Various concurrency control schemes are examined. We describe the problem associated with those schemes by looking into their general characteristics. We also discuss the object-oriented design models and introduce our object based model.

2.1 Concurrency Control Problem

Since a database allows sharing among users to be one of its major functions and provides fast response time to users, more than one user may access the database during the same period of time. In a distributed database system, executions of queries by different users may actually take place simultaneously. We define a database system to be a collection of data objects which are shared by many users. Similarly, a distributed database system can be defined as one which consists of a
finite set of transactions, communication links, and physically scattered
data objects which logically constitute the database. Although data may
be replicated, logically there should be only one consistent view provided
to the users.

Distributed database systems are especially considered in this thesis
because they are more general than nondistributed database systems and
provide more reliability as well as availability. The term 'distributed' has
to be clarified to avoid any confusion. A truly distributed system as
defined by Enslow has the characteristic of cooperative autonomy, which
means that the same degree of cooperation must exist between com­
ponents, all components follow a "master plan", and there is no hierar­
chy of control within the system [20]. This is very essential to a dis­
tributed system, because components can no longer be tightly-coupled
due to their physical locations and for the purpose of reliability. There­
fore, the above argument has excluded database systems that are physi­
cally distributed, but centrally controlled.

A user interacts with a database by executing a transaction, which
consists of a sequence of read, write, and other operations on data ob­
jects. The interaction between concurrent users, accessing the same por­
tion of the database can be best explained by using the following ex­
ample. Figure 2.1 depicts the actions of two transactions running con­
Currently, Transaction T1 is transferring 100 dollars from the savings account to the checking account while transaction T2 is producing a report of the total balance of these two accounts. Without any synchronization between T1 and T2, it would be possible for T2 to read the balances of both accounts after T1 had subtracted one hundred dollars from the savings but before T1 added one hundred dollars into the checking. Therefore, the result of T2 would be incorrect even if transaction T2 is semantically correct by itself.

\[
\begin{align*}
T1 & \quad \text{read(savings)} \\
& \quad \text{write(savings)} \\
& \quad \text{read(checking)} \\
& \quad \text{write(checking)} \\
\end{align*}
\]

\[
\begin{align*}
\text{Results: savings} &= 400.00 \\
\text{checking} &= 150.00
\end{align*}
\]

\[
\begin{align*}
T2 & \quad \text{read(checking)} \\
& \quad \text{read(savings)} \\
& \quad \text{write(total)} \\
\end{align*}
\]

\[
\begin{align*}
\text{total} &= 450
\end{align*}
\]

\textbf{Figure 2.1. Example of Two Interfering Transactions}

The interaction between co-existing transactions has been well studied in the literature, leading to the development of a number of concurrency control schemes. In [25], three forms of inconsistencies have been defined in order to describe incorrect user interaction. They are
lost-update, dirty read, and unrepeatable read. Lost-update occurs when two transactions read and update the same object in an interleaved fashion, hence one update is lost. Dirty read occurs when one transaction reads an uncommitted update which is later rolled back; consequently the data read should never have existed. Unrepeatable read is the opposite of dirty read and occurs when one transaction reads the data before it is updated by another concurrent transaction. This means the state of the database system could change in the middle of one transaction execution, which obviously may lead to inconsistency. Similar observations have also been made in [51], where relationships used for detecting inconsistency were identified. They are the write-write, write-read, and read-write dependencies. The term conflict has been used to capture the meaning of different dependencies; two transactions conflict, if one writes a data object which is also written or read by another transaction. The inconsistency resulting from two conflicting transactions must be resolved by exercising concurrency control.

Concurrency control imposes certain semantic relationships, known as consistency assertions, which guarantee the database to be consistent when these assertions are satisfied. In [2], Badal has argued serializability to be an appealing correctness criterion for concurrency control for the following reasons. A serial execution of transactions preserves the database semantic integrity, if each single transaction preserves the
database semantic integrity. If an interleaved execution of the trans-
actions produces the same effect as the serial execution of those same
transactions, then the execution is called *serializable* or serially
reproducible. Obviously, *serializable* execution preserves database semantic
integrity. The write-write, write-read, and read-write dependencies, or
simply called *precedence relationships*, can now be used to define a suf-
ficient and necessary condition for the serializability of a given sequence
of executed transactions [4, 36].

The above arguments are based on the assumption that the seman-
tic knowledge of each transaction cannot be obtained. Using semantic or
application knowledge in transaction execution will increase parallelism
[23, 24]. In [24], transactions are divided into *steps*. Each step is a col-
lection of conventional database operations. Two transactions are
*compatible*, if it is possible to interleave their steps in any fashion with-
out violating database consistency. The users of the database define
which transactions are compatible by classifying them into semantic
types, and for each type, the compatible types are given. The perfor-
mance evaluation of using semantic knowledge is also given in [24].
2.2 Concurrency Control Schemes

In this section, we give an overview of the concurrency control schemes that have been proposed and show their disadvantages, setting the stage for the schemes we develop. There are three basic strategies for ensuring the consistency of a database. They are two-phase locking, timestamping, and commit-time validation.

2.2.1 Two-phase Locking

Concurrency control schemes based on this strategy [21, 39, 6] require a transaction to obtain a lock on the data object the transaction wants to access, and the locks acquired will not be released until all the locks needed for the transaction are obtained. Therefore, it consists of an expanding phase during which locks are acquired and a shrinking phase during which locks are released. Since a transaction cannot finish unless it acquires all the locks it needs, the lockpoint [16] (the point at which all the locks are acquired) can be used to determine the serializable order among transactions. There are a number of drawbacks in applying locking schemes to a distributed system. In particular,

- Strong serializability is enforced when not required by the application and thus may decrease the degree of concurrency.
• Deadlock detection and resolution methods must accompany the locking scheme and, due to the fact that the global state of a distributed system cannot be easily determined, deadlock detection could be very costly.

To remedy the first drawback, a number of non-two-phase protocols that ensure serializability have been proposed; however, most of them rely on a priori information as to how the database entities are organized [44]. Although non-two-phase locking allows more concurrency, it also suffers from cascading rollback. Therefore, the tradeoff in selecting a locking protocol is between guarding against cascading rollbacks versus increasing parallelism.

Deadlock detection schemes used in centralized database systems are expensive and impractical when they are applied to distributed database systems since it is difficult for any single site to maintain correct information about the global status of the system. Hence, several distributed deadlock detection schemes have been established to cope with such difficulties [18, 47]. However, most schemes proposed require each site to maintain more or less global information for deadlock detection by exchanging complex messages. And only a few have considered deadlock resolution according to the findings in [47]. An elegant distributed deadlock detection and resolution algorithm is also proposed in [47], which requires each site to resolve local deadlocks internally and
allows simple messages to be exchanged among sites in order to detect and resolve global deadlocks.

2.2.2 Timestamping

Schemes such as the basic timestamping [31, 6], conservative timestamping [6, 15], and multiversion timestamping [38] are all based on this strategy. A general timestamping scheme requires each transaction to be assigned a unique timestamp. A transaction is not allowed to read a data object, if it has been updated by a transaction with a higher timestamp, and a transaction is not allowed to update a data object if it has been either read or updated by a transaction with a higher timestamp. In other words, transactions are required to execute in the order of their timestamps. This also implies that a timestamping scheme enforces only one serializable order of transaction execution, which is determined by their timestamp ordering. The major drawback of this scheme is that concurrency is sacrificed in order to avoid possible inconsistency.
2.2.3 Commit-time Validation

Commit-time validation schemes [30, 9, 14] allow a transaction to execute freely, assuming that there is no conflict. But at the end of the transaction execution, the data objects read and updated by the transaction must be validated to ensure that there has been no conflict (i.e., the transaction has viewed the database in a consistent state during its execution). If so, the transaction commits. If not, the transaction is aborted and restarted. This also implies three phases of a transaction execution: the read phase, the validation phase, and the write phase. Updates produced by a transaction will be applied to the database only if the transaction has been validated against other committed transactions.

The main argument in favor of such schemes are a) a higher degree of concurrency is provided and therefore shorter delays and b) the overhead associated with nonconflicting transaction synchronization is low. However, the overhead of conflicting transaction synchronization can be high. Another drawback is that the validation requires some kind of synchronization among various sites, which could be impractical in a real system. Modifications made to the original scheme in [30] have been proposed in [9, 14], which allow several transactions to be validated at the same time and allow transactions to be committed in a serializable order, not necessarily in the same order of their entering the validation phase.
2.2.4 Summary

All three strategies offer solutions to the concurrency control problem; however, they all have various drawbacks. To compare various schemes, a lot of work has been done to evaluate their performance [42, 13]. However, due to the different characteristics of various concurrency control strategies and the different assumptions made by each scheme, it is often hard to find a fair analysis that can be used to choose a good concurrency control scheme for a particular application.

Since many concurrency control schemes incorporate more than one strategy [6, 8], it is useful to describe a scheme to be either optimistic or pessimistic. Optimistic schemes detect and resolve inconsistency based on the assumption that inconsistency is unlikely to occur while pessimistic schemes try to prevent inconsistency from occurring. As mentioned in this thesis, there is lack of a uniform representation method for concurrency control schemes; therefore, one goal of this thesis is to construct a high level specification method that can be easily used for describing concurrency control schemes without ambiguity. Consequently, the imprecision caused by using general terms such as two-phase locking to characterize any scheme, can be eliminated.
2.3 An Object-oriented Design Model

The concept of objects, the notion of data abstraction by merging data and processing [22], originates from programming languages and has been extended to operating systems and database systems. Each object consists of the data and the operations to manipulate the data; therefore, invariants can be defined regarding the state of each object. An object-oriented approach is comprised of objects accessed or updated by users through transactions. Since interaction between transactions and objects can be better defined, object-oriented designs offer an attractive approach to constructing reliable systems [50, 32].

Using an object-oriented model in designing concurrency control schemes offers other advantages. First, the communication mechanisms can be better defined between objects. This is because operations are encapsulated within objects, hence, it is possible to define the interface control more easily. Second, the correctness of an object can be easily defined since most errors can be confined within the object. This also implies that each object can be managed as a recoverable unit in designing resilient database systems.

We have chosen an object-oriented design model for its attractive features. However, we have further extended the model to include transactions as objects also. For the same reason of merging data with opera-
tions, we have derived a different kind of object which merges processing with data. Processes that handle transaction executions are now associated with concurrency control data, and together they constitute objects. One advantage we obtain is uniformity. Since both the transaction execution and the accessed data are represented as objects, the design of concurrency control schemes is more standardized. The other advantage is to distribute the concurrency control to various objects by describing the distribution of control among objects based on the object-oriented model.

2.4 A CSP-based Specification

CSP was proposed by Hoare [26] for describing concurrent processing in distributed systems. It has combined nondeterminism and guarded commands [17, 27, 5, 11] to specify possible alternatives without built-in preferences. This gives the programmer the freedom of designing programs without being restricted to the built-in preferences that come with the language. This feature is especially attractive to us because we are not concerned with such preferences in a high level specification.

CSP was originally designed for synchronous communication between concurrent processes even though how asynchronous communication can be described using unbounded buffers was also included in the
original article by Hoare. Synchronous communication does offer several advantages. The most important one is that the programmer is very clear about the states of two communicating processes. However, as described in the next chapter, we prefer asynchronous communication for the following reasons. First, the actual system being modeled relies on asynchronous communication. In other words, message delays have to be considered. Second, synchronous communication requires each process to be aware of the possible communication that might take place in order to avoid communication deadlocks. However, using asynchronous communication allows concurrent processes to select both synchronized and asynchronous executions. Synchronized executions can be carried out by requiring processes to exchange certain messages in order to continue while asynchronously executions can be carried out by allowing processes to execute without exchanging messages.

CSP was designed for a static system where the number of processes are fixed. However, in our model, transaction handlers are created and terminated dynamically. Therefore, the communication mechanisms, i.e., the input and output commands, had to be modified. The original input and output commands require the other communicating process to be identified. Due to the dynamic creation and termination of processes as mentioned above, the other communicating process cannot be easily identified. Therefore, the communication mechanisms are
modified such that only the types of messages exchanged are identified. This modification can be implemented using unbounded buffers in the original CSP; one buffer for each different type of messages.

Although objects considered in our model do not contain sub-objects, such nesting can be well described based on the hierarchy of processes of CSP. In CSP, a process can have several subprocesses, and communication between these subprocesses are internal to the process. This can be very useful when considering complicated objects that contain internal subobjects. Therefore, the structure of our specification need not be modified. The modifications made to CSP for the high level specification are described in Section 3.3.4.
CHAPTER III
High Level Specification Method

3.1 Introduction

Concurrency control is one of the major issues in database systems. For the purpose of concurrency control, many schemes, based on different strategies, have been proposed. Unfortunately there is still lack of a general model for describing these schemes. Hence, schemes cannot be uniformly presented, which makes it hard to understand them and to prove their correctness. Another issue that has gradually attracted more attention is that the concurrency control problem should be integrated with related problems, such as recovery control and file management. This will resolve the concurrency control problem more effectively by considering related issues at the same time. This chapter discusses a representation model suitable for high level specification of concurrency control schemes. Concurrency control schemes are presented in a high level fashion without losing their formality. This allows easy expansion to allow for formal specification that can be used for verification and
validation on one hand, and to integrate other issues related to the problem on the other hand.

Many concurrency control schemes have been proposed in the last decade. The three strategies most commonly used are two-phase locking \([6, 21, 39]\), timestamping \([6, 31, 48, 49]\), and commit-time validation \([30]\). We briefly give an overview of these strategies here, since they are used as examples in latter sections. The two-phase locking strategy requires a transaction to obtain a lock on the data object before the access, and locks will not be released until all needed locks have been acquired. The timestamping strategy assigns a unique timestamp \([31]\) to each transaction, and accesses from different transactions to one data object are sequenced according to their timestamps. Since transactions do not access data objects in the order of their timestamps, the arrival of an out of sequence request will cause the requesting transaction to abort. The commit-time validation strategy defers all updates of a transaction till its commit time; only if no other committed transactions invalidate the data read by the transaction will all its updates be applied to the database.

Most concurrency control schemes proposed are based on these three basic strategies with some modifications; for example, in the commit-time validation scheme, transactions are validated not according
to their serializable order as in [9]. Some others integrated more than one scheme, for example, to allow the system to adapt itself according to the system status [41], etc. This has resulted in many schemes that are much more complicated than the schemes from which the original idea came, and each seems to have some advantages over the others in some way.

These schemes have been compared by analyzing the performance of each. However, assumptions made by different schemes are not necessarily the same. Thus, it is often hard to give a fair comparison between two schemes. For example, if one scheme produces fewer aborts than another scheme at the cost of increased communication overhead, the difference in the performance of the two schemes depends on whether the amount of cpu time saved from fewer aborts can justify the communication overhead or not.

One possible solution would be to set up some kind of testbed in which all costs are fixed for comparison [28, 29]. However, we feel that it is more desirable to describe these schemes by constructing a uniform model, and to specify these schemes in a uniform way. This will make it easier to compare the schemes.
Without a uniform model, concurrency control schemes have been described in many different ways. Some have been explained in plain words, some written in high level programming languages, and some have been explained using graphs to show their logic. This makes it hard to understand new schemes, let alone to prove their correctness. Work similar to ours has been proposed in [12] in which an abstract model was suggested mainly for a centralized database system. Although it may be extended to a distributed environment by extending predicates from a centralized system to include all sites in a distributed system, the interaction among components in a distributed system cannot be expressed. We believe that both the types of messages used by a concurrency control scheme and the way they are exchanged play an important role in a distributed system. Formalizing the description of concurrency control schemes also eliminates the ambiguity of using terms such as deadlock detection, deadlock prevention, optimism, and pessimism, which are not specific enough for identifying the characteristics of one particular scheme.

Another goal of presenting concurrency control schemes in high level is to include other related issues without actually describing those issues. For example, the recovery scheme [1] or the buffer management [37] that might affect the concurrency control schemes. To incorporate other schemes relating to the concurrency control scheme while keeping
the description as simple as possible, we suggest a hierarchical fashion to describe them. The specification introduced here serves as the top layer in a hierarchy of specifications for concurrency control schemes; each lower layer provides more details in terms of implementation than an upper layer. For example, in our specification, a logging scheme might be specified as the recovery method without the details of how recovery will actually be performed. Because of the hierarchy, implementation details need not be concerned here. Other examples include how messages are delivered and how rollback is performed. Therefore, our specification method simply includes the invariants regarding consistency kept by individual objects in the system, the way an object reacts to requests while maintaining the invariants true, and other issues that will help explaining the behavior of that object. Subschemes are elaborated in subprograms in our specification, which complies with the top-down design concept in high level programming languages.

To provide a formal way to specify concurrency control schemes, all data representations must be unified. Since concurrency control schemes are based on the relationship of conflicting transactions in one way or another, we have found that a general table containing the history of transaction accesses and a set of general functions to infer dependency relationship from the table are sufficient for describing any relationship between two conflicting transactions. We shall refer to this table as the AccessTable.
We object to the use of systemwide predicates, such as *serializable*, in the specification because during some period of time the system could be in an inconsistent state which would be resolved later without being visible to outside users. Instead, we use local invariants pertaining to each individual component in our specification. This not only explains the local behavior clearly but also eases the task of proving the correctness of the scheme.

The high level specification uses CSP [26], with modification, as the fundamental language for describing schemes because CSP is suitable for describing concurrent processes in distributed environments based on message passing. Hence the need for designing a new specification language is eliminated. Another reason is that CSP can be easily customized to user needs. To provide the capability of describing concurrency control schemes in a uniform way, the format of the language should be kept minimal, which was also the intention of CSP.

This chapter is organized as follows, Section 3.2 describes the object-oriented model on which high level specification is based. Section 3.3 describes the high level specification. Section 3.4 contains examples expressed in the high level specification. Section 3.5 concludes this chapter.
3.2 The Model

Because the architecture of a database system varies from one system to another, we have chosen an abstract view of a distributed database system. Figure 3.1 depicts one possible architecture of a distributed database system consisting of 3 sites. Each site contains a transaction manager (TM) and a data manager (DM). There is one...
process, called transaction handler (TH), initiated by the TM for each transaction running at that site. The TH terminates at the time of the termination of its transaction. Communication is involved whenever there is a remote access (accessing the data residing at a remote site) or there is a need for synchronization among replicated copies. In a distributed database system, a running transaction does not necessarily reside on the same machine with the data object it wishes to access; therefore, a lot of communication has to take place for both actual data accesses and synchronization with other sites. Being a distributed system, the communication network is there by definition.

Our model represents an abstract view of the actual database organization as depicted by Figure 3.2. In Figure 3.2, THs and DMs are independent objects such that no direct accesses to the content of one object can be made by other objects and there is no master-slave relationships between these objects. The TMs are not included in the figure because their major role is to initiate and terminate THs, a role which is of little concern here. Our model is object-oriented not only because object-oriented models are as favored as a rule, but also because it is capable of describing the behavior of a distributed system easily.

A transaction and the data object it accesses could be viewed as two objects that have to cooperate through communication in order to
Figure 3.2. Abstract View of A Distributed Database System

finish executing the transaction; neither of them knows the global status of the system, and each can act on its local information. Hence, the concurrency control relies on the cooperation of these objects and each object follows some rule, and together they keep the system in a consistent state. Next, we will describe in detail the objects mentioned above.

In our model, a distributed database system is represented by a finite set of transaction handlers and physically scattered data objects which logically constitute the database. There might be other kinds of objects such as a process that does the deadlock detection, but they are of little concern here.
Each transaction handler acts as a guardian of its transaction. A transaction consists of a series of operations requested by the user such as read and write and computations. The transaction handler issues transaction requests to the appropriate data objects and commits its transaction if the transaction runs to completion successfully or aborts the transaction when it fails. The transaction handler plays an important role in concurrency control, for example, to coordinate the commit protocol.

Since in our model, each transaction handler handles only one transaction, a transaction handler ceases to exist when its transaction either commits or aborts. Figure 3.3 shows the structure of a transaction handler. A transaction handler receives and sends messages in order to communicate with data objects to be accessed or to coordinate commit or abort operation. A transaction can be in five states: *active*, *waiting*, *waiting to commit*, *committed*, and *aborted*, as controlled by the transaction handler. A transaction handler terminates when its transaction either commits or aborts. In the latter case, another transaction handler will be created and be responsible for restarting the transaction.

Normally after a transaction successfully accesses a data object, it allows other transactions to access the same data. However, in the case of a transaction locking the data exclusively for a long period, other
Figure 3.3. Structure of A Transaction Handler

waiting transactions are forced to wait in the queue. In this case, the data manager can regain the lock by aborting the transaction if a deadlock or other similar condition is detected.
The data object consists of a data manager in addition to the data item. All requests must be served by the data manager; direct accesses by transactions are not allowed. One of the two important tasks performed by the data manager is to obtain a serializable order of transaction operations by applying some concurrency control rule, which is part of the concurrency control for the entire system. The other one is to maintain the data in a consistent state against replicated copies, rollback, and recovery. The granularity of data items determines the number of data objects in a database system.

Figure 3.4 shows the structure of a data object. When a transaction requests a data item, a request will be issued by the transaction handler and will be placed in the waiting queue of that data object by the data manager. It is totally up to the data manager, based on the concurrency control rules imposed, which request to serve and which to reject. Due to different concurrency control schemes, on one hand the data manager might do nothing to a request except for reporting its wait-for status for the purpose of deadlock detection, or on the other hand, it might be involved in checking for any possible violation of the consistency constraints if the request is granted. Actual data accesses are performed by the file manager.
Figure 3.4. Structure of A Data Object
3.3 High Level Specification

This section explains various parts of the high level specification. The specification of an object in our model may contain the following parts: the concurrency control datablock, the concurrency control invariants, message types, processing, and related issues. We describe the functionality of these five different parts in the following.

3.3.1 The Concurrency Control Datablock

The concurrency control datablock part in the specification describes the information needed and maintained by an object to exercise concurrency control. Different schemes might require different kinds of information. For example, in a locking scheme, it is very important to know which transaction is waiting for the lock, the so called wait-for relationship. Hence the database should contain all the wait-for relationships to detect and resolve deadlocks. However, in a timestamping scheme, the timestamps of the transactions that access the data items replace the need for the wait-for relationship since deadlock is impossible.

As mentioned in Section 3.1, a general table scheme with a set of predefined functions is used to infer any relationship between two conflicting transactions. The table, called AccessTable, contains the history of transaction accesses to the data object, including the operations
that are currently active and pending. In other words, the history of data accesses made and possible future data accesses are all treated the same way. It is up to the concurrency control schemes to decide what information is needed for each data access, executed or not. Therefore, a data object selects the kind of information to enter into the table. This is done by declaring the attributes of each table entry in the concurrency control datablock part. The most commonly used attributes are transaction_ids, operation type, and the status of the operation. The operation type could be either read or write, and there are four possible statuses of one data access: committed, finished (but not yet committed), active, and waiting.

Table 3.1 shows an example of a typical table for a locking scheme. In this table, T4 and T5 are the two transactions currently holding read locks on the data object, and T6 is waiting for a write lock.

The set of functions to be used with the AccessTable can be categorized into three kinds:

1) Functions used to derive dependency relationships among conflicting transactions:

- \( \text{prev}(\text{parameter\_list1};\text{parameter\_list2}) \) first finds a unique entry in the AccessTable according to \( \text{parameter\_list1} \) and then
Table 3.1. Example of an AccessTable

<table>
<thead>
<tr>
<th>tranid</th>
<th>operation</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>read</td>
<td>committed</td>
</tr>
<tr>
<td>T2</td>
<td>write</td>
<td>committed</td>
</tr>
<tr>
<td>T3</td>
<td>read</td>
<td>finished</td>
</tr>
<tr>
<td>T4</td>
<td>read</td>
<td>active</td>
</tr>
<tr>
<td>T5</td>
<td>read</td>
<td>active</td>
</tr>
<tr>
<td>T6</td>
<td>write</td>
<td>waiting</td>
</tr>
</tbody>
</table>

...returns a previous entry which matches the description according to parameter_list2. If parameter_list2 is omitted, it simply returns the entry right before the one described by parameter_list1. For example, prev(tranid) returns the entry right before the entry of tranid. prev(tranid;operation) returns the first entry before tranid with the operation type equal to operation. Since prev() is used to find an earlier data access, the entry described by parameter_list1 must be unique. To access the information of one particular entry in the table, simply add the attribute name after prev(), separated with a period. In the example of Table 3.1, prev(T5) returns the entry of T4, which acquires a read lock. prev(T5).tranid would be T4, and prev(T5).operation would be read. prev(T5;committed) returns the entry of T2, whose write
operation has been committed. This function is especially useful in determining the serializable order for the wait-for relationship.

• \( \text{succ}(\text{parameter_list}1;\text{parameter_list}2) \) is similar to \( \text{prev()} \) except that it returns an entry after the one described by \( \text{parameter_list}1 \). If no entry fits into the description, \( \text{succ()} \) returns 0.

2) Functions used to retrieve information regarding data accesses in some state. Since the status of a data access is significant, functions based on the status of data accesses can be defined. Some examples are:

• \( \text{active()} \) returns the last (if more than one) entry in the table whose status is active. To find out all active transaction requests, one can combine the use of \( \text{active} \) and \( \text{prev}() \).

• \( \text{waiting()} \) returns the last entry in the table whose status is waiting. Parameters could be provided to distinguish entries waiting for read or write operations. For example, \( \text{waiting(read)} \) returns the last entry of a read request.

• \( \text{committed()} \) returns the last entry in the table whose status is committed. Parameters could be provided to distinguish the types of operation performed. For example, \( \text{committed(write)} \) returns the entry of the latest committed write operation.

• \( \text{finished()} \) is similar to \( \text{committed()} \).

To randomly select an entry in the AccessTable, the following function can be very helpful.
any\( \text{parameter\_list} \) returns any entry whose attributes are matched with \text{parameter\_list}.

3) Functions used to modify the AccessTable:

- \text{enter(\text{parameter\_list})} is used to add a new entry.

- \text{modify(\text{parameter\_list1};\text{parameter\_list2})} is used to modify the entry described by \text{parameter\_list1} to contain the values listed in \text{parameter\_list2}.

- \text{remove(\text{parameter\_list})} is used to remove an unwanted entry.

The data object is responsible for entering information into the table by \text{enter()}. Each new entry is added at the end of the table with no exceptions. However, the data object can modify an entry in the table by \text{modify()} and remove an entry by \text{remove()}. Such a table is sufficient to describe the dependency relationship for any concurrency control schemes. The actual data manipulation such as retrieval and update are not shown in the table; the table is not to simulate, for example, logs in database systems.

Besides the AccessTable, local variables are used to maintain other information. Taking the locking scheme for example, local variables are needed to keep track of the number of locks that have been assigned by the data object and data objects that have been visited by transaction
handlers. All local variables must be declared in this part to give an idea of how much data storage would be needed to implement a concurrency control scheme.

3.3.2 The Concurrency Control Invariant

The invariant indicates what is guaranteed by an object to other objects in the model. Although the internal processing of an object is not visible to other objects, the invariant must always be satisfied at the time this object communicates with other objects. In other words, the invariant represents the behavior of an object to the outside objects. Proving the correctness of one scheme can be carried out as proving individual objects and then integrating various proofs.

3.3.3 Declaration of Message Types

Since message sending and receiving are the only ways objects communicate with each other, the types of messages exchanged by objects play an important role in explaining the entire scheme. Since different actions could be taken upon the receipt of different types of messages, message types must be clearly defined in order to specify the processing part of an object. Also, by declaring all types of the messages handled by an object, it is easier to validate the specification of two interacting
objects against their completeness in including all possible incoming messages.

Each message type is declared with parameters. Message types are used to identify what kind of a message it is while the parameters actually contain the contents of the message. We have omitted the information pertaining to both the source and the destination of a message, which can be added to the sending or receiving mechanisms described next. This kind of messages also appeared in [3]; however, our specification declares the types of both input and output messages. Some examples of message types are read or write requests, and request to commit and request to abort, etc. Due to the fact that the types of messages vary with the types of objects, this part in our specification reveals the behavior of the objects in some sense.

3.3.4 The Processing

This part describes the synchronization tasks performed by an object in order to control concurrency problems. The actual data retrieval and update, lock assignment, and lock release have been hidden away from the user at this layer.
Nondeterminism and guarded commands [17, 27, 5, 11] have provided an elegant way of specifying possible alternatives without built-in preferences as in normal sequential programming languages. In the case that preferences among possible alternatives must be distinguished, only slight modification is needed as in [19]. We have adopted CSP as our host language for describing the processing involved in exercising concurrency control; however, we have made several modifications. We shall briefly introduce the language pertaining to our use here with the modifications we have made.

- Assignment commands - An assignment command specifies evaluation of its expression and assignment of the denoted value to the target variable. For example, the command

\[
\text{tranid} := \text{succ(tranid; \text{`waiting'})}.\text{tranid}
\]

involves the call to the function \text{succ}, which returns one entry in the AccessTable (i.e., the next entry with status equal \text{`waiting'} after the entry of \text{tranid}), and the attribute value of the entry returned is assigned to the target variable \text{tranid}.

- Alternative commands - An alternative command specifies execution of exactly one of its constituent guarded commands. Therefore, if all guards fail, the alternative command fails. In our specification, if all guards fail, this alternative command will have no effect.

- Input and output commands - Input and output commands specify communications between two concurrently executing objects. One major modification is that we distinguish types of messages instead
of processes involved in communications. This is because objects such as transaction handlers are dynamically created and terminated. In case it is desirable to include destination processes, they can be added as the normal CSP representation, i.e., \texttt{messagetype}(\texttt{contents})?\texttt{processid}, or with both the source and destination processes included in the contents of the message, i.e., \texttt{messagetype}(\texttt{source,contents,destination})?. The modified CSP is equally powerful as the original CSP because message types can be modeled in the original CSP with one separate process to handle each message type. Hence sending/receiving a message of type X can be considered as sending/receiving a message to/from the process responsible for message type X. The input guard in our language contains a message type; the guard is satisfied, if a message of that type has arrived. The communication is done asynchronously, since the separate processes can function as buffers for storing and forwarding messages. This avoids communication deadlocks.

- Repetitive commands - A repetitive command specifies as many iterations as possible of its constituent alternative commands. Hence, when all guards fail, the repetitive command terminates. This implies that when input commands are included as guards in a repetitive command, all guards fail if all the sources named by the input guards have terminated. However, this definition is not applicable here, since in our specification data objects never terminate while transaction handlers may cease to exist after their transactions have terminated. This is why we distinguish message types rather than sources; even though processes may be created and may terminate dynamically, the types of messages remain the same.
3.3.5 Related Issues

Since some concurrency control schemes require rollback as a means to maintain database consistency, necessary information is kept in order to return to a previous state. Similar information is also needed by recovery schemes to recover the system from system failures to a previous consistent state [52]. Although the reasons for restoring the system to a previous state are different in concurrency control than in recovery, the information maintained for backup purposes may well be shared by both schemes. Furthermore, choosing the right recovery scheme to go with a concurrency control scheme might increase the system performance [1].

Another important issue is replica control. In real systems it is very likely that data will be replicated for both availability and reliability reasons. The replica control can be directly implemented into the concurrency control scheme or solely handled by data objects as mutual consistency problems. Our specification allows both to be expressed. For example, in the case of the read-any-write-all replica control, the transaction handler can be specified to request locks on all copies of data objects whenever a write occurs, if it is the responsibility of transaction handlers; or it could be specified in data objects as internal synchronization among all replicated copies, if it is the responsibility of data objects.
Since each of these issues relating to concurrency control is so complicated that they have been discussed separately, this part simply provides a means to integrate the problems in these areas with the concurrency control problem. Because the emphasis here is on concurrency control, issues included in this part are described in a very high level fashion under the assumption that users are familiar with the terms used in [52].

3.4 Examples

In this section, two-phase locking, timestamping, and commit-time validation are described using the proposed specification method. The emphasis here is placed on the data objects instead of transaction handlers due to the following two reasons. First, the behavior of data objects in these three schemes shows the significance of these schemes more clearly than transaction handlers. The determination of wait-for relationships in the two-phase locking, the maintenance of a serial order of timestamps in the timestamping, and the validation of data items accessed by transactions are all performed by the data objects. Second, there is lack of distributed control between data objects and transaction handlers in these schemes. As shown in the next two chapters where distributed control is considered, the behavior of transaction handlers can be specified in the same fashion as data objects.
3.4.1 Two-phase Locking

Figure 3.5 shows how a two-phase locking scheme could be specified from the viewpoint of a data object. The concurrency control datablock needs to include the AccessTable, and two local variables. The AccessTable contains three attributes, since the information of interest to a data object in this scheme is simply the transaction identifier, the type of lock requested, and the status of each request. Local variables are needed to keep track of the total number of outstanding locks assigned; two local variables also imply that only two types of locks are implemented under this scheme.

The invariant part stated that write is exclusive and read is sharable. From the definition of messages, one can see that only \textit{readreq} and \textit{writereq} pertaining to data access are included. This indicates that only the two kinds of data requests will be generated by transaction handlers, hence, no prewrites. Other types of messages are used for committing or aborting a transaction; among them the wait\_for message is used to send out information necessary for deadlock detection.

The processing part describes how read and write requests are handled. What is worth noticing here is that the two-phase commit protocol is included as part of the processing of \textit{commitreq}. And because of the structure of the CSP language, the concept of nested transactions
Object data\_object X:

Concurrency control datablock:

\{ #readlocks and #writelocks are the number of outstanding locks \}
\{ The entries of AccessTable have three fields \}
\{ operation = (read, write); \}
\{ status = (committed, finished, active, waiting); \}
#readlocks : integer; initially 0;
#writelocks : integer; initially 0;
AccessTable = ( tranid, operation, status );

Concurrency control invariant:
\((#readlocks = 0 \land #writelocks \leq 1) \lor (#writelocks = 0 \land #readlocks \geq 0)\)

Message type declaration:
\{ input message types \}
readreq(tranid);
writeq(tranid);
commitq(tranid);
abortq(tranid);
commit();
\{ output message types \}
ready();
wait\_for(tranid1, tranid2);

Processing:

\* \[ readreq(tranid)? ---\> \[ #writelocks \geq 1 \rightarrow \]
\[ \] #writelocks \geq 1 \rightarrow [ enter(tranid,'read','active');
\[ \] #readlocks:=#readlocks+1 ]
\[ \] #writelocks \geq 1 \rightarrow [ enter(tranid,'read','waiting');
\[ \] wait\_for(tranid, prev(tranid;\'write\')); ]
\]
\[ writeq(tranid)? ---\> \[ #readlocks \geq 1 \land #writelocks \geq 1 \rightarrow \]
\[ \] #readlocks \geq 1 \land #writelocks \geq 1 \rightarrow [ enter(tranid,'write','active');
\[ \] #writelocks:=#writelocks+1 ]
\[ \] #readlocks \geq 1 \land #writelocks \geq 1 \rightarrow [ enter(tranid,'write','waiting');
\[ \] prev(tranid).operation = 'read' ---\> [ tranid2:=tranid;\* (prev(tranid2).operation='read' ---\> [ tranid2:=prev(tranid2).tranid;
\[ \] wait\_for(tranid, tranid2) ]
\[ \] prev(tranid).operation='write' ---\> wait\_for(tranid, prev(tranid).tranid); ]
\]
\]

Figure 3.5. Specification for Two-phase Locking Scheme
Other issues:

assignlock(tranid): [ active.operation = 'read' --> #readlocks := #readlocks - 1;

  [ succ(tranid; waiting).operation='read' -->
    modify(succ(tranid; waiting); tranid; active);
    #readlocks := #readlocks + 1;
    tranid := succ(tranid; waiting).tranid ]

  [ succ(tranid; waiting).operation='write' &
    #readlocks = 0 -->
    modify(succ(tranid; waiting); tranid; active);
    #readlocks := #readlocks + 1 ]

  active.operation = 'write' --> #writelocks := #writelocks - 1;

    [ succ(tranid; waiting).operation='write' -->
      modify(succ(tranid; waiting); tranid; active);
      #writelocks := #writelocks + 1;

      [ succ(tranid; waiting).operation='read' -->
        modify(succ(tranid; waiting); tranid; active);
        #readlocks := #readlocks + 1;
        tranid := succ(tranid; waiting).tranid ]

]}

Recovery is based on logging

Figure 3.5 continued
is clearly revealed here. As one can see from the specification, the data manager will not process any other message once it enters the two-phase commit protocol, during which it waits for the final decision as to commit or not.

In the last part, the details regarding lock assignment are included. For this particular two-phase locking scheme, logging is selected as the recovery means.

3.4.2 Timestamping

Figure 3.6 shows the specification for a basic timestamping scheme. There is one more attribute in the AccessTable, since the timestamps of transactions that have accessed the data determine whether subsequent requests can be granted or not. Local variables are used to keep track of the highest read and write timestamps. As shown here, no multi-version [7, 43] is applied in this example.

The invariant part states that a write request can be granted only if its timestamp is greater than the timestamp of the previous write (by the first statement) and is greater than the timestamps of all the previous reads (by the third statement). The two any(read,finished) in the second statement refer to the same entry.
Object data.object X;

Concurrency control database:
| its and wts are timestamps for the read and write operations | operation = (read, write); |
| The entries of AccessTable have four fields | status = (committed, finished, active, waiting); |
| Its : integer; initially 0; | its ; integer; initially 0; |
| Wts ; integer; initially 0; | AccessTable = (tranid, trans, operation, status); |

Concurrency control invariant:
| finished('write').trans > prev(finished('write').tranid,'write').trans |
| finished('read').trans > prev(finished('read').tranid,'write').trans |
| any('read', finished').trans < succ(any('read', finished').tranid,'write').trans |

Message type declaration:
| input message types | readreq(tranid,trans); |
| | writereq(tranid,trans); |
| | commitreq(tranid); |
| | abonreq(tranid); |
| output message types | accept(); |
| | reject(); |

Processing:

* readreq(tranid,trans)? —> [ trans > wts —> enter(tranid,trans,'read','finished'); |
| | trans > wts —> wts := trans; |
| | accept(); |

* writereq(tranid,trans)? —> [ trans > wts | trans > wts —> enter(tranid,trans,'write','finished'); |
| | wts := trans |

* commitreq(tranid)? —> modify(tranid,'committed') |

* abonreq(tranid)? —> rollback(); remove(tranid,'read' | 'write') |

Other issues:
Recovery is done using shadow pages

Figure 3.6. Specification of Timestamping
In addition to the four common types of input messages, *accept()* and *reject()* are used respectively to notify the requesting transaction handler whether a request succeeds or not. The decision can be made immediately because the timestamping scheme only relies on local information to schedule data requests. And for the same reason, the processing needed is simpler than in the two-phase locking scheme as shown by the processing part. The two-phase commit protocol can also be applied in order to preserve fault-tolerance.

### 3.4.3 Commit-time Validation

Figure 3.7 shows an example of specifying commit-time validation using our high level specification. Since every write generates a temporary copy of the data, all copies are assigned a version number in order to differentiate their order. \texttt{C\_version} is used to keep track of the current version number of the latest copy. Because validation is performed at the end of a transaction, operations will be committed only if no other transactions (committed after the time the current transaction started but before the current transaction finishes ) have invalidated the data. The AccessTable maintains all information regarding committed writes and their version numbers. The invariant states that writes will be committed according to the version numbers of their update copies in increasing order.
In the processing part, a read request is always granted while a write request creates a newer version of the data (which is invisible to other reads at this point). When a transaction requests to commit, the data object enters a critical section. If any previous read has been invalidated (the $C_{version}$ is greater than the version it read), it simply rejects the request, otherwise it replies with a positive message and waits for the final decision as determined by other data objects.

3.4.4 Comparison of the three schemes

From the above examples, we can derive the following results. The timestamping scheme is the easiest one among all three to specify. This is because it is very straightforward and requests are all determined by local status. It is deadlock free; therefore, there is no need to check the AccessTable to collect wait_for relationships between conflicting transactions. However, the drawback of this scheme is actually hidden inside the rollback() routine. When one restart induces a sequence of other restarts, the performance of this scheme will definitely deteriorate.

The commit-time validation is the most difficult one to specify, because of the existence of several temporary copies of the data and only one of which can finally commit. And because the readset of a committing transaction must not intersect the writerset of precommitted trans-
Object data_object X:

Concurrency control datablock:
(C_version is the current version number of updates
| The entries of AccessTable have four fields
| operation = (read, write);
| status = (committed, finished, active, waiting);
| C_version : integer initially 0;
AccessTable = (tranid, version, operation, status);

Concurrency control invariant:
committed('write').version > prev(committed('write').tranid; 'committed').version

Message type declaration:
| input message types
readreq(tranid);
write(req(tranid);
commitreq(tranid);
abortreq(tranid);
commit();

| output message types
ready();
reject();

Processing:
* readreq(tranid)? --- enter(tranid, C_version, 'read', 'finished')
  |
write(req(tranid)? --- any(tranid, 'read') = 0 --- enter(tranid, any(tranid, 'read').version+1,
  | 'write', 'finished')
  | any(tranid, 'read') = 0 --- enter(tranid, C_version+1,
  | 'write', 'finished')
  |
abortreq(tranid)? --- remove(tranid, 'read' I 'write')
  |
commitreq(tranid)? --- [ any(tranid, 'read') ≠ 0 ---
  | [ any(tranid, 'read').version ≥ committed('write').version
  | --- ready();{ commit? ---
  | [ modify(tranid, 'read', 'committed');
  | any(tranid, 'write') ≠ 0 ---
  | C_version := any(tranid, 'write').version;
  | enter(tranid, C_version, 'write', 'committed')
  ]
  | [ abortreq? --- remove(tranid, 'read' I 'write')
  ]
  | any(tranid, 'read').version < committed('write').version
  | --- reject(); remove(tranid, 'read' I 'write')
  ]
  | any(tranid, 'read') = 0 --- ready(); { commit? ---
  | C_version := committed('write').version+1;
  | enter(tranid, C_version, 'write', 'committed');

Figure 3.7. Specification of Commit-time Validation
Other issues:

Recovery is done by shadow pages
actions, as described in the original paper [30], read operations of a committing transaction must be thoroughly checked against newer committed updates. This really complicates the description of the scheme. Due to the fact that not all objects enter the critical section at the same time, two committing transactions might cause deadlock, if no preventive method is taken. This problem can be detected if the communication deadlock problem [35] is taken into consideration by the scheme. However, aborts are very easy to handle since no rollback is required.

As for the two-phase locking, the count of outstanding locks really simplifies the whole thing. Although lock assignment becomes more complicated as more types of locks are implemented, the scheme in general can be specified with a reasonable amount of effort.

3.5 Conclusions

We have presented a high level specification based on an object-oriented model for specifying concurrency control schemes. The specification method can also be used as a design tool. When used as a design tool, the distributed concurrency control handled by individual objects can be specified separately. The model provides both the data structure and communication mechanisms needed by concurrency control schemes. The specification also eliminates the implementation details during the
course of designing. Because different kinds of objects are specified separately, the process of validation or verification can also be broken down accordingly.

In this chapter we have shown the need for such kind of specification and its importance. To further explain the use of the model, examples of three different types of concurrency control schemes have been included and compared. By a uniform representation, concurrency control schemes can be compared in terms of the storage needed for specifying these schemes, the distribution of control between transaction handlers and data managers, etc.

Since little work has been done in the specification of concurrency control schemes, our goal here was to present a formal way of describing concurrency control schemes. Although the specification method presented in this chapter may be enhanced in the future work, we have laid the foundation of formally describing concurrency control schemes. Other future work includes integrating recovery into the specification of concurrency control schemes, and validating and verifying concurrency control schemes based on their specifications.
CHAPTER IV
Nonfreezing Optimistic Scheme

4.1 Introduction

Optimistic concurrency control schemes have the tendency of partially freezing the system in order to validate a transaction at the end of the transaction's execution phase. This chapter presents first an optimistic concurrency control scheme that does not freeze the database system and then a formal specification of the scheme. Based on our design philosophy of distributed control, concurrency control is divided into two counterparts. While one part is exercised at the data object, which responds instantly to consistency violation, the other part is exercised by simple messages flowing, as background tasks, between transaction handlers. The whole system need not stop executing transactions during the time concurrency control is exercising nor during the time the system is restoring its state when consistency is violated. This makes the scheme more attractive, since a higher degree of parallelism is provided between consistency control and transaction execution. This scheme is based on
the object-oriented model for a distributed database system in which communication between objects relies heavily on message passing.

Among the popular concurrency control schemes, commit-time validation [30] is considered optimistic, as compared to two-phase locking (2PL) [21, 39, 6], timestamp ordering [49, 31, 6], and a combination of both [6, 44, 41]. An optimistic scheme assumes that transactions rarely conflict; hence transactions execute independently with updates deferred till the commit time. Updates of a transaction will not be made to the database unless the system validates that consistency is preserved after the execution of the transaction. The validation phase involves checking the set of data objects read by the transaction with the set of data objects written by committed transactions. Consistency is preserved, if there is no intersection between these two sets. The validation process, however, tends to freeze a portion of the system (the portion that is being validated against the committing transaction), since no other transactions can enter the validation phase at the same time.

Because the global status of the system is costly to obtain, this shortcoming becomes even worse, when concurrency control is extended from a single-site database system to a distributed database system in which each stand-alone system is represented by a node connected to other nodes by communication links. In this chapter, we present an op-
timistic concurrency control scheme that records information related to conflicting transactions. The information stored can be used to validate transaction execution while the transaction is updating the database. Since our scheme does not defer transaction updates, the recorded information can also be used to restore the database. Neither validation nor restoration freezes the system.

The schedule of operations in a database system is correct, if and only if it is serializable, which assures that transactions executing concurrently will have the same effect as if they were executed sequentially. To acquire a serializable schedule, every precedence relationship must be preserved on every data object in the system. In other words, the consistency of a database system can be observed from a lower level, referred to as the data-object level, at which operations of one transaction must either precede or follow another transaction's operations to the same data object. Its consistency can also be observed from a higher level, referred to as the system level, at which all precedence relationships must be consistent such that a partial ordering among all concurrently executing transactions exists.

To incorporate our design philosophy into the optimistic concurrency control schemes, we distributed the consistency control to data objects and transaction handler objects. This will increase the degree of
concurrency in consistency checking. Therefore, the scheme captures the idea of real optimism without deferring transaction updates while the validation process runs concurrently with transaction executions.

This chapter is organized as follows: Section 4.2 describes the model and assumptions used to present our scheme. We will delineate our model with details pertaining to this scheme. Section 4.3 presents our concurrency control scheme. Section 4.4 discusses state restoration due to transaction rollbacks; a rollback procedure using checkpointing to avoid the cascade of aborts is included. Section 4.5 gives an informal proof of the correctness of the scheme. Section 4.6 compares the proposed scheme with other schemes. Section 4.7 presents the specification of this scheme with some explanations. The last section concludes this chapter with some final remarks.

4.2 System Model and Assumptions

We describe the object-oriented design model with more details for the purpose of presenting our scheme in this section. The model represents a distributed database system, which consists of a finite set of transactions guarded by transaction handlers, channels and physically scattered data objects which logically constitute the database. Therefore, several transaction handlers and data objects reside on each node of the
network with the union of the data objects equaling the complete database. We present our scheme in a high level fashion to avoid discussing implementation details.

A transaction consists of a series of operations, such as read and write, requested by the user. A special process, called transaction handler (TH), is created by the transaction manager (TM) to handle a transaction. A TH acts as a guardian of a transaction; the job of a TH includes interacting with data objects to get transaction requests serviced, detecting system level consistency, committing a transaction, and aborting a transaction if the transaction fails. A transaction tries to run to completion and waits only for the output from the database or the permission to commit, whereas the associated TH terminates when the transaction commits. If a transaction fails to commit, the TH keeps running until it is no longer needed in maintaining the system level consistency.

A data object is an abstract data type; in addition to the data stored, it has its own operations to manipulate the data. Here we adopt the logging scheme, for the history of data accesses plays an important role in the data-object level consistency control, in addition to recovery. THs interact with data objects via message passing; all requests from THs will be queued and processed in a sequential order. A request is
serviced only when it does not violate the data-object level consistency. When a new precedence relationship occurs, a data object notifies the requesting TH of conflicting transactions via messages. Each data object can also handle rollbacks when transactions fail to commit; certain undo and redo logic is implied.

Whenever a new precedence relationship occurs between two conflicting transactions, a channel will be set up between the two associated transaction handlers. Setting up a channel requires the initiating TH to send a control message with the type of the channel to the destination TH. Both will update their local information about the newly created channel. We shall refer to the initiating TH as the child TH* and the destination TH as the parent TH with respect to the channel between them. A channel between two transaction handlers conveys the ordering between the two conflicting transactions as well as the data object(s) with which these two are in conflict.

If we consider each data object to be one unique type, we can associate some type(s) with each channel. The type of a channel is determined by the type of the data object(s) on which the precedence order

---

*We identify the initiating TH as the child because from the viewpoint of a sequential execution, it is affected by the output from the parent TH.
exists. If two transactions are in conflict on more than one data object, say data objects X and Y, with the same precedence relationship, then the channel between them will be of both types X and Y. In other words, the type of a channel is simply the set of data objects on which the two transactions are in conflict. Channels are associated with types so that when transaction abortion occurs, the serializable order of transactions is automatically adjusted as described in the next section. This is totally different from conventional schemes, such as a graphing scheme in which a graph is constructed with each node representing a transaction handler and each arc representing the dependency between two nodes. When a cycle exists in the graph, which indicates some executions are not serializable, removing a node requires a graph reconstruction. The type of a channel could expand during the execution depending on the degree of interaction between these two transactions.

For the ease of describing the scheme, all channels mentioned in this chapter are conceptual such that a channel can be set up between any two transaction handlers, regardless of where the two transaction handlers reside. The advantage of choosing conceptual channels in our model is that we do not have to pay attention to physical details including network topology, which are concerned only at the implementation level. However, these conceptual channels can be part of real channels between nodes for interprocess communication.
Unless indicated otherwise, most of the messages mentioned in this chapter are control messages for concurrency control purposes that are different from information messages for computations. These control messages are sent over typed channels. Messages may be delayed, and it is assumed the network guarantees their delivery, and they will be received in the same order they were sent. In the next section we describe the underlying concurrency control scheme that determines when the precedence order occurs, when channels are created, how inconsistency is detected and how to recover from an inconsistent state to a consistent one.

4.3 The Concurrency Control Scheme

4.3.1 Data-Object Level Consistency

To maintain serializable order at data-object level, a transaction is not permitted to write a data object if the data object it reads has been updated by another transaction. In other words, the data it reads has become obsolete. A locking concurrency control scheme eliminates such a problem by locking the data object so that other transactions cannot modify it. However, efficiency is being sacrificed because other transactions are unnecessarily kept waiting to read between the time the
write lock is set and the time the data is actually written. If it takes a
long time for a transaction to process retrieved data before updating it,
the system throughput will greatly decrease since other transactions are
blocked during such a long period.

In our object-based model, operations as well as private data are
encapsulated in the data object. This enables data objects to control ac­
cess from transactions. The log kept by a data object, as in any
database systems, contains the history of the operations performed. The
information contained in the log shows existing precedence relationships
among transactions that have accessed the data object.

According to the concurrency control scheme, a write request is
rejected if the data has been read by the same transaction earlier and
has been updated by another transaction since then. Since transactions
are not required to specify whether they will later change the data or
not when they retrieve them, a data object needs to check the active log
to see if an earlier read was issued by the same transaction. The data
object can also determine what kind of dependency would incur due to
the request if accepted. This does not cause extra overhead, for only a
small portion of the log needs to be searched. This kind of overhead is
unavoidable whenever concurrency is allowed; other schemes, such as
locking or timestamping, require the lock table or read/write timestamps
to be checked and set. When a request is accepted, the data object performs the operation, records it into the log, and replies with the information about the new precedence relationships. Otherwise, the TH will be notified of the rejection.

4.3.2 System Level Consistency

The system level consistency requires that all the transactions are serializable, which is done by channels between conflicting transactions and by messages sent over the channels to detect any nonserializable cases. One channel is needed between every two transaction handlers that have the precedence relationship imposed on their transactions. With all the channels considered, the systemwide global serializable ordering can be obtained easily. We use dynamic channels to allow the flexible expansion and shrinking of precedence relationships. The types associated with channels play an important role in dynamically merging or splitting precedence relationships defined by different data objects. Our method is similar to the edge-chase in [33]; however, with multiple-type channels ours is more complicated than in [33].

When a transaction aborts, its effect on the data objects will be undone. This of course changes the overall view of the system, since the transitive property of the dependency may no longer exist. This
means that if the aborted transaction is in conflict with its parent transaction on a data object different from the one on which it conflicted with its child transaction, precedence relationships do not necessarily exist between its parent transaction and its child transaction. The following example explains this.

Table 4.1. Precedence Relationship by X and Y

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T4</td>
</tr>
<tr>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
<td>T5</td>
</tr>
</tbody>
</table>

Example I Assume the precedence relationships defined by data objects X and Y as depicted in Table 4.1. The overall view with a channel (shown by a directed line) representing each precedence relationship is shown in Figure 4.1. Figure 4.2 explains how precedence relationships are detached according to distinct types of the channels when T2 is aborted. Note that after T2 is aborted, T4 and T3 are no longer related in terms of precedence relationships.
Next we describe how messages are initiated and forwarded by THs when a TH sets up a channel, and when to commit/abort a transaction.

When a TH is notified by a data object, say of type X, about newly occurred precedence relationships, it does the following for each precedence relationship.

1) If such a precedence relationship already exists, i.e., there is a channel representing this precedence relationship, simply expand the type of the channel to include X.
2) If an opposite precedence relationship already exists, abort its transaction and restore the database to the beginning of the transaction as described in the next section.

3) Otherwise, set up a channel for the new precedence relationship. The type of this channel would be X. Also initiate a message with an increasing sequence number and the id of the initiating transaction handler and send it over the newly set up channel.

A TH handles an incoming message as follows.

1) If the incoming message was initiated by itself earlier, which means some cyclic relationship exists among transactions, the TH aborts its transaction to break the cycle in order to preserve serializability.

2) If the message was initiated by some other TH, and is received for the first time (each TH keeps track of the highest number of messages sent by every other TH), the message is then forwarded over channels to all its parent transactions. If there is no parent transaction, the message stops propagating right here.

3) If a message initiated by the same TH with the same sequence number or higher has been received and forwarded, the current message can be discarded.

System level consistency is violated if a cycle of channels exist such that at least one message will eventually be returned to the initiating TH. Therefore, a transaction is allowed to commit only when it is
guaranteed not to be involved in any cycle. This scheme is deadlock free because transactions are never blocked, and a TH can continue to receive and forward messages when it is waiting for a reply from a data object.

### 4.3.3 Commits

When a transaction finishes all the operations successfully, it cannot commit until all its parent nodes have committed. Such delay is needed to guarantee that its result does not have to be converted after being made permanent. Furthermore, sometimes it is impossible to convert the result of a committed transaction, such as printing a refund check. Therefore, when a transaction finishes its executions, it is allowed to commit, only if no transaction preceding it is still running. When a transaction finally can commit, our commit protocol requires the committing TH to remove all the channels connected to its child transactions. Only parent transactions can remove a channel connected to its child transaction; this is done by sending a message to the child to terminate the precedence relationship. Note that there is at most one link between two transaction handlers because if there is any precedence order imposed on them, either one precedes the other. However, a channel can be of more than one type. The committing TH can be terminated after removing all channels connected to its child transactions.
The abortion of a transaction may result in the abortion of child transactions. We shall discuss the issues on state restoration in the next section.

4.3.4 Aborts

Here we propose a two-phase abort protocol, which describes the behavior of a TH when it aborts its transaction. Phase one begins when the TH marks itself as waiting-to-abort, terminates its transaction, and sends a request-to-abort message to all affected data objects to undo the changes. During phase one, a TH handles incoming messages differently, since the combined view of the system is changed when the transaction is aborted. Therefore, incoming messages from the channel of type X will be forwarded only onto outgoing channels of type X.

Table 4.2. Precedence Relationship by X and Y

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T5</td>
</tr>
<tr>
<td>T2</td>
<td>T2</td>
</tr>
<tr>
<td>T3</td>
<td>T6</td>
</tr>
<tr>
<td>T4</td>
<td></td>
</tr>
</tbody>
</table>
Example II Consider Table 4.2. Before any inconsistency is detected, the precedence relationships determined by X are T1 precedes T2, T2 precedes T3, T3 precedes T4; and those determined by Y are T5 precedes T2, T2 precedes T6. Since T2 has precedence defined by both X and Y, T5 also precedes T3 and T4 from the system level viewpoint. Similarly, T1 precedes T6. Figure 4.3 illustrates channels representing the precedence relationships. Whenever a message arrives at TH2, it will be forwarded to both TH1 and TH5, as depicted by the thicker lines in Figure 4.4.

Figure 4.3. Precedence Relationship Represented by Channels
Now assume T2 has to be aborted and TH2 enters the waiting-to-abort state. Note that T2 no longer has any effect on X or on Y. However, to avoid reconstructing those channels, we keep TH2 there to maintain the relationship between T1 and T3 as well as between T5 and T6. We require channels to be typed so that, in this example, a message arriving at TH2 from TH6 will only be forwarded to TH5 due to the same type of channels as shown in Figure 4.5. It is easier to control the merger and the split of precedence relationships that are defined by different data objects. Note that during phase one, the TH
does not allow its child transactions to commit, even if all its parent transactions have terminated.

When a data object receives the request-to-abort message, it has to perform some undos and maybe some redos, depending on the recovery technique used. It could also determine which of the succeeding transactions must be aborted. Then, it sends a reply including the list of transactions that must be aborted to the waiting-to-abort TH. This is to
prevent the TH from allowing any of those transactions to commit.

When the waiting-to-abort TH receives all the replies, it enters the second phase. Because by now all of the affected data objects are aware of the abortion, no channels involving the aborted transaction will be created. The TH has to continue to forward messages until all its predecessors have either committed or aborted; then it will remove all channels between itself and its child transactions before terminating itself. This enables child transactions to commit, since the abortion is completed.

Figure 4.6. Cycles in Precedence Relationships

One problem worth noticing is that there is a possibility of having a cascade of abortions. When a TH detects cyclic relationships, the TH

*Otherwise, due to possible message delays, those succeeding transactions could commit after the TH terminates but before receiving any 'abort' message.
can identify on which channel the returned message was sent and on which channel the message was returned. If these two channels have disjoint types, aborting the transaction will break the cycle, and only transactions that read the data object updated by the aborting transaction need to be rolled back. This is because aborting a read operation does not change any output of the system, nor does aborting a write operation immediately followed by another write operation make any difference. If these two channels have common type(s), aborting the transaction will not eliminate the cycle. For example, T2 detects a cycle involving T1 and T3 as shown in Figure 4.6, aborting T2 will separate different relationships; however, due to the fact that these two channels have a common type X, removing T2 from the cycle will still leave a cycle with T1 preceding T3 (by the channel from T3 to T2 and the channel from T2 to T1). Let us refer to these data objects on which inconsistency is caused as trouble-makers. All subsequent transactions that accessed trouble-makers must be aborted. The following section describes how a transaction can be partially rolled back when the transaction preceding it is aborted.
4.4 State Restoration

An elegant method to avoid the domino effect caused by rollbacks in an asynchronous system of communicating transactions has been proposed in [40]. However, this kind of state restoration was considered in systems with directed process interactions, i.e., interactions that are directed from one process to another. According to [40], with proper checkpointing and a history which records all messages sent and received, rolling back a process does not cause the process from which it receives messages to back out at the same time, since the messages can be reread from the history. Our restoration procedure is based on a similar idea requiring THs to record all data retrieved and make checkpoints before every read request. Since this affects subsequent transactions that read the output produced by the aborting transaction, all affected transactions will be rolled back to the point before the read was performed (this time with the effect of the aborted transaction removed). If there is a chain of transactions that need to be rolled back, the data object will monitor these operations such that they are redone in the same order as they are recorded in the log. As long as all transaction executions are serializable, no cyclic rollbacks should occur.

Due to communication delay, an aborting TH can still receive newly created channels or messages from child THs. Upon receiving such
a message from a child TH, the waiting-to-abort TH will act according to the following cases.

1) If the new channel or new type came from a trouble-maker, accept the channel or the type, but do not forward it, since the child transaction will later be aborted.

2) Otherwise, handle incoming messages as usual. The child transaction will probably be rolled back later, if it is affected by the output of the aborting transaction.

We require an aborting TH to enter phase two only after all participating data objects have completed their undo/redo procedures. This is to prevent its child transactions that need to be rolled back from committing immaturity.

4.5 Informal Proof of Correctness

In this section, we give an informal proof of the correctness of our scheme. First of all, our scheme is deadlock free since transactions wait only for the replies from data objects when read or write requests are issued. Hence it allows us to ensure that the data object will never be blocked, which might cause the requesting transaction to wait forever. Since the data object determines whether to serve or reject the request based on its log, no transaction would be kept waiting forever.
Second, we need to prove that transactions involving in cyclic relationships cannot commit until the cycle is resolved. Since channels representing precedence relationships are created at the time transactions make data access requests, cyclic relationships mean that several transactions and the channels associated with them form a cycle. Because of the condition that governs transaction commits, i.e., a transaction cannot commit until all its outgoing channels have been removed, none of these transactions involving in a cycle can commit.

Finally, we need to show that in the case that cyclic relationships do exist, eventually the cycle will be detected and resolved. Since a message will be initiated and forwarded to all its ancestors whenever a new channel is created, eventually at least one message will have traveled through every node in the cycle and returned to the initiator, thus detecting the cyclic relationship. Let's consider a cycle involving n nodes, and use $T_0..T_{n-1}$ to represent the n nodes involved in the cycle in order. That is,

$$T_0 \rightarrow T_1 \rightarrow .... \rightarrow T_{n-1} \rightarrow T_0$$

Let $M_i$ represent the message initiated upon the creation of channel $T_i \rightarrow T_{(i+1) \mod n}$.
The proof can be given as follows. Assume $M_0$ did not return to $T_0$, and $M_0$ has traveled to node $T_j$ the farthest. This implies at that time

$$T_j \rightarrow T_{(j+1) \mod n} \text{ does not exist, } 1 \leq j \leq n-1$$

When $T_j \rightarrow T_{(j+1) \mod n}$ creates, $M_j$ will be initiated and forwarded. If $M_j$ did not return to $T_j$, then $M_j$ must have traveled to node $T_j$, the farthest, where $j < j' \leq n-1$. Note that there is already a path from node $T_0$ to $T_j$.

If we repeat this step for a maximum of $n$ times, eventually at least one message will be returned to the initiating node, thus detecting the cycle. Hence, this scheme is guaranteed to be correct.

4.6 Comparison to Other Schemes

Our scheme differs from other optimistic schemes due to the fact that updates are not deferred. This is based on the argument that optimistic schemes that defer updates till the write phase have a higher overhead of conflicting transactions, even though the overhead of nonconflicting transactions is low. Our scheme does not incur a greater overhead of nonconflicting transactions than those optimistic schemes, since no channels are needed. For conflicting transactions, our scheme requires
fewer restarts than the scheme proposed in [30], since updates made by 
one transaction can be accessed by subsequent transactions sooner. Hence 
subsequent transactions are unlikely to be aborted.

When considering the scheme proposed in [9], which maintains 
precedence relationships among validating transactions, our scheme will 
have better performance because we allow transactions to commit in any 
possible serializable order without any need of validation. The scheme 
proposed in [9] relaxed the condition for validation a little bit by allow­
ing transactions that preserve serializable order to commit not necessarily 
according to their order of entering the validation phase. However, the 
extra freedom gained is very limited, and it still suffers from the draw­
back of getting into validation phase.

Our scheme can be compared to the non-two-phase locking schemes 
[44] due to the fact that updates are made to the database im­
mediately. The major advantage our scheme has over the non-two-phase 
locking schemes is that our scheme detects inconsistency 'on-the-fly', 
using an elegant approach that does not require centralized control nor 
reconstruction of the relationships upon aborts.
4.7 Formal Specification

In this section the formal specification of the nonfreezing optimistic scheme is presented and discussed. Our purpose of presenting the scheme using the high level specification is twofold. One is to show the difference between the way schemes are usually presented, as shown in previous sections, and the way high level specification is used. The other purpose is to provide more examples of the use of the specification method.

Based on the abstract view of a distributed database system as mentioned in Chapter 3, two kinds of objects are considered. The transaction handler coordinates interactions due to data access, and terminates itself when the transaction terminates. It intercepts the data access operations issued in the transaction and communicates with data objects in order to have the data access operations performed. The data object schedules data requests and carries them out. A rule is provided in order to resolve inconsistency caused by concurrent access. The details of how communications are set up between two parties have been abstracted away in our model.

Since the behavior of two different types of objects is described in specifying the concurrency control scheme, for the purpose of clarity, we shall assign names to the objects and include the names of both the source object and destination object in the message content.
Figure 4.7 contains the specification of a transaction handler object, and Figure 4.8 contains the specification of a data object in the non-freezing optimistic scheme.

The transaction handler is responsible for maintaining conceptual channels with appropriate types (the types of data objects that created them) and for forwarding messages along the channels to detect any inconsistency. Therefore, an array(Out_c) for keeping track of all outgoing channels and their types is used as part of the concurrency control database. Similarly, array In_c is used to maintain all incoming channels. Since a transaction cannot be aborted without notifying all the data objects visited, the array Data_lists is specially used to specify the two-phase abort protocol in the specification. Wait_to_abort and Wait_to_commit are two state variables, since the way messages are handled depends on the current state of the transaction handler.

In this scheme, communications occur not only between data objects and transaction handlers, but also among transaction handlers themselves. This is fully reflected by the message declaration part. To make sure all message types are accounted for, we can validate all output message types of one object against the input message types of the object with which communications take place.
Object transaction_handler Y:

- Out_c keeps tracks of all the outgoing channels, especially their destination (dest) and types (dtypes).
- In_c keeps tracks of all incoming channels, their source (source) and types (dtypes).
- data_lists keeps tracks of all the data objects visited.
- ptr_out, ptr_in, ptr_data show the number of entries in out_c, in_c, data_lists.
- wait_to_abort and wait_to_commit are two state variables.

Concurrency control datablock:
- out_c = array [1..maxint] of (dest, dtypes);
- in_c = array [1..maxint] of (source, dtypes);
- data_lists = array [1..maxint] of dtype;
- ptr_out, ptr_in, ptr_data: integer;
- wait_to_abort: boolean, initially false;
- wait_to_commit: boolean, initially false;
- i, found: integer.

Message type declaration:

- retrieve(dtype); [operation issued by the transaction]
- update(dtype);
- accepted(dtype); [messages from data objects]
- reject(dtype);
- dependency(tranid, tranid2, dtype);
- forward(Initiator, Initiator, source, dest); [messages from other transaction managers]
- cancel(source, dest);
- abort(source, dest, dtypes);
- new_link(source, dest, dtype); [output message types]
- readreq(tranid, dtype);
- writereq(tranid, dtype);
- commitreq(tranid, dtype);
- abortreq(tranid, dtype);
- new_link(source, dest, dtype);
- forward(Initiator, Initiator, source, dest);
- cancel(source, dest);
- abort(source, dest, dtypes);

Processing:

- retrieve(dtype) ? —> readreq(Y, dtype)
- update(dtype) ? —> writereq(Y, dtype)
- accepted(dtype) ? —> accept(Y, dtype)
- reject(dtype) ? —> reject(Y, dtype)
- dependency(Y, tranid, dtype) ? —> found = 0; i = 1;
- forward(Y, tranid, dtype) ? —> found = 0; i = 1;
- cancel(Y, tranid, dtype) ? —> cancel(Y, tranid, dtype)
- abort(Y, tranid, dtype) ? —> abort(Y, tranid, dtype)

Figure 4.7. Specification of the Transaction Handler Object
new_link(source, Y, dtype)? \rightarrow ptr_in := ptr_in + 1;
  in_c[ptr_in].source := source;
  in_c[ptr_in].dtype := dtype;
  \{ wait_to_abort \rightarrow i := 1; \[i \leq ptr_out \rightarrow [out_c[i].dtype \land (dtype \neq \phi) \rightarrow forward(source, Y, Y, out_c[i].dest)];
  i := i + 1 \}
  \} \rightarrow wait_to_abort \rightarrow i := 1; \[i \leq ptr_out \rightarrow forward(source, Y, Y, out_c[i].dest); i := i + 1 \]

wait_to_abort \land ptr_data = 0 \rightarrow i := 1; \[i \leq ptr_in \rightarrow cancel(in_c[i].source, Y)];
  i := i + 1; terminate() \}

abort done(Y, dtype)? \rightarrow ptr_data := ptr_data - 1
The behavior of a transaction handler is described by using a repetitive command in CSP. The repetitive command consists of several alternatives with input guards. This indicates that a transaction handler may receive many types of messages and continue to execute until the transaction terminates. The details about this scheme will not be repeated here. However, the behavior of a transaction handler is clearly shown here by the tasks performed upon the receipt of messages. Since the scheme divides concurrency control into data-object level and system level, the concurrency control enforcement is naturally distributed to both data objects and transaction handlers. From the specification of the transaction handler, we can see how channels are created and destroyed and how messages are forwarded. Also, the system level consistency is maintained by checking for any cyclic relationships in the dependency relationships and by resolving them at runtime. To fully understand this scheme, one has to refer to the specification of the data objects.

Since the data-object level consistency is achieved locally by not allowing any write to perform, if the data a transaction read earlier has been invalidated by writes from other transactions. This is stated in the invariant part of the specification of a data object; either a transaction writes without first reading it, or no other writes have invalidated the data it read.
Object data_object X;

Concurreny control datablock:
( The entries of AccessTable have three fields
operation = (read, write);
status = (committed, finished, active, waiting);
AccessTable = ( tranid, operation, status );

Concurreny control invariant:
Let any('write','done').tranid = t
any(t,'read')=0 v succ(t,'read','write').tranid=t

Message type declaration:
( input message types )
readreq(tranid, dtype);
writereq(tranid, dtype);
commitreq(tranid, dtype);
abonreq(tranid, dtype);
( output message types )
dependency(tranid1,tranid2, dtype);
accept(tranid, dtype);
reject(tranid, dtype);
abandon(tranid, dtype);

Processing:

* [ readreq(tranid,X)? ---e enter(tranid,'read','done');
dependency(tranid,prev(tranid,'write'),tranid)!

[] write req(tranid,X)? ---e any(tranid,'read'=0 ---e
[ enter(tranid,'write','done');
[ prev(tranid).operation='read' --->
tranid2=tranid;
[*prev(tranid2).operation='read' --->
dependency(tranid,prev(tranid2).tranid)!
tranid2=prev(tranid2).tranid)

[] prev(tranid).operation='write' --->
dependency(tranid,prev(tranid).tranid)
]
accept(tranid,X)!
[] any(tranid,'read') = 0 --->
[ succ(tranid; 'read') = 0 --->
[ enter(tranid,'write','done');
tranid2=tranid;
[*prev(tranid2).operation='read' ---> [prev(tranid2).tranid=tranid --->
dependency(tranid,prev(tranid2).tranid,X)];
tranid2=prev(tranid2).tranid];
accept(tranid,X)!

Figure 4.8. Specification of the Data Object
Other issues:

Recovery is based on logging

Figure 4.8 continued
The message declaration part is much shorter than the transaction handler's because data objects only communicate with transaction handlers. In addition to the four common types of requests, a data object always replies with accept, reject, and the dependency caused by the request. Since a transaction cannot be aborted without receiving the confirmation regarding the abortion from data objects, abortdone is specially designed to acknowledge to a transaction handler that rollback has been completed by a data object. Because a transaction commits only if all its predecessors have committed, there is no need to rollback after committing. Hence, the commitreq has no great effect on data objects except for the bookkeeping.

From the specifications for the two parts, we can conclude that the concurrency control task becomes simpler, since it has been distributed. The scheme would be much more complicated if the concurrency control task were performed solely by either transaction handlers or data objects.

4.8 Conclusions

This chapter presents an effective method to solve the concurrency control problem in a distributed database system. In order to convey the idea of our scheme, we have chosen typed channels to be used in the object-oriented model. These channels represent the ordering of trans-
action executions from the viewpoint of data objects; however, messages are sent over channels to check for system level consistency without freezing normal executions of transactions. Transactions commit according to the precedence order represented by conceptual channels.

By separating data-object level consistency from system level consistency, each data object can fully act on its own and resolve most of the conflicts while the whole system can easily maintain a possible serial order of all transaction executions. THs are never blocked; they are capable of exchanging information all the time. Transactions wait only for the response to some read or write operations. Since a data object can determine whether to grant permission in a fixed amount of time, each transaction eventually will either proceed or abort.

To minimize the overhead caused by a cascade of aborts, we have presented a distributed checkpointing scheme to allow transactions to be partially rolled back. This will not increase the complexity of our scheme, since transactions will be rolled back only when inconsistency occurs, which is less likely to happen based on the assumption of an optimistic scheme.

One drawback of fully distributing the function of detecting a cycle to all the transactions is that more than one transaction involved in the
same cycle could detect the inconsistency simultaneously, thereby aborting more than one transaction. This is called the \textit{phantom cycle}, since fewer transactions need to be aborted in order to break that cycle. The phantom cycle problem can be solved by introducing priorities into the concurrency control scheme, similar to the one described in [34], to filter out unnecessary messages and to allow only one TH to detect one cycle.

If we allow a TH to store and forward only messages with smaller transaction ids than itself, then only one transaction (the one with the lowest transaction id) will detect a cycle. However this scheme cannot dynamically adjust relationships between transactions when an abortion occurs. Another reason that makes our scheme (i.e. not introduce any priorities) preferable is when fairness becomes an issue. It is not fair to arbitrarily choose either the highest or the lowest transaction id as the victim.

One way to decrease possible rollbacks is to give writes with earlier reads a higher priority over reads arriving at the same time. Normally, requests from THs are queued and served in a first-come first-serve order. If intelligent queues, in which requests could be examined and selected, are used, selecting such a write request to execute before other reads waiting in the queue will cause other transactions less likely to abort or roll back, since they are reading more current information. This
definitely requires an efficient way to distinguish whether a write is
based on the data read earlier. One possible way to make this distinc-
tion is to leave it to the THs to decide.
CHAPTER V
Variable Timestamping Scheme

5.1 Introduction

Two-phase locking and timestamp ordering are two popular concurrency control schemes in database systems. A major drawback of two-phase locking in distributed database systems is the complexity involved in deadlock detection and resolution. While the timestamp ordering scheme eliminates overhead due to deadlocks, it forces conflicting transaction accesses to execute in the order of their timestamps. In this chapter, we present a new concurrency control scheme which incorporates adjustable timestamps into two-phase locking to avoid the need for deadlock detections. This scheme also incorporates the concept of distributed control, which allows cooperation among objects. Each transaction is assigned a unique timestamp that can be adjusted during its lifetime to a certain extent to eliminate many unnecessary rollbacks while remaining deadlock free. The proposed scheme allows multiple transactions to have their timestamps adjusted concurrently without getting into deadlocks.
To show the resulting improvements, simulation results comparing the basic timestamping scheme, modified for the fairness of comparison, with our variable timestamping scheme are provided.

We define a potential deadlock as the situation in which a transaction requests to lock a data item which is already locked by another transaction with a higher timestamp. Such a kind of conflict is characterized as antagonistic in [46]. The timestamp ordering scheme aborts the transaction whose request caused an antagonistic conflict to occur, thereby avoiding any deadlock. Since the presence of an antagonistic conflict does not necessarily indicate a real deadlock situation, the preventive measure in timestamp ordering is too severe. Our scheme provides a remedy to such severe action, by attempting to adjust the timestamp of the transaction causing antagonistic conflict to accommodate the fact that a younger transaction has grasped the data item first, thereby removing the antagonistic conflict. The concurrency of transaction execution will be considerably improved because fewer transactions need to be aborted when a potential deadlock arises.

Since adjusting the timestamp of a transaction results in extra messages being exchanged, it is very important to know the improvements in performance by our scheme compared to the overhead incurred. Therefore, simulation results are included. It has been shown that the
overhead is nominal because the total number of messages exchanged depends on the degree of conflict in the system, and message exchange is needed only when a potential deadlock exists in which case the transaction handler, described later, will be idle anyway.

This chapter is organized as follows: In Section 5.2, the details regarding the use of our model for presenting the scheme are introduced. Section 5.3 presents our concurrency control scheme. Additional discussions such as shared locks and timestamp assignment are included in Sections 5.4 and 5.5, respectively. A simulation based performance evaluation is presented in Section 5.6. In Section 5.7 we compare our scheme to other related schemes. The formal specification of the variable timestamping scheme is presented and discussed in Section 5.8. Finally, Section 5.9 concludes this chapter.

5.2 System Model

The possible communication between various objects in our object-oriented model is depicted by Figure 5.1. This figure shows that communication could take place between different objects but not between the same type of objects. We shall use $DM_x$ to represent the data manager of data item $x$ in this chapter, because several data managers may be involved in exercising the concurrency control at one time.
Data items can be replicated, and some replica control must be applied to ensure the mutual consistency between copies of one logical data item. We will ignore the replica control here since it is no different than in any two-phase locking scheme except for added variable timestamps.

Figure 5.1. Communication between Objects

A transaction consists of read and write operations to data items besides computations. Since in most cases transactions are executed sequentially, each transaction is assumed to make at most one lock request at any time. Each transaction is handled by a transaction handler (TH). A TH handles transaction requests, commits, and aborts. It also acts as a coordinator to find an agreeable new timestamp among the DMs involved when antagonistic conflict occurs. Since a transaction issues at most one lock request at a time, the search for a new timestamp is
greatly simplified. If the TH cannot find a mutually agreeable times-
tamp for a transaction when antagonistic conflict occurs, it aborts the
transaction, restores the database to the original state, and releases all
acquired locks.

Therefore, DMs cooperate with THs in finding a new timestamp for
a blocked transaction, in addition to handling the locks. A compatible
lock must be acquired before a transaction can access the data item
stored; we assume two basic types of locks are implemented: sharable
and exclusive. We consider only the exclusive locks while describing our
scheme in Section 5.3; sharable locks are discussed in Section 5.4.

Transactions waiting for a lock on a data item are kept in a queue
according to their arrival time. Once a previously locked data item is
unlocked, a lock is assigned to the first transaction waiting in the queue.
Each transaction is assigned a unique timestamp, which accompanies the
request to a DM. Each DM is involved in finding a new timestamp for
either the transaction currently holding the lock (referred to as the
holder of the data item) or a transaction waiting in the queue; it will be
informed about the result of the search by the TH that is in charge of
the search, since more than one DM may be involved.
5.3 The Concurrency Control Scheme

Our concurrency control scheme follows two-phase locking, i.e., a transaction does not release any lock until it has acquired all the locks needed. To avoid the domino effect [40], we require the transaction to hold all the locks till it commits*. Each transaction is assigned a unique timestamp at the start-up time [31]; the timestamp can be adjusted during the execution in an increasing order. The phase for determining any possible new timestamp is called 'interrogation phase' (IP). An IP is initiated by the DM which detects antagonistic conflict; however, it is coordinated by the requesting TH among DMs involved. We shall refer to the DM that has initiated an IP as initiator with respect to that IP. Each DM initiates at most one IP for one of the transactions waiting for the data item with antagonistic conflict. Other waiting transactions will wait till the current IP ends in order to be processed by the DM.

We distinguish between two states of a DM. A DM is in normal state if there is no pending IP that prevents the DM from detecting any antagonistic conflict. A DM is in quiescent state if either the holder of the data item enters an IP before any lock request arrives in the queue or an IP initiated by the DM has not ended. In quiescent state, an-

*Our scheme will not be affected without this assumption.
tagonistic conflict for subsequent transactions cannot be detected immediately because the new timestamp for the preceding transaction determines whether antagonistic conflict exists or not. However, a quiescent DM can still communicate with THs. Note that a DM can be involved in an IP while remaining in normal state if an IP with respect to the holder starts after the first request, which has a higher timestamp, arrives. This is because the holder's timestamp cannot be adjusted beyond that (higher) timestamp, hence making the DM to change from a quiescent state to a normal state.

An IP involves interrogating all DMs whose data items have been locked by the blocked transaction. Each DM will reply with the range of the allowable timestamps. When the coordinating TH receives replies from all DMs involved, it determines if the intersection of allowable timestamps is non-empty. If yes, the timestamps in the intersection is agreeable to all participating DMs. The coordinating TH selects one within that range and informs all participating DMs. If no agreeable new timestamp can be found, the TH aborts the transaction, restores all states to the beginning of the transaction, and releases all the locks. Therefore, the purpose of an IP is to find whether the timestamp of the blocked transaction can be adjusted, which must be agreed to all DMs involved, to eliminate any antagonistic conflict.
Below we describe our scheme in more detail.

1) When a TH sends a request together with its current timestamp to a DM,

(a) if the requested lock can be acquired, the DM records down the timestamp as the timestamp for the holder.

(b) if the DM is in a quiescent state, the request will be put in the queue waiting for the check of any antagonistic conflict in first-come-first-serve order.

(c) if the DM is in a normal state and the requesting transaction has a smaller timestamp than the timestamp of any preceding transactions including the holder, an IP is initiated by the DM. When a DM initiates an IP, it sends a message(START_IP), supplemented with the largest timestamp of transactions waiting ahead as the lower bound, to the TH and becomes quiescent.

2) Upon receiving such a message (START_IP) from a DM, the TH does the following:

(a) It sends out a 'query' to every DM from which it has acquired the lock. Each DM is supposed to reply with an acceptable range of new timestamps.

(b) When all replies are collected, the TH selects the minimum timestamp acceptable to all participating DMs if it exists. If no mutually agreeable timestamp can be found, the transaction is aborted and restarted later. All participating DMs will be informed of the result, thus ending the IP.
3) A DM receives a query only from the TH of the current holder; upon receiving such a query, the DM does the following:

(a) If no other transaction is waiting for the data item, it simply responds with the timestamp of the holder as the lower bound on acceptable timestamps, and enters the quiescent state because no upper bound is given out at this time.

(b) If the first waiting transaction has a higher timestamp, say $T_s$, than the current timestamp of the holder, the DM replies $T_s$ as the upper bound and enters the normal state.

(c) If the first waiting transaction is involved in an IP, the DM responds only when the IP has terminated successfully. This is because only then can it determine the upper bound of acceptable timestamps. If the IP fails, the DM will process the following requests until one has a higher timestamp that can be used as the upper bound. After replying the upper bound to the inquiring TH, the DM enters the normal state.

Although it is possible that an IP cannot complete until another IP completes, there will be no deadlock because each transaction can only wait for one data item, and an IP is pending on the outcome of another IP only if the latter IP is initiated for a transaction with a smaller timestamp than the transaction for which the former IP is initiated; therefore, deadlock is not possible. Note that every new times-
Example I

Let's consider 3 data items x, y, z and 4 transactions T₁, T₂, T₃, T₄ as shown in Figure 5.2. Let TS(Tᵢ) represent the current timestamp of transaction Tᵢ. Initially TS(T₁)=10, TS(T₂)=20, TS(T₃)=30, TS(T₄)=40, assuming the time interval is 1. Assume that x is locked by T₃, both y and z are locked by T₁, and T₄ is waiting for z. The following is one of the possible executions.

1) When the lock request for x from T₁ arrives at DMₓ, DMₓ initiates an IP for T₁ and sends 30 as the lower bound for the new timestamp to TH₁. If T₂ arrives at DMₓ before the IP ends, the check on T₂ cannot proceed until the IP for T₁ terminates.
2) \( \text{TH}_1 \) sends a query to both \( \text{DM}_y \) and \( \text{DM}_z \). Since no transaction is waiting for \( y \), \( \text{DM}_y \) responds with 10 as the lower bound for the new timestamp and enters the quiescent state. On the other hand, since the first transaction waiting for \( z \) has a higher timestamp, it returns 40 as the upper bound for the new timestamp to \( \text{TH}_1 \) and remains in the normal state because no matter what turns out for \( T_1 \), subsequent transactions will not be affected.

3) When \( \text{TH}_1 \) receives all replies, it selects the smallest timestamp between 30 and 40, which is 31.

4) When \( \text{DM}_x \) receives the new timestamp for \( T_1 \), it ends the IP and enters the normal state again. When it sees that \( T_2 \) has a smaller timestamp than \( T_1 \), it will initiate another IP for \( T_2 \). When \( \text{DM}_y \) receives 31 for \( T_1 \), it returns to the normal state and proceeds to process incoming requests. \( \text{DM}_z \) will simply update the \( \text{TS}(\text{holder}) \) to 31.

5.4 The Scheme with Shared and Exclusive Locks

When shared and exclusive locks are considered, the scheme only needs to be more specific regarding the type of locks requested. Since more than one transaction can acquire a shared lock on the data item at the same time, the ordering between these transactions is no longer significant. Therefore, when a blocked transaction is requesting a shared lock, an IP will be initiated only when its timestamp is smaller than the previous transaction requesting an exclusive lock. When a blocked trans-
action is requesting an exclusive lock, its timestamp must be greater than all the transactions waiting ahead of it; otherwise, an IP is initiated. This is because we allow transactions requesting shared locks to wait in the queue in any order.

Furthermore, when a DM replies to the query concerning the holder of an exclusive lock with the upper bound of acceptable timestamps, it must enter the quiescent state. This is because the upper bound is determined by the waiting transactions at that time and new requests may arrive afterwards. This also means subsequent requests with timestamps smaller than the one given out cannot be processed until the new timestamp for the holder is determined. This is explained in Example II below. To avoid any starvation, if a holder has a shared lock on the data item, subsequent requests for the shared locks can be accepted only if there is no transaction waiting for an exclusive lock.

Example II Let's consider two data items x and y and 5 transactions $T_1$, $T_2$, $T_3$, $T_4$ and $T_5$ with initial timestamps 10, 20, 30, 40, 50, respectively. Assume $T_4$ has a shared lock on x and $T_2$ has an exclusive lock on y.

1) When the request for a shared lock on x from $T_1$ arrives, $T_1$ gets a shared lock on x. When $T_5$ requests for a shared lock on y, it is put in the queue and no IP is initiated.
2) When $T_2$ requests an exclusive lock on $x$, an IP is initiated by $DM_x$ since its timestamp (20) is smaller than one of the holders ($T_4$). $DM_x$ sends a message with 40 as the lower bound to $TH_2$.

3) Upon receiving such a message from $DM_x$, $TH_2$ sends a query to $DM_y$ and $DM_y$ replies with 50 as the upper bound and becomes quiescent. If $T_3$ requests a shared lock on $y$ when $DM_y$ is quiescent due to the IP for $T_2$, the check on $T_3$ cannot be performed until either $T_2$ gets a new timestamp or aborts.

4) $TH_2$ selects 41 as the new timestamp for $T_2$, informs all participating DMs, and ends the current IP.

5) When $DM_y$ receives the new timestamp for $T_2$, it updates its record and continues to process subsequent requests. At this point, $DM_y$ initiates an IP for $T_3$ since its timestamp is smaller than that of $T_2$. 
5.5 Related Issues

In this section we discuss some issues related to implementation details.

5.5.1 Timestamp Assignment

When a TH selects a new timestamp for a blocked transaction, it is important not to select one that has already been assigned to another transaction. Because transaction startup time is no longer used to assign timestamps, a new timestamp may collide with other existing transactions. Fortunately this only happens to transactions issued at the same site if the pair \( t,i \) is used for a timestamp. A simple but less efficient scheme might require the TH to check for all outstanding timestamps to avoid collision. This scheme, however, implies that the assignment of a new timestamp must be atomic.

Another potential problem is that it might not be always possible to find an available timestamp using a discrete timestamp generator. In the case of simulating the time instants as a sequence of increasing numbers, there will be no room for a transaction to adjust its timestamp unless there is some appropriate ‘slot’ open. A new scheme based on the one reported in [31] to solve the problem is proposed as follows.
The basic idea is to divide each time instant into N slots, where N is the maximum number of outstanding timestamps at any time at one site. Since there cannot be more than N transactions active, each transaction can have one particular slot reserved in every time instant. Each site maintains a list of available slots. To prevent the same timestamp from being assigned to more than one transaction at the same site, our timestamp has one more field than the one proposed by Lamport [31].

Timestamp $T_1 = <t_1, n_1, i_1>$ is said to be smaller than timestamp $T_2 = <t_2, n_2, i_2>$ if

1. $t_1 < t_2$
2. $t_1 = t_2$ and $i_1 < i_2$, or
3. $t_1 = t_2$ and $i_1 = i_2$ and $n_1 < n_2$

where $t$, $n$, $i$ represent time instant, slot number and site id, respectively.

Note that the slot number is significant only when comparing two timestamps with the same time instant and the same site id.

5.5.2 Timestamp Selection

Whenever a transaction's timestamp needs to be adjusted, the range of acceptable timestamps must be determined. Selecting the smallest timestamp within that range has the advantage of affecting fewer succeeding transactions. On the other hand, selecting the largest timestamp within that range will leave more room for preceding transactions.
to adjust their timestamps. The middle point may be a good compromise. A better alternative is to allow THs to use an adaptive policy which selects a smaller timestamp if transactions are short and their arrival rate is high, and a bigger one otherwise.

5.5.3 Determining the Acceptable Range for a New Timestamp

When a transaction's timestamp needs to be adjusted, the minimum acceptable timestamp can be determined immediately, which is not the case for determining the maximum acceptable timestamp. This may be seen as follows: First, when a data item, whose holder is involved in an IP, has no other requests waiting in the queue, it cannot predict the timestamp of the next incoming request. Therefore, if it gives no upper limit on the new timestamp, it has to block itself from handling incoming requests until the IP ends. Second, even when there is at least one request in the queue whose timestamp can be used to determine the maximum acceptable timestamp as described in our scheme, it does not mean that all waiting transactions cannot be adjusted in order to accommodate a bigger range for the transaction involved in an IP. For example, consider data items x,y and transactions T_1,T_2,T_3 as depicted in Figure 5.4. T_2 needs a new timestamp larger than T_1's. Since T_3 is the only one waiting for T_2 and T_3 has not acquired any locks yet, it would be nice to have T_3 adjusted as well in order to accommodate T_2.
If we allow a DM to initiate IPs for the transactions waiting in the queue to maximize the possibility of having the holder's timestamp adjusted successfully, our scheme will be similar to two-phase locking. This is because that having no cycle in the wait-for graph in two-phase locking implies that all the timestamps in timestamp ordering can be reassigned without antagonistic conflicts. However, the autonomy feature in timestamp scheme is sacrificed. We favor the scheme we have proposed because less time will be consumed and fewer DMs as well as THs will be involved.
5.5.4 Minimizing the Need for IP

To preserve the first-come-first-served feature of two-phase locking, transactions waiting in a queue are processed according to their arrival time. However, this restriction can be relaxed by allowing a DM to assign lock(s) according to their timestamp ordering. This minimizes the need for IPs since requests are sort of 'in order'. This, however, requires extra software or even hardware to allow DMs to scan through all requests pending in the queues.

5.5.5 Minimizing Communication Overhead

The number of queries sent for one IP can be kept to a minimum since the purpose of sending a query to a DM is to find out the range in which the timestamp could be adjusted, and the range is determined by the transactions waiting in the queue. If we allow the TH to record down the upper bounds of acceptable timestamps replied by DMs during an earlier IP, no query to these DMs is needed during subsequent IPs since the upper bounds set by these DMs cannot be changed unless the current transaction releases the locks. Special care must be taken when shared and exclusive locks are considered; the upper bound as determined by a DM earlier might not be valid anymore if the holder has an exclusive lock and a transaction waiting for a shared lock, with a times-
tamp smaller than all other transactions waiting in the same situation, has arrived since then. When an IP ends, all participating DMs must be informed no matter whether they received a query or not.

5.6 Performance Evaluation

It is important to evaluate the performance of the proposed scheme. Since analytical evaluation is very difficult for such a complicated scheme, our scheme is evaluated and compared to a timestamp ordering scheme which does not allow variable timestamps using simulations. The basic timestamp ordering scheme may result in a sequence of aborts due to the domino effect; therefore, it is modified by integrating locks so that subsequent transactions cannot observe premature data items until locks on them are released.

The biggest difference between the two schemes is that one tends to adjust the timestamp of a requesting transaction whenever its timestamp violates the rule of the timestamp ordering while the other simply aborts and restarts it later. Since adjusting the timestamp for a transaction incurs extra communication overhead, we have simulated various message delay time against various degrees of conflicts. The following describes our simulation model, parameters of the model and simulation results. A more detailed discussion on the performance evaluation can be found in [45].
5.6.1 Simulation Model

The primary objective of the simulation model is to evaluate our scheme (with adjustable timestamps) against timestamp-based schemes (without adjustable timestamps). The model is used to study the relative performance of these two types of timestamping scheme, not to predict their exact performance in a real-life database system. Therefore, we make some simplifying assumptions regarding the arrival rate and service rate of transactions. In our simulation model, an object-oriented distributed database system consists of M processing sites (where transactions are executed) and N data sites (where data are located). Processing sites and data sites are treated as independent sites. This has simplified the need for investigating the file allocation strategy to determine whether a data access is local or remote.

Our simulation model is written in SIMPAS [10]. Transactions arrive at various sites according to Poisson distribution as justified in [42]. The arrival of a transaction at a processing site is simulated by a Poisson process and a job queue. The creation of a new transaction involves determining certain characteristics of that transaction for the purpose of simulating restarts. The number of transactions that are run concurrently at any site is one of the parameters in our model as well as the number of sites simulated. Transactions waiting in the job queue will be served
in first-come-first-serve order. When a transaction aborts, it is put back at the end of the job queue waiting for restart. The access pattern of a transaction is preserved when it is restarted.

One of the characteristics of transactions is their access pattern. Since a major factor affecting the performance of these two schemes is the degree of conflict, which depends on the transaction access pattern, i.e. the more data items the transactions access, the higher possibility they will conflict. Therefore, instead of defining transaction's size to be the needed length of execution, we use REQPER, the probability that a transaction will access another data item, as one of the transaction characteristics in our simulation. This is different from predetermining transaction sizes as the percentage of small update transactions and large read-only transactions as in [14]. In other words, after each access, the probability that this transaction will make another request is equal to REQPER. Thus, a higher REQPER will cause the transactions to conflict more likely. The total number of requests a transaction can make is limited by the maximum number of data items simulated.

Since the data access pattern in a database varies with application and time, for simplicity, we assume the data access pattern to be uniformly distributed for the entire system, i.e., every data item is accessed with equal probability. When one transaction issues a data access
request, a random generator is used to model the selection of a data item. However, in our model, we make sure that any data item will not be selected by the same transaction more than once. This is under the assumption of only exclusive locks are implemented, of course. When a transaction issues a request and selects a data item, it is put into the waiting queue of that data item. Since each transaction makes at most one lock request at any time, a transaction can be waiting in at most one queue.

To compare the two schemes, we have implemented two simulators with most of the parameters identical except the concurrency control rules. The complete program listings for the two simulators are included in Appendices A and B respectively. The DM in one simulator checks the timestamp of a requesting transaction using a concurrency control scheme that is different than the one used in another simulator and the results are compared. In the simulator for variable timestamping (VT), a DM might enter the IP while waiting for the information regarding the new timestamp to arrive. It has been shown that the blocking, if any, is only temporary, and is reflected in the simulation results. In the simulator for the timestamping with locks (TSLK), a DM rejects a request if the timestamp order is violated without consulting other sites; no extra communication is involved as required by our scheme.
Since we do not distinguish local from remote access, the service time of the communication medium is assumed to be constant. That is, the message delay, defined as the time it takes by a message to travel from one site to another site is constant. However, during various simulation runs, we test different length of communication delay to study the effect of communication overhead on the performance of the database system.

Several measures are used in our performance evaluation. The total number of transactions executed including restarts, total completions (number of transaction that completed), total aborts (number of aborts occurred), response time (the time between a transaction first starts and the time it finally commits, which includes the time waiting for locks and restarts) and system throughput. The system throughput is not determined solely by the number of transactions completed because the overhead caused by aborts should be considered as well. Therefore, our notion of performance is the ratio of the number of completed transactions to the number of transactions executed.
5.6.2 Statistics Collection

All performance measures were collected as simulation results by running two simulators with various message delay and REQPER given. Message delay varies from 0.05 to 0.5 and REQPER varies from 0.1 to 0.9, both measured in units of simulation time. Statistics were collected after both simulators reached a steady state. Usually a steady state was reached when both simulators ran for a period of time, measured in units of simulation time, long enough to show the characteristics of two different schemes.

5.6.3 Simulation Results

Tables 5.1 and 5.2 show the simulation results gathered from the variable timestamping (VT) scheme and timestamping with locks (TSLK) scheme with 0.1 unit of simulation time message delay and 0.1 to 0.9 REQPER values. As shown in Tables 5.1 and 5.2, as REQPER increases, not only fewer transactions were started, there was a significant drop in system throughput. This means more transactions ended in abort. Figures 5.5, 5.6 and 5.7 show the overall system throughput of the two schemes with various message delays.

Since TSLK does not incur any processing overhead to adjust transaction timestamps, more transactions will be started under TSLK
### Table 5.1. Variable Timestamping (VT)

<table>
<thead>
<tr>
<th>REORDER</th>
<th>Total</th>
<th>Completion</th>
<th>Abort</th>
<th>Throughput</th>
<th>Response time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>659</td>
<td>573</td>
<td>37</td>
<td>86.95</td>
<td>3.061</td>
</tr>
<tr>
<td>0.2</td>
<td>555</td>
<td>454</td>
<td>52</td>
<td>81.80</td>
<td>3.591</td>
</tr>
<tr>
<td>0.3</td>
<td>448</td>
<td>323</td>
<td>76</td>
<td>72.10</td>
<td>4.665</td>
</tr>
<tr>
<td>0.4</td>
<td>401</td>
<td>251</td>
<td>101</td>
<td>62.59</td>
<td>4.898</td>
</tr>
<tr>
<td>0.5</td>
<td>295</td>
<td>146</td>
<td>100</td>
<td>49.49</td>
<td>5.412</td>
</tr>
<tr>
<td>0.6</td>
<td>327</td>
<td>147</td>
<td>131</td>
<td>44.95</td>
<td>5.783</td>
</tr>
<tr>
<td>0.7</td>
<td>314</td>
<td>100</td>
<td>165</td>
<td>31.85</td>
<td>5.717</td>
</tr>
<tr>
<td>0.8</td>
<td>297</td>
<td>65</td>
<td>183</td>
<td>21.89</td>
<td>6.885</td>
</tr>
<tr>
<td>0.9</td>
<td>318</td>
<td>43</td>
<td>226</td>
<td>13.52</td>
<td>6.942</td>
</tr>
</tbody>
</table>

### Table 5.2. Timestamping with Locks (TSLK)

<table>
<thead>
<tr>
<th>REORDER</th>
<th>Total</th>
<th>Completion</th>
<th>Abort</th>
<th>Throughput</th>
<th>Response time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>1084</td>
<td>575</td>
<td>460</td>
<td>53.04</td>
<td>4.371</td>
</tr>
<tr>
<td>0.2</td>
<td>1028</td>
<td>488</td>
<td>491</td>
<td>47.47</td>
<td>4.717</td>
</tr>
<tr>
<td>0.3</td>
<td>888</td>
<td>392</td>
<td>447</td>
<td>44.14</td>
<td>4.797</td>
</tr>
<tr>
<td>0.4</td>
<td>860</td>
<td>302</td>
<td>509</td>
<td>35.12</td>
<td>4.819</td>
</tr>
<tr>
<td>0.5</td>
<td>744</td>
<td>224</td>
<td>471</td>
<td>30.11</td>
<td>5.265</td>
</tr>
<tr>
<td>0.6</td>
<td>764</td>
<td>197</td>
<td>518</td>
<td>25.79</td>
<td>5.132</td>
</tr>
<tr>
<td>0.7</td>
<td>735</td>
<td>144</td>
<td>542</td>
<td>19.59</td>
<td>5.870</td>
</tr>
<tr>
<td>0.8</td>
<td>620</td>
<td>73</td>
<td>498</td>
<td>11.77</td>
<td>5.569</td>
</tr>
<tr>
<td>0.9</td>
<td>667</td>
<td>49</td>
<td>569</td>
<td>7.346</td>
<td>5.768</td>
</tr>
</tbody>
</table>
than VT. However, this also contributes to a greater number of aborts. The VT scheme outperforms the TSLK scheme especially when the degree of conflict is low. This is because a transaction’s timestamp can be adjusted more easily. Further, it is unlikely that it needs to be adjusted again when the transactions are less likely to interfere with each others. Thus many unnecessary aborts and restarts are avoided. From Figures 5.5, 5.6 and 5.7 we see that message delay is not a major factor affecting the system throughput. Longer communication delay simply means it takes longer time for a transaction to finish and this affects every transaction more or less. It is the transaction access pattern that dominates the system throughput.

The difference between the overall response time of the two schemes is not significant because the extra time needed by our scheme to reassign a new timestamp to a transaction is leveled off by the abort-restart time wasted in TSLK. This can be seen from Figure 5.8. Our scheme has smaller average response time than TSLK when the degree of conflict is low because the extra communication overhead is less than the overhead incurred by the aborts in TSLK. However, as the degree of conflict becomes higher, a transaction might still end in abort after several attempts to adjust its timestamp. That is why our scheme has higher response time than TSLK when REQPER is high. The reason that the overall response time does not increase noticeably as the trans-
Figure 5.5. System Throughput vs REQPER
With Message Delay = 0.1
Figure 5.6. System Throughput vs REQPER
With Message Delay = 0.05
Figure 5.7. System Throughput vs REQPER
With Message Delay = 0.5
action size increases is that only the completed transactions were accounted. Because both schemes do not impose any policy to handle large transactions that are repeatedly aborted, the time wasted on these transactions were not included due to the fact that they were unable to finish within a certain limited period of time.

Figure 5.9 compares the average response time against various REQPER for various message delay times under VT. Figure 5.10 does the same under TSLK. When communication delay increases, the response time also increases. The way that message delay affects the response time under TSLK scheme is not as significant as that in our scheme because our scheme requires more message exchanges. However, unless the message delay is as big as 0.5 unit of the simulation time under VT, the difference is not significant due to the fact that a lot of communication is eliminated by storing the ranges of acceptable timestamps collected for future references. The resource time saved by our scheme can well justify the extra communication incurred. Figure 5.11 shows the difference in average response time between the two schemes as message delay increases for REQPER equal to 0.1 and 0.5. Once again it shows that communication delay is an important factor in determining the system performance. However, unless the communication cost is extremely large as compared to the time to execute one transaction, our scheme seems more prominent than TSLK.
Figure 5.8. Average Response Time vs REQPER
With Message Delay = 0.1
Figure 5.9. Average Response Time vs REQPER under VT
With Various Message Delay
Figure 5.10. Response Time vs REQPER under TSLK with Various Message Delay
Figure 5.11. Average Response Time vs Message Delay
5.7 Comparison to Other Schemes

The two-phase locking scheme does not impose any a priori fixed order of execution (like timestamp ordering does). By setting appropriate locks on data items, interference between two conflicting transactions is eliminated and a serial order, determined by the lockpoints, is also maintained on these transactions. However, deadlock could occur due to a cycle in the wait-for graph. This greatly reduces the performance of a two-phase locking scheme since deadlock detection and resolution in distributed database systems incurs extra overhead. Our scheme preserves features of a locking scheme while eliminating deadlocks with little extra overhead by detecting and resolving potential deadlocks.

When compared to the timestamp ordering, our scheme has fewer aborts since it does not force transactions to execute in the order of initial timestamps.

As compared to deadlock detection schemes such as the one proposed in [34], our scheme does not have to use messages to check every node involved in the wait-for graph. Furthermore, according to the scheme proposed in [34], only the node with the smallest transaction_id will detect the deadlock. In order to make it work, extra storage is required at each node to store messages received with a smaller transaction_id. Our scheme has incorporated the deadlock avoidance idea
into the timestamp ordering so individual sites can immediately
determine any possible deadlock without extra storage nor waiting for
messages traveling through the edges in the wait-for graph to return.

That potential deadlock in our scheme does not necessarily imply
real deadlock; this gives us a concurrency control scheme which responds
more promptly. To guarantee that a transaction is aborted only when
there is indeed a real deadlock, our scheme can be modified so that an
IP involves in adjusting the timestamps in any possible way, including
adjusting timestamps of subsequent transactions. In this case, a real
deadlock exists when no new timestamp can be found for the blocked
transaction. However, this involves recursive initiations of IPs just like
probes propagating along the edges in the wait-for graph, and special
care must be taken on cyclic IPs when there is a cycle in the graph.

5.8 The Formal Specification

In this section, the variable timestamping scheme is presented using
the high level specification with necessary discussions. As in Chapter 4,
two kinds of objects are considered: the transaction handler and the data
object. Names are assigned to these objects in the specification for easier
identification.
The variable timestamping scheme is based on the timestamping scheme. Transactions are assigned timestamps and those timestamps are used to sequence transaction access to the data objects. However, what makes our scheme different from other timestamp-based schemes is that when a transaction's timestamp no longer preserves the serializable ordering among data access, an effort is made by both the data objects and the transaction handler to adjust the timestamp. This can be shown by the *interrogate* and *inquiry* messages exchanged between data objects and transaction handlers in the specifications depicted in Figures 5.12 and 5.13.

In addition to the common requests, a data object issues an Interrogation Phase (IP) by sending the *interrogate* message to the transaction handler. The transaction handler in response sends necessary *inquiry* messages* to the data objects it has visited to find a possible new timestamp for the transaction. When replies from all the data objects that were sent the *inquiry* have arrived, the transaction handler can immediately determine whether its transaction can be assigned a new timestamp or not. In case a new timestamp can be found, a *newts* message will be sent to each of the data objects visited to inform the new timestamp; otherwise, the failing transaction handler will abort all the

*The number of inquiries sent can be minimized as mentioned earlier.
Object transaction_handler Y;

Concurrency control dblock:
[ Array data_lists is used to record the range of timestamps from various dataobjects]

data_lists = array [1..maxint] of (dobjectid,low, high);
pr_data : integer, initially 0;
trans: integer, initialized at start up time;
count: integer;
min, max: integer;

Message type declaration:
[ input message types ]
interrogate(tranid, low, high, dobjectid);
accessreq(dobj);
finished();

[output message types ]
inquiry(tranid, dobjectid);
abortreq(tranid, dobjectid);
commitreq(tranid, dobjectid);
news(tranid, trans, dobjectid);
locrreq(tranid, trans, dobjectid);

Processing:

* [accessreq(dobjectid)? —> locrreq(Y, tranid, dobjectid)];
pr_data:=pr_data+1;
  data_lists[pr_data].dobjectid:=dobjectid;
  data_lists[pr_data].low:=low;
  data_lists[pr_data].high:=high;

* finished() ? —> i:=1; *[i<=pr_data —> commitreq(Y, data_lists[i].dobjectid)]; i:=i+1]

* interrogate(Y, low, high, dobjectid)? —> min := low; count := 0; i := 1;
  *[1 <= pr_data —> (data_lists[i].high = 1 v data_lists[i].high = 0)
    & data_lists[i].dobjectid = dobjectid —> count := count + 1;
    inquiry(Y, data_lists[i].dobjectid)]; i := i + 1
  ]

* [count > 0, interrogate(Y, low, high, dobjectid)? —> count := count - 1; i := 1;
  *[1 <= pr_data —> (data_lists[i].dobjectid = dobjectid —> data_lists[i].high := high); i := i + 1
  ]

* [high < max —> max := high]

* [max > min —> choose(min, max, i); trans := i; i := 1;
  *[1 <= pr_data —> news(Y, tranid, data_lists[i].dobjectid)]; i := i + 1
  ]

* [max <= min —> i := 1; *[i <= pr_data —> abortreq(Y, data_lists[i].dobjectid)]; i := i + 1
  ]

Figure 5.12. Specification of the Transaction Handler Object
activities associated with data objects by sending the abortreq message to those data objects. The actual procedure (Choose) to select the new timestamp is not elaborated here since it is more implementation related.

Since a data object cannot proceed to validate the next request unless the current request has been validated, state variable freeze is used to keep the data object from continuing to process the next request. State variable ip is used to indicate whether the freeze situation is caused by the initiation of an interrogation phase or is simply caused by the attempt to adjust the timestamp of the transaction that currently holds the lock. Consequently, a data object can proceed to validate other requests when either the interrogation phase or the adjustment of the timestamp of the transaction that currently holds the lock is over.

This scheme was developed by incorporating the concept of distributed control, which allows cooperation among objects. This scheme has been shown to have better performance than the original timestamping ordering scheme. The specification method has helped in expressing the main concept of our scheme in a precise fashion without any ambiguity or confusion. Since the scheme is presented in a high level specification, details involving the actual programming have been hidden away while the structure of a high level programming language which helps the explaining of the scheme remained. To explain a new scheme
Object data_object X;

Concurrency control datalock:
| The entries of AccessTable have three fields, |
| • tranid, current timestamp and status. |
| • status = (committed, active, validated, waiting) |
| • freeze and ip are two state variables |

Accessible = (tranid.tranid.status);

Concurrence control invariant:
active.tranid > prev(active.tranid).tranid

Message type declaration:
| input message types |
| inquiry(tranid, dobjectid);
| lockreq(tranid, tranid, dobjectid);
| abortreq(tranid, dobjectid);
| commitreq(tranid, dobjectid);
| newts(tranid, tranid, dobjectid);

| output message types |
| interrogate(tranid,low,high,dobjectid);

Processing:

• lockreq(tranid, tranid, X) ? —> enter(tranid.tranid, 'waiting');

• commitreq(tranid, X) ? —> assignlock(); modify(tranid; 'committed');

• freeze A (waiting() = 0) —> succ(validated().tranid).tranid > validated().tranid —>
  modify(succ(validated().tranid).tranid; 'validated');

• succ(validated().tranid).tranid < validated().tranid —>
  freeze := true; ip := true;

  • interrogate(tranid, validated().tranid, = , X);

• inquiry(tranid, X) ? —> [ validated() = 0 —> interrogate(tranid,0,succ(tranid; 'validated').tranid,X)];

• validated() = 0 —> [ ip —> wakeup:=true;
  • —> interrogate(tranid,0,X);freeze:=false ]

• newts(tranid, tranid, X)? —> [ any(tranid).status = 'active' —> modify(tranid;tranid);

  —> ip —> freeze:=false]

• newts(tranid, tranid, X)? —> [ any(tranid).status = 'waiting' —> modify(tranid; 'validated'); ip:=false;

  • freeze := false;

  • [ wakeup := interrogate(active.tranid,0,

    succ(active.tranid).tranid,X);]

  • wakeup := false

Figure 5.13. Specification of the Data Object
[] abortreq(tranid,X)? —> [ any(tranid).status = 'active' —> rollback(); assignlock();
   remove(tranid);
   [ip —> freeze := false]
   ] any(tranid).status = 'waiting' —> ip := false; remove(tranid);
   [wakeup —> interrogate(active.tranid,0,=X)!]
   [— wakeup —> freeze := false
   ]

Other issues:
Assignlock(): succ(active.tranid).status = 'validated' —> modify(succ(active.tranid).tranid,'active')

Figure 5.13 continued
without using such specification methods, the cases of how aborts, requests and other types of communications should be handled will have to be introduced separately. Even by using examples that connect various parts together, often readers are left confused without a clear picture about the scheme let alone finding its significant aspects.

5.9 Conclusions

We have presented a concurrency control scheme which combines the merits of the two most commonly used concurrency control schemes, viz., two-phase locking and timestamp ordering. With the feature from the two-phase locking scheme, the transactions are executed in a natural, serializable order. And with variable timestamps, an IP is initiated only when there is a antagonistic conflict. The IP not only detects any inconsistency but also automatically adjusts the serializable order when inconsistency can be resolved.

Another advantage of our scheme is that only a subset of DMs are involved in detecting antagonistic conflict rather than the entire system (as in constructing the wait-graph for the entire system). This implies that more than one subgroup can undergo the detection phase and resolve antagonistic conflict at the same time, thereby greatly increasing the fault tolerance of the system.
We have also discussed the case in which both shared and exclusive locks are allowed; in this case, only slight modification needs be made to determine the range for a new timestamp. Related issues on selecting and assigning timestamps for our scheme have been discussed. Our scheme has been compared to two-phase locking and timestamp ordering, and shown to have preserved the advantages of both. From the simulation results, we conclude that our scheme greatly improves the system performance without the threat of deadlocks.
CHAPTER VI
Conclusions

6.1 Summary

The research in 'concurrency control' in database systems has always been an interesting subject in addition to its importance. Because the characteristics of a database system varies with its application, the concurrency control scheme designed for one database system differs from those for other systems with different characteristics. Hence, a large number of concurrency control schemes have been developed and their performances evaluated. However, little research has been done on the design philosophy of concurrency control schemes and their formalism. Our research was aimed at both the design philosophy and the formal specification of concurrency control schemes.

Despite all the concurrency control schemes that have been developed, there will always be more in the future due to the rapid advance in database technologies. The design philosophy should be applicable to different concurrency control schemes developed in the future.
Therefore, the design philosophy must be studied in order to guide the design of new concurrency control schemes. The methodology presented in this work was based on distributed database systems with inherent parallel processing. Because of this, we studied schemes based on both pessimism and optimism, and demonstrated the effect of distributed control on concurrency control schemes.

Chapter 2 contains the background material on concurrency control in database systems. Problems were identified and existing solutions discussed. This chapter also laid the foundation for our research since our interest was stimulated by studying various solutions to the concurrency control problems. To present our idea on the design of concurrency control schemes, we discussed schemes in two general categories. The benefits from selecting an object-oriented design model and a CSP-based specification method were also discussed.

Chapter 3 introduces the representation model of our specification. The constructs of the high level specification were explained and examples were chosen from three well-known concurrency control schemes. From the examples provided in this chapter and the specification of the two proposed schemes in Chapters 4 and 5, we have demonstrated the use of this specification. Since better concurrency control schemes can be designed with the understanding of some existing schemes, presenting ex-
isting schemes in a formal way makes it easier to understand those schemes without ambiguity. Therefore, a uniform model and a formal way of specifying concurrency control schemes are as important as the schemes themselves.

Chapter 4 contains a new concurrency control scheme resulting from incorporating the concept of distributed control into optimistic schemes. The scheme presented was optimistic because it allows many possible sequences of concurrently executing transactions to occur; however, only sequences preserving serializability will be finally accepted. The distribution of concurrency control is applied so that the task of seeking a serializable order of transaction executions is fully distributed to both transaction handlers and data objects. This is different from the design of most existing schemes, in which the control is centralized and handled either by the transaction handlers or the data managers.

Chapter 5 contains a new concurrency control scheme resulting from incorporating the concept of distributed control into pessimistic schemes. The scheme presented was pessimistic because it prevents any possibility of inconsistency from occurring by eliminating transaction executions that might, although not necessarily cause inconsistency. The concept of distributed control was applied so that more information regarding an execution, which might lead to inconsistency, can be
gathered through the communication network for the components to make better judgment on that execution. In order to support our findings, we reported simulation results comparing our scheme with another scheme. From the simulation results, our design philosophy, which encourages the distribution of concurrency control among various objects, seems very promising.

6.2 Concluding Remarks and Future Work

To cope with the fast evolution of database technologies, the idea of software engineering must be introduced. We have argued that distributed control with cooperation among the different components to be the main ideas in the design of concurrency control schemes because the future of database systems belongs to distributed systems. In designing concurrency control schemes for distributed systems, we must try to fully utilize the system to achieve maximum performance. In order to present concurrency control schemes for the purpose of understanding and comparison, methods for formal specification must be developed.

While we have developed a methodology for the design of concurrency control schemes in distributed database systems, further efforts are needed in the same direction as indicated by this work. They are described as follows.
• Hierarchy of concurrency control scheme specification:

As mentioned in this work that our specification serves as the top layer of a hierarchy of concurrency control schemes representation. Each layer should provide more detail than an upper layer and more abstraction than a lower layer. For example, the top layer assumes some rollback procedure to resolve inconsistency while the lower layer has to specify how rollback should be performed. However, the interface between two adjacent layers has not been defined. This requires functions of all layers to be well defined first.

• Measures for deriving the actual program from its specification:

Although schemes specified using the high level specification cannot be compared on a fair basis due to the abstraction in the presentation, a set of measures will be needed in deriving the actual program from its specification. Since the specification is different from the simulation and is quite far from the actual implementation, measures for the purpose different from comparison need to be defined. Measures such as types of messages handled, number of messages generated, the possibility of invoking a rollback subroutine, and the amount of processing needed for each request are just some examples. Based on these measures, one can argue an implementation to be better than another one.

• Verification and validation methods:

Although there are many works on program verification and validation, studies in applying various methods to verify or validate concurrency control programs are needed in order to compare different methods and to find if any modification is needed. Concurrency control programs must be proved to be free of deadlock, which includes communication deadlock as well, and it must be proved to preserve
system consistency. This will eliminate possible errors in the design before being put into implementation.

- Integrating recovery control:
  Before recovery control schemes can be formally specified, they cannot be integrated with concurrency control specification. Although many recovery control schemes were introduced informally, formal specification can still be achieved by starting with the definition of common operations in the context of recovery control. For example, an 'undo' operation that replaces a new value with an old value and a 'redo' operation that does the opposite. Recovery from crashes can be specified as receiving a message from the database manager and then performing necessary operations according to the recovery control database, while recovery from rollback can be specified as receiving a message from the concurrency control scheduler.
APPENDIX A

Simulation Program for the Variable Timestamping Scheme

The complete simulation program for the variable timestamping scheme is listed here. REQPER and message delay time are two input parameters to this simulator. Other simulation parameters are made constants for the ease of modification.

```plaintext
(* 1*) program cc_vt(input, output);
(* 2*)
const
{ version 3.5 simpas output }
noseed = 100;
(* 2*)
(* 3*) freeseed = 50;
(* 4*) cdelay = 0.01;
(* 5*) nosite = 10;
(* 6*) noob = 20;
(* 7*) maxtranno = 5;
(* 8*) maxtotaltran = 30;
(* 9*) normalterm = 1;
(* 10*) servicerate = 0.6;
(* 11*) arrivalrate = 0.25;
(* 12*) tsinterval = 20;
(* 13*)
type
t0ev01 = (no0event,
begincheck,
```
146
```
generatran,
create,
endcheck,
dequeue,
departure,
arrival,
main);
ptrOtran = tran;
statOtype = (noOstat, tally, accumulate, accum, interval);
ptrOevent = eventOnotice;
statistic = record
  nobs : integer;
  max, min : real;
  case stype : statOtype of
    tally,
    accumulate,
    accum :
      (mean, variance : real;
       total, mk, tk : real;
       tostart, tolast : real);
    interval :
      (asum, ysum : real;
       asum2, ysum2 : real;
       sumprod : real);
end;
(* 13*)
(* 14*) checkstatus = ( normal, wait, freeze );
(* 15*) tran = record
  (* 15*) next, prev, qhead: ptrOtran ;
  (* 15*) inqueue: boolean;
  (* 15*)
  (* 16*) tag : integer;
  (* 17*) stime : real ;
  (* 18*) trid : integer;
  (* 19*) slotid : integer;
  (* 20*) siteid : integer;
  (* 21*) checked : boolean;
  (* 22*) lockobtained : array [1..noob] of boolean;
  (* 23*) nextts : array[1..noob] of integer;
  (* 24*)
  (* 25*) nowaits : integer;
\( (* 26*) \) \texttt{sd1 : integer;}

\( (* 27*) \) \texttt{sd2 : integer;}

\( (* 28*) \) \texttt{sd01, sd02 : integer;}

\( (* 29*) \) \texttt{end;}

\( (* 30*) \) \texttt{tranqueue = record}

\( (* 30*) \) \texttt{head: ptrOtran ;}

\( (* 30*) \) \texttt{size: integer;}

\( (* 30*) \) \texttt{empty: boolean;}

\( (* 30*) \) \texttt{stat: statistic;}

\( (* 30*) \) \texttt{end}

\( (* 30*) \) \texttt{;}

\( (* 31*) \)

\( t0\text{begincheck} = record \)

\( t : integer ; \)

\( end ; \)

\( t0\text{generatrans} = record \)

\( s : integer ; \)

\( end ; \)

\( t0\text{create} = record \)

\( s : integer ; \)

\( end ; \)

\( t0\text{endcheck} = record \)

\( t : integer ; \)

\( end ; \)

\( t0\text{dequeue} = record \)

\( t : integer ; \)

\( end ; \)

\( t0\text{deparature} = record \)

\( \text{request: ptrOtran ;} \)

\( \text{ss : boolean ;} \)

\( end ; \)

\( t0\text{arrival} = record \)

\( t : integer ; \)

\( \text{request: ptrOtran ;} \)

\( end ; \)

\( t0\text{main} = record \)

\( \text{status : integer ;} \)

\( end ; \)

\( \text{eventOnotice} = record \)

\( \text{next, prev, qhead: ptrOevent;} \)

\( \text{inqueue, named: boolean;} \)
eventtime: real;
case eventtype: t0event of
  noevent : ();
  begincheck : (a0begincheck : t0begincheck );
  generatran : (a0generatran : t0generatran );
  create : (a0create : t0create );
  endcheck : (a0endcheck : t0endcheck );
  dequeue : (a0dequeue : t0dequeue );
  departure : (a0departure : t0departure );
  arrival : (a0arrival : t0arrival );
  main : (a0main : t0main );
end;

var
time : real;
g0notice: ptr0event;
e0set : record
  head : ptr0event;
  size : integer;
  empty : boolean;
end;
current : ptr0event;
i0i: integer;
seed0v: array[1..n0seed] of integer;

(* 31*)
(* 32*) mdelay : real;
(* 33*) reqper : real;
(* 34*) status : integer;
(* 35*) objectstatus : array [1..noob] of record
(* 36*) position : ptr0tran;
(* 37*) chst : checkstatus;
(* 38*) waited : integer;
(* 39*) end;
(* 40*) localtran: array [1..nosite] of integer;
(* 41*) tranno:array [1..nosite] of integer;
(* 42*) slotav : array [1..nosite] of
  array [1..maxtranno] of integer;
(* 43*) incmqueue : array[1..nosite] of tranqueue;
(* 44*) objects : array[1..noob] of tranqueue;
(* 45*) objectstock : array [1..noob] of boolean;
(* 46*) objectts : array [1..noob] of record
(* 47*) tranid: integer;
(* 48*) slotid: integer;
(* 49*) siteid: integer
(* 50*) end;
(* 51*) sitets: array [1..nsite] of integer;
(* 52*) i, j, totaltran: integer;
(* 53*)
(* 54*) completed, aborted: integer;
(* 55*) transtat: statistic;
(* 56*) objbusy: array [1..noob] of statistic;
(* 57*)
(* 58*)
(* 59*)

procedure c0notice(var ptr: ptrOevent; tag: tOev01);
begin (* c0notice *)
  case tag of
  begincheck : new(ptr, begincheck);
  generatran : new(ptr, generatran);
  create : new(ptr, create);
  endcheck : new(ptr, endcheck);
  dequeue : new(ptr, dequeue);
  departure : new(ptr, departure);
  arrival : new(ptr, arrival);
  main : new(ptr, main);
  
  end;
  with ptr do begin; (* with *)
    qhead := nil;
    next := nil;
    prev := nil;
    inqueue := false;
    named := false;
    eventtype := tag;
  end; (* with *)
end; (* c0notice *)

procedure error0x;
(* implementation dependent *)
(* berkeley pascal version *)
begin (* error0x *)
  halt;
end; (* error0x *)
procedure errorOp(errorOid, line : integer);
begin (* print execution error and stop *)
  writeln('***error at SIMPAS source line ',line:1);
  case errorOid of
    1 : writeln(' tried to reschedule a ',
                'non-existent event notice');
    2,6 : writeln(' tried to reschedule a ',
                'scheduled event notice');
    3 : writeln(' before/after phrase ',
                'refers to unscheduled event');
    4 : writeln(' tried to destroy a ',
                'scheduled event notice');
    5 : writeln(' tried to schedule event at ',
                'time already passed');
    7 : writeln(' tried to cancel ',
                'unscheduled event');
    8 : begin
      writeln(' tried to remove entity ',
              'from queue it is not in');
      writeln(' or tried to remove ',
              'first/last from an empty set');
    end;
    9 : writeln(' tried to insert entity ',
              'already in a queue');
    10 : writeln(' tried to insert before/after entity not in that queue');
    11 : writeln(' user removed the loop ',
                'variable in a forall loop');
  end; (* case *)
  errorOx;
end; (* errorOp *)

procedure sOcontrol(var status:integer); forward;

function scheduled(name: ptrOevent):boolean;
begin
  if name <> nil then
    scheduled:= name^.qhead.evOset.head
  else
    end;
scheduled := false;
end; (* scheduled *)

procedure schedOn(name: ptrOevent; line: integer);
begin (* schedOn *)
  if not scheduled(name) then
    begin
      errorOp(3, line);
    end;
end; (* schedOn *)

procedure doNotice(var ptr: ptrOevent);
begin (* doNotice *)
  if ptr<> nil then
    begin
      ptr := nil;
    end;
end; (* doNotice *)

procedure eOinsert(ptr: ptrOevent; line: integer);
var
  gOnotice: ptrOevent;
begin (* eOinsert *)
  if (ptr^.evtime < time) then
    errorOp(5, line)
  else if scheduled(ptr) then
    errorOp(6, line)
  else
    with evOset do begin
      size := size + 1;
      empty := false;
      gOnotice := head;
      while (gOnotice^.prev^.evtime >
        ptr^.evtime) do
        gOnotice := gOnotice^.prev;
      with ptr do begin
        next := gOnotice;
        prev := gOnotice^.prev;
        gOnotice^.prev^.next := ptr;
        gOnotice^.prev := ptr;
inqueue := true;
qhead := evOset.head;
end; /* with */
end; /* else */
end; /* e0insert */

function r0random(var seed : integer) : real;
const
  K = 16807;
  P = 2147483647; (* 2^31 - 1 *)
  Q = 32768; (* 2^15 *)
  R = 65536; (* 2^16 *)
var x1, x2, y1, y2 : integer;
begin
  (* portable version of LLRANDOM *)
  (* assumes 32 bit or larger word size *)
  x1 := seed mod Q * K;
  x2 := seed div Q * K;
  y1 := x2 mod R * Q;
  y2 := x2 div R;
  if x1 <= P - y1 then
    y1 := x1 + y1
  else
    begin
      y1 := x1 - P + y1 - 1;
      y2 := y2 + 1;
    end;
  if y1 < P - y2 then seed := y1 + y2
  else seed := y1 - P + y2;
  r0random := seed / P;
end; /* r_random */

function u0random(stream: integer): real;
(* uniform random number generator *)
(* it is assumed that u0random returns a *)
(* number in [0,1) *)
begin (* u0random *)
  if stream > 0 then
    u0random := r0random(seed0v[stream])
else
  u0random := 1.0 -
  r0random(seed0v[-stream]);
end; (* u0random *)

procedure clear(var s : statistic;
  typ : stat0type);
begin
  with s do
    begin
      stype := typ;
      nobs := 0;
      max := 0.0;
      min := 0.0;
      case stype of
        tally,
        accumulate,
        accum :
          begin
            total := 0.0;
            mk := 0.0;
            tk := 0.0;
            mean := 0.0;
            variance := 0.0;
            t0start := time;
            t0last := time;
          end;
        interval :
          begin
            ysum := 0.0;
            asum := 0.0;
            ysum2 := 0.0;
            asum2 := 0.0;
            sumprod := 0.0;
          end;
      end; (* case *)
    end; (* with *)
end; (* clear *)

procedure r0observe(value : real;
  var s:statistic);
var
  temp1, temp2 : real;
begin
  with s do
  begin
    nobr := nobr + 1;
    total := total + value;
    if nobr = 1 then
      begin
        max := value;
        min := value;
      end
    else
      begin
        if value > max then max := value;
        if value < min then min := value;
      end;
    case stype of
      noOstat : begin
        writeln(output, '*** tried to use ');
        error0x;
      end;
      tally : begin
        temp1 := (value - mk);
        mean := mk + 1/nobr * temp1;
        tk := tk + 1/nobr * temp1 * temp1 * (nobs - 1);
        if nobr > 1 then variance := tk/(nobs - 1);
        mk := mean;
      end;
      accum, accumulate : if time > t0start then
        begin
          temp1 := (time-t0last)/(time-t0start);
          temp2 := value - mk;
          mean := mk + temp1 * temp2;
          tk := tk + temp1 * temp2 * temp2
            * (t0last-t0start);
          if nobr>1 then
\begin{verbatim}
variance := (tk * nobs) / ((nobs - 1) * (time - t0start));
mk := mean;
t0last := time;
end;
end; (* case *)
end; (* with *)
end; (* r_observe *)

procedure i0observe(val : integer;
var s : statistic);
var
temp : real;
begin
    temp := val;
r0observe(temp, s);
end (* i0observe *);

procedure b0observe(val : boolean;
var s : statistic);
begin
    if val then r0observe(1.0, s)
    else r0observe(0.0, s);
end (* b0observe *);

procedure c0observe(var s, cs : statistic);
var yval, aval : real;
begin
    case s.s.type of
        noOstat : begin
            writeln(output, '*** tried to use ');
            writeln('uninitialised statistic variable');
            error0x;
        end;
tally : begin
            aval := s.nobs;
yval := s.total;
        end;
        accum,
        accumulate : begin
            aval := s.t0last - s.t0start;
end;
end;
end;
end; (* c0observe *)
\end{verbatim}
yval := aval's.mean;
end;

interval : begin
    writeln(output, '*** tried to ',
         'observe an interval statistic');
    error0x;
end;
end; (* case *)

if (s.max > cs.max) or (cs.nobs = 0)
    then cs.max := s.max;
if (s.min < cs.min) or (cs.nobs = 0)
    then cs.min := s.min;
clear(s, s.type);
if cs.stype <> interval then
    begin
        writeln(output, '*** Interval statistic ',
                 'required by c_observe');
        error0x;
    end;

with cs do
    begin
        nobs := nobs + 1;
ysum := ysum + yval;
asm := asum + aval;
ysum2 := ysum2 + yval*yval;
asum2 := asum2 + aval*aval;
sunprod := sumprod + yval*aval;
    end;
end; (* c_observe *)

procedure c0calc(var cs : statistic;
    sig : real; var point, hwidth : real);
var ybar, abar, yvar, avar, covar,
    sigma2, sigma : real;
begin
    if cs.stype <> interval then
        begin
            writeln(output, '*** Interval ',
                    'statistic required by c_calc');
            error0x;
        end;
with cs do
begin
if nobs <= 1 then
begin
  write(output, '*** c_calc: Too ',
    'few regeneration cycles to ');
  writeln(output, 'compute ',
    'confidence interval');
  error0x;
end;
ybar := ysum/nobs;
abar := asum/nobs;
yvar := (ysum2 - nobs*ybar*ybar)/
(nobs - 1);
avar := (asum2 - nobs*abar*abar)/
(nobs - 1);
covar := (sumprod - nobs*ybar*abar)/
(nobs - 1);
if asum = 0.0 then
begin
  writeln(output, '*** c_calc: ',
    'Regeneration cycle of length zero');
  error0x;
end;
point := ysum/asum;
sigma2 := yvar - 2*point*covar +
  point*point*avar;
if sigma2 < 0.0 then
begin
  writeln(output, '*** c_calc: ',
    'negative variance estimate ***');
  error0x;
end;
sigma := sqrt(sigma2);
hwidth := (sig*sigma)/(sqrt(nobs)*abar);
end;
end; (* c_calc *)

function poisson(lambda: real;
  stream: integer): integer;
const
  lowerbound = 0.000001;
  upperbound = 0.999999;
var
  u, a, b: real;
  variate: integer;
begin
  if (lambda <= 0.0) then
    begin
      write('*** error in function poisson - ');
      writeln('called with parameter lambda=',
                lambda, ' ***');
      error0x;
    end;
  a := exp(-lambda);
  variate := 0;
  b := a;
  u := u0random(stream);
  while (u > a) and (a <= upperbound) and
    (b >= lowerbound) do begin
    variate := variate + 1;
    b := (b * lambda) / variate;
    a := a + b;
  end;
  poisson := variate;
end; (* function poisson *)

procedure c0tran (var e: ptr0tran);
begin
  new(e);
  with e^ do begin
    inqueue := false;
    qhead := nil;
    next := nil;
    prev := nil;
  end;
end;

procedure d0tran (var e:ptr0tran);
begin
  if e^.inqueue then begin
write('***tried to destroy an entity',
    'of type tran');
write(' which is still in a queue***');
errorOx;
end;
e:=nil;
end;

procedure iOtranqueue (var q: tranqueue);
begin
  new(q.head);
  with q.head do
  begin
    next := q.head;
    prev := q.head;
    qhead := nil;
    inqueue := false;
  end;
  q.size := 0;
  q.empty := true;
  clear(q.stat, accumulate);
end;

(* 59*)
(* 60*)
(* 61*) procedure rOarrival (rOarg: tOarrival); forward;
(* 62*) procedure rOdeparture (rOarg: tOdeparture); forward;
(* 63*) procedure rOdequeue (rOarg: tOdequeue); forward;
(* 64*) procedure rOendcheck (rOarg: tOendcheck); forward;
(* 65*)
(* 66*) procedure rOcreate (rOarg: tOcreate);
(* 67*) var found: boolean; i: integer;
(* 68*) newtran: ptrOtran; x:real;
(* 69*) begin with rOarg do
(* 70*) if not incmqueue[s].empty and
    (tranno[s] < maxtranno) then
begin
begin
newtran := incmqueue[s].head'.next;
with incmqueue[s] do
begin
if newtran'.qhead <> head then error0p(8,72);
roobserve(size,stat); size := size - 1;
empty := size=0; end;
with newtran' do
begin
inqueue := false; qhead := nil;
prev'.'next := next; next'.'prev := prev;
next:=nil; prev:=nil;
end; end

tranno[s] := tranno[s] + 1;
newtran'.'checked :=false;
newtran'.'siteid := s;
totaltran := totaltran + 1;
if newtran'.'tag=0 then begin
localtran[s] := localtran[s] + 1;
newtran'.'tag := localtran[s]*100+s;
newtran'.'stime := time;
end;
found := false; i:=1;
while (i<=maxtranno) and not found do
begin
if slotav[s][i]=0 then begin
slotav[s][i]:=1;
newtran'.'slotid := i;
found := true
end;
i := i+1
end;
sitets[s] := sitets[s] + tsinterval;
newtran'.'trid := sitets[s];
for i:=1 to noob do
begin newtran'.'lockobtained[i]:=false;
newtran'.'nextts[i]:=0
end;
(* 98*)  newtran^.nowaits:=0;
(* 99*)  newtran^.sd1:=newtran^.sd01;
    newtran^.sd2:=newtran^.sd02;
(*100*)  seed0v[50]:=newtran^.sd2;
(*101*)  x := u0random(50) * noob;
(*102*)  newtran^.sd2:=seed0v[50];
(*103*)  if x < noob then i := trunc(x) + 1
     else i := trunc(x);
(*104*)
(*104*)  begin
(*104*)  c0notice{ g0notice, arrival    };
(*104*)  with g0notice do begin
(*104*)  a0arrival .t := i ;
(*104*)  a0arrival .request := newtran ;
(*104*)  end;
(*104*)  g0notice^.evtime:= time +
(*105*)  poisson(servicerate, nosite+s)+mdeay ;
(*105*)  e0insert{ g0notice, 105 };
(*105*)  g0notice:= nil;
(*105*)  end
(*105*)
(*106*)  end
(*107*)  end
(*107*)  end
(*107*)
(*108*)
(*109*)  procedure r0generatran
     (r0arg: t0generatran );
(*110*)
(*110*)  var
(*111*)  newtran:ptr0tran; x:real;
(*112*)  begin with r0arg do
(*112*)  begin
(*113*)
(*114*)
(*115*)
(*116*)  c0tran(newtran);
(*117*)  newtran^.sd01:= seed0v[ freeseed+s ];
(*118*)  newtran^.sd02:= seed0v[ freeseed+nosite+s ];
(*119*)  x:=u0random( freeseed+s);
    x:=u0random( freeseed+nosite+s );
(*120*)  newtran^.tag:=0;
begin
  if newtran^\ast.inqueue then errorOp(9,121);
  with newtran^\ast do
    with incmqueue[s] do begin
      qhead:= head^\ast.prev;
      next:=qhead^\ast.next; prev:=qhead;
      qhead^\ast.next:=newtran;
      next^\ast.prev:=newtran;
      inqueue:=true; qhead:=head;
      r0observe(size,stat); size:=size+1;
      empty:=false; end; end
end
begin
  c0notice( g0notice, generatran );
  with g0notice^\ast do begin
    a0generatran .s := s ;
    g0notice^.evtime:= time + poisson(arrivalrate,s) ;
    e0insert( g0notice, 122 );
    g0notice:= nil;
  end
end
if tranno[s] < maxtranno then begin
  c0notice( g0notice, create );
  with g0notice^\ast do begin
    a0create .s := s ;
    g0notice^.evtime:= time + 0
  end
  e0insert( g0notice, 124 );
  g0notice:= nil;
  end
end
end
end
end

function greater(tr1, sitel, slot1,
tr2, site2, slot2: integer) :boolean;
begin
if tr1 > tr2 then greater := true
else if tr1 < tr2 then greater := false
else if sitel > site2
then greater := true
else if sitel < site2
then greater := false
else if slot1 > slot2
then greater := true
else greater := false
end;
procedure rObegincheck
(rOarg: TObegincheck);
var item : ptrOtran;i:integer;
found:boolean;
begin with rOarg do
begin
item := objectstatus[t].position;
if greater(item^.trid, item^.siteid,
item^.slotid, objectts[t].tranid,
objectts[t].siteid, objectts[t].slotid)
then begin
coNotice( gOnotice, endcheck );
with gOnotice^ do begin
aOendcheck .t := t ;
end;
gOnotice^.evtime:= time + cdelay
end;
eOinsert( gOnotice, 146 );
gOnotice:= nil;
end
else begin
objectstatus[t].chst:=wait;
found:=true;
for i:=1 to noob do
if item^.lockobtained[i] and
(*151*) then if objects[i].empty then begin
(*152*) objectstatus[i].chst:=freeze;
(*153*) item^nextts[i]:=-1 end.
(*154*) else
(*155*) item^nextts[i]:=
objects[i].head^.next^.trid
(*156*) else begin
(*157*) found:=false;
(*158*) item^.nowaits:=item^.nowaits+1;
(*159*) item^.nextts[i]:=1;
(*160*) objectstatus[i].waited:=t
ead;
(*162*) if found then
(*163*) begin
(*164*) c0notice( g0notice, endcheck );
(*165*) with g0notice do begin
(*166*) a0endcheck .t := t ;
(*167*) end;
(*168*) g0notice^.evtime:=time+cdelay+mdelay
ead;
(*169*) e0insert( g0notice, 163 );
(*170*) g0notice:= nil;
end
end
end
end
end
end
end
end
end
end
end

(*165*)

(*166*) procedure r0endcheck;
(*167*) var
(*168*) next0item, item : ptr0tran;
i,j,max,min:integer;
(*169*) a,b,c,d,k:integer; found : boolean;
(*170*) begin with r0arg do
(*171*) item := objectstatus[t].position;
(*172*) next0item := item^.next;
(*173*) if objectstatus[t].chst = normal
(*174*) then begin item^.checked := true;
objectts[t].tranid := item*.trid;
*/176*/ objectts[t].slotid:=item*.slotid;
*/177*/ objectts[t].siteid:=item*.siteid;
*/178*/ if not objecttslock[t] then
*/178*/ begin
*/178*/ c0notice( g0notice, dequeue )
*/178*/ with g0notice* do begin
*/178*/ a0dequeue .t := t ;
*/178*/ end;
*/178*/ g0notice*.evtime := time + 0
*/179*/ ;
*/179*/ e0insert( g0notice, 170 );
*/179*/ g0notice := nil;
*/179*/ end
*/179*/ end
*/180*/ else begin
*/181*/ max := maxint;
*/182*/ min := objectts[t].tranid;
*/183*/ for i:= 1 to noob do
*/184*/ if (item*.lockobtained[i]) and
*/184*/ (item*.nextts[i]<max) then
*/185*/ and (item*.nextts[i]<max) then
*/186*/ max := item*.nextts[i];
*/187*/ if (max - min) > 1 then begin
*/188*/ k:=item*.siteid;
*/189*/ a:=min div tsinterval;
*/190*/ b:=max div tsinterval;
*/191*/ c:=sitets[k] div tsinterval;
*/192*/ found := false;
*/193*/ d:=1;
*/194*/ while not found and (c+d < b) do
*/195*/ if c+d > a then found := true
*/196*/ else d:= d+1;
*/197*/ if found then
*/197*/ begin sitets[k]:=(c+d)*20;
*/198*/ j := sitets[k] end
*/199*/ else j := trunc( (max+min)/2);
*/200*/ item*.trid:=j;
*/201*/ item*.checked := true;
*/202*/ objectts[t].tranid := j;
*/203*/ objectts[t].slotid := item*.slotid;
(*204*) objectts[t].siteid := item*.siteid;
(*205*) for i := 1 to noob do
(*206*) if item*.nextts[i]=-1 then
(*207*) objectts[i].tranid := j;
(*208*) if not objectslck[t] then
(*208*) begin
(*208*) c0notice( g0notice, dequeue );
(*208*) with g0notice* do begin
(*208*) a0dequeue . t := t ;
(*208*) end;
(*208*)
(*209*) g0notice*.evtime := time + 0 ;
(*209*) e0insert( g0notice, 209 );
(*209*) g0notice := nil;
(*209*) end
(*210*) ;
(*211*) end else begin
(*212*) begin
(*212*) with objectts[t] do begin
(*212*) if item*.qhead <> head
(*212*) then error0p(8,212);
(*212*) r0observe(size,stat);
(*212*) size := size - 1;
(*212*) empty := size=0; end;
(*212*) with item* do begin
(*212*) inqueue := false; qhead := nil;
(*212*) prev*.next := next;
(*212*) next*.prev := prev;
(*212*) next:=nil; prev:=nil;
(*212*) end; end
(*212*)
(*213*) begin
(*213*) c0notice( g0notice, departure );
(*213*) with g0notice* do begin
(*213*) a0departure .request := item ;
(*213*) a0departure .ss := false ;
begin
  g0notice^.evtime := time + mdelay;
end;
end;
e0insert( g0notice, 214 );
g0notice := nil;
end;
end;
end;

objectstatus[t].chst := normal;
for i := 1 to noob do
  if item^.nextts[i] = -1 then
    begin
      objectstatus[i].chst := normal;
      item^.nextts[i] := 0;
    end;
  if not objects[i].empty then begin
    objectstatus[i].position :=
      objects[i].head^.next;
    begin
      conotice( g0notice, begincheck );
      with g0notice do begin
        abegincheck .t := i ;
      end;
      g0notice^.evtime := time + mdelay;
      e0insert( g0notice, 223 );
g0notice := nil;
end;
end;
end;
end;
if next0item <> objects[t].head then begin
  objectstatus[t].position := next0item;
begin
  conotice( g0notice, begincheck );
  with g0notice do begin
    abegincheck .t := t ;
  end;
  g0notice^.evtime := time + cdelay
end;
e0insert( g0notice, 229 );
g0notice := nil;
(*229*) end
(*230*) i := objectstatus[t].waited;
(*231*) if i <> 0 then begin
(*232*) item := objectstatus[i].position;
(*233*) if objects[t].empty then begin
(*234*) objectstatus[t].chst := freeze;
(*235*) item^.nextts[t]^ := -1;
(*236*) item^.nowaits := item^.nowaits - 1;
(*237*) if item^.nowaits = 0 then
(*238*) begin
(*239*) c0notice( g0notice, endcheck );
(*240*) with g0notice do begin
(*241*) a0endcheck .t := i ;
(*242*) end;
(*243*) g0notice^.evtime := time + mdelay ;
(*244*) e0insert( g0notice, 237 );
(*245*) g0notice := nil;
(*246*) end
(*247*) ;
(*248*) objectstatus[t].waited := 0
(*249*) end
(*250*) else
(*251*) if objects[t].head^.next^.checked then begin
(*252*) item^.nextts[t]^ :=
(*253*) objects[t].head^.next^.trid;
(*254*) item^.nowaits := item^.nowaits - 1;
(*255*) if item^.nowaits = 0 then
(*256*) begin
(*257*) c0notice( g0notice, endcheck );
(*258*) with g0notice do begin
(*259*) a0endcheck .t := i ;
(*260*) end;
(*261*) g0notice^.evtime := time + mdelay ;
(*262*) e0insert( g0notice, 243 );
(*263*) g0notice := nil;
(*264*) end
(*265*) ;
(*266*) objectstatus[t].waited := 0
(*267*) end
procedure rOarrival;
begin with rOarg do
begin
request^.checked := false;
begin
if request^.inqueue then errorOp(9, 252);
with request^ do
with objects[t] do begin
qhead:= head^.prev;
next:=qhead^.next; prev:=qhead;
qhead^.next:=request;
next^.prev:=request;
inqueue:=true; qhead:=head;
rOobserve(size, stat); size:=size+1;
empty:=false; end; end

if objectstatus[t].chst=normal then
if (objects[t].head = request^.prev )
then begin
objectstatus[t].position := request;
begin
cOnotice( gOnotice, begincheck );
with gOnotice^ do begin
aObegincheck .t := t ;
end;
gOnotice^.evtime:= time + cdelay
end;
eOinsert( gOnotice, 258 );
gOnotice:= nil;
end
else if request^.prev^.checked then begin
procedure r0departure;
var i : integer;
begin with r0arg do
begin
transno[request^.siteid] :=
transno[request^.siteid]-1;
slotav[request^.siteid]
[request^.slotid] := 0;
if ss then completed:=completed+1
else aborted:=aborted+1;
for i:=1 to noob do
if request^.lockobtained[i] then
begin
begin
begin
with g0notice^ do begin
a0dequeue .t := i ;
end;
end;
e0insert( g0notice, 274 );
g0notice:= nil;
end
end
end
end
end
begin
c0notice( g0notice, main );
with g0notice do begin
   a0main.status := normalterm;
end;
with g0notice do begin
   evtime := time;
   next := ev0set.head^.next;
   ev0set.head^.next := g0notice;
   prev := ev0set.head;
   next^.prev := g0notice;
   inqueue := true;
   qhead := ev0set.head;
end; (* with *)
with ev0set do begin
   size := size + 1;
   empty := false;
end; (* with *)
g0notice := nil;
end
else
begin
c0notice( g0notice, create );
with g0notice do begin
   a0create^.s := request^.siteid;
end;
g0notice^.evtime := time + 0;
e0insert( g0notice, 270 );
g0notice := nil;
end
end
if ss then begin
   r0observe( time-request^.stime, transtat);
d0tran(request) end
else begin
   if request^.inqueue then error0p(9,280);
   with request do
   with incmqueue[request^.siteid]
do begin
   qhead := head^.prev;
procedure rodeque;

var request:ptr0tran; x:real;

begin with roarg do

if objects[t].empty then begin
  objectslock[t] := false;

  observe(true, objbusy[t])
end else
  if objects[t].head^.next^.checked
  then begin
    begin
      request := objects[t].head^.next;
      begin with objects[t] do
        if request^.qhead <>
          head then errorOp(8, 288);
      end;
    end;

    observe(size, stat);
    size := size - 1;

    empty := size=0; end;

    begin
      request := nil;
      begin
        inqueue := false; qhead := nil;
        prev^.next := next;
        next^.prev := prev;
        next := nil; prev := nil;
      end; end

    request^.lockobtained[t] := true;

    if not objectslock[t] then begin
      objectslock[t] := true;

      observe(false, objbusy[t])
    end
  end;
end;
end;
seed0v[50]:=request^.sdl;
x := u0random(50);
request^.sdl:=seed0v[50];
if x > reqper then
begin
  c0notice(g0notice, departure);
  with g0notice^ do begin
    a0departure.request := request;
    a0departure.ss := true;
  end;
g0notice^.evtime := time+poisson(servicerate, nosite+request^.siteid)
+mdelay;
end;
e0insert(g0notice, 299);
g0notice:= nil;
end
else begin
  repeat
    seed0v[50]:=request^.sdl2;
x := u0random(50) * noob;
request^.sdl2:=seed0v[50];
if x < noob then i:= trunc(x) + 1
else i:=trunc(x);
until request^.lockobtained[i] = false;
begin
  c0notice(g0notice, arrival);
  with g0notice^ do begin
    a0arrival.t := i;
    a0arrival.request := request;
  end;
g0notice^.evtime := time+poisson(servicerate, nosite+request^.siteid)
+2*mdelay;
end;
e0insert(g0notice, 309);
g0notice:= nil;


```plaintext
procedure sOcontrol;
label 9999;
begin (* sOcontrol *)
  with evOset do begin
    current := head'.next;
    while current <> head do begin
      with current' do begin
        time := evt ime;
        next'.prev := prev;
        prev'.next := next;
        next := nil;
        prev := nil;
        qhead := nil;
        inqueue := false;
      end; (* with *)
      size := size - 1;
      empty := size=0;
      case current'.evettype of
        begincheck: r0begincheck(current'.a0begincheck);
        generatran: r0generatran(current'.a0generatran);
        create: r0create(current'.a0create);
        endcheck: r0endcheck(current'.a0endcheck);
        dequeue: r0dequeue(current'.a0dequeue);
        departure: r0departure(current'.a0departure);
        arrival : r0arrival(current'.a0arrival);
        main: begin
          status := current'.a0main.status;
          goto 9999;
        end;
```
end; (* case *)
if (current <> nil) then
  if (not scheduled(current)) and
      (not current^.named) then
    d0notice(current);
  current := head^.next;
end;
end;
status := 0;
9999:
end; (* socontrol-*
begin
  time := 0;
new(evOset^.head, no0event);
with evOset^.head do begin
  next := evOset^.head;
  prev := evOset^.head;
  qhead := evOset^.head;
  evtime := -1;
inqueue := false;
end; (* with *)
with evOset do begin
  empty := true;
sise := 0;
end;

{* set default seeds for LLRAND0M *}
seed0v[1] := 1973272912;
seed0v[2] := 319340905;
seed0v[3] := 887514864;
seed0v[4] := 1638801628;
seed0v[5] := 1928201733;
seed0v[6] := 911320271;
seed0v[7] := 87021785;
seed0v[8] := 88473758;
seed0v[9] := 1868734899;
seed0v[10] := 26784697;
seed0v[11] := 197372912;
seed0v[12] := 3140905;
seed0v[13] := 8875864;
seed0v[14] := 38801628;
seed0v[15] := 92201733;
seed0v[16] := 910271;
seed0v[17] := 821785;
seed0v[18] := 843758;
seed0v[19] := 18734899;
seed0v[20] := 2697;

seed0v[21] := 19272912;
seed0v[22] := 9340905;
seed0v[23] := 8754864;
seed0v[24] := 163801628;
seed0v[25] := 128201733;
seed0v[26] := 9120271;
seed0v[27] := 871785;
seed0v[28] := 8873758;
seed0v[29] := 18734899;
seed0v[30] := 2784697;

seed0v[31] := 45974321;
seed0v[32] := 3609825;
seed0v[33] := 456943214;
seed0v[34] := 268095438;
seed0v[35] := 8999933;
seed0v[36] := 62091771;
seed0v[37] := 34865425;
seed0v[38] := 98993678;
seed0v[39] := 11118899;
seed0v[40] := 529983188;

seed0v[41] := 67064912;
seed0v[42] := 12384205;
seed0v[43] := 9538104;
seed0v[44] := 17028;
seed0v[45] := 87392;
seed0v[46] := 4702111;
seed0v[47] := 7895435;
seed0v[48] := 194297608;
seed0v[49] := 984327810;
seed0v[50] := 764319442;
$\text{seed0v[51]} := 77064912$
$\text{seed0v[52]} := 62384205$
$\text{seed0v[53]} := 2538104$
$\text{seed0v[54]} := 47028$
$\text{seed0v[55]} := 57392$
$\text{seed0v[56]} := 670211$
$\text{seed0v[57]} := 9895435$
$\text{seed0v[58]} := 394297608$
$\text{seed0v[59]} := 584327810$
$\text{seed0v[60]} := 164319442$

$\text{seed0v[61]} := 6704912$
$\text{seed0v[62]} := 1234205$
$\text{seed0v[63]} := 953104$
$\text{seed0v[64]} := 1708$
$\text{seed0v[65]} := 8732$
$\text{seed0v[66]} := 47011$
$\text{seed0v[67]} := 789435$
$\text{seed0v[68]} := 19497608$
$\text{seed0v[69]} := 98427810$
$\text{seed0v[70]} := 76419442$

$\text{seed0v[71]} := 8764912$
$\text{seed0v[72]} := 124205$
$\text{seed0v[73]} := 958104$
$\text{seed0v[74]} := 1728$
$\text{seed0v[75]} := 8792$
$\text{seed0v[76]} := 47211$
$\text{seed0v[77]} := 785435$
$\text{seed0v[78]} := 19297608$
$\text{seed0v[79]} := 98327810$
$\text{seed0v[80]} := 76319442$

$\text{seed0v[81]} := 7064912$
$\text{seed0v[82]} := 2384205$
$\text{seed0v[83]} := 538104$
$\text{seed0v[84]} := 7028$
$\text{seed0v[85]} := 7392$
$\text{seed0v[86]} := 70211$
$\text{seed0v[87]} := 895435$
$\text{seed0v[88]} := 94297608$
seed0v[89]:=84327810;
seed0v[90]:=84319442;

seed0v[91]:=6064912;
seed0v[92]:=1384205;
seed0v[93]:=938104;
seed0v[94]:=1028;
seed0v[95]:=8392;
seed0v[96]:=40211;
seed0v[97]:=795435;
seed0v[98]:=14297608;
seed0v[99]:=94327810;
seed0v[100]:=64319442;

(*315*) readln(mdelay,reqper);
(*316*) totaltran:=0;
(*317*) completed:=0;
(*318*) aborted:=0;
(*319*) clear(transt,tally);
(*320*) for i:=1 to noob do (*321*) clear(objbusy[i],accumulate);
(*322*) for i:=1 to nosite do begin
(*323*) iotranqueue(incmqueue[i]);
(*324*) localtran[i]:=0;
(*325*) tranno[i]:=0;
(*326*) for j:=1 to maxtranno do (*327*) slotav[i][j]:=0;
(*328*) sitets[i]:=0
(*329*) end;
(*330*) for i:=1 to noob do begin
(*331*) iotranqueue(objects[i]);
(*332*) objectstatus[i].chst:=normal;
(*333*) objectstatus[i].waited:=0;
(*334*) objectlock[i]:=false;
(*335*) objectts[i].tranid:=0;
(*336*) objectts[i].slotid:=0;
(*337*) objectts[i].siteid:=0
(*338*) end;
(*339*) for i:=1 to nosite do (*340*)
(*340*) begin
c0notice( g0notice, generatran );

with g0notice do begin
  a0generatran.s := i;
end;
g0notice.evtime := time +
  poisson(arrivalrate, i);
endinsert( g0notice, 340 );
g0notice := nil;
end

s0control( status);
writeLn('Adjustable timestamps');
writeLn('mdelay=', mdelay: 6:2);
writeLn('reqper=', reqper: 6:2);
writeLn('total no. of transactions',
  'generated:', totaltran: 5);
writeLn('total completions:', completed: 5);
writeLn('total aborts:', aborted: 5);
writeLn('mean response time:',
  transtat.mean: 10);
writeLn('ratio:',
  completed * 100 / totaltran: 10 );
APPENDIX B

Simulation Program for the Timestamping with Locks Scheme

The complete simulation program for the timestamping with locks scheme is listed here. REQPER and message delay time are two input parameters to this simulator. Other simulation parameters are made constants for the ease of modification.

(* 1*) program cc tslk(input,output);
(* 2*)
const
{ version 3.5 simpas output }
noseed = 100;
(* 2*)
(* 3*) freeseed = 50;
(* 4*) cdelay = 0.01;
(* 5*) nosite = 10;
(* 6*) noob = 20;
(* 7*) maxtranno = 5;
(* 8*) maxtotaltran = 30 ;
(* 9*) normalterm = 1;
(* 10*) servicerate = 0.6;
(* 11*) arrivalrate = 0.25;
(* 12*) tsinterval = 20;
(* 13*)

type
t0evol = (no0event,
beginc0ehck ,
generatran, 
create, 
endcheck, 
dequeue, 
departure, 
arrival, 
main 
};
ptrOtran = tran;
statOtype = (noOstat, tally, accumulate, accum, interval);
ptrOevent = "eventOnotice; 
statistic = record 
nobs : integer; 
max, min : real; 
case stype : statOtype of 
  tally, 
  accumulate, 
  accum : 
    (mean, variance : real; 
     total, mk, tk : real; 
     tOstart, tOlast : real); 
  interval : 
    (asum, ysum : real; 
     asum2, ysum2 : real; 
     sumprod : real);
end;
(* 13*)
(* 14*) checkstatus = ( normal, wait, freeese );
(* 15*) tran = record 
  (* 15*) next, prev, qhead: ptrOtran ;
(pop) inqueue: boolean; 
(* 15*)
(* 16*) tag : integer;
(* 17*) stime : real ;
(* 18*) trid : integer;
(* 19*) slotid : integer;
(* 20*) siteid : integer;
(* 21*) checked : boolean;
(* 22*) lockobtained : array [1..noob] of boolean;
(* 23*) nextts : array[1..noob] of integer;
(* 24*)
(* 25*) nowaits : integer;
(* 26*) sd1 : integer;
(* 27*) sd2 : integer;
(* 28*) sd01, sd02 : integer;
(* 29*) end;
(* 30*) tranqueue = record
(* 30*) head : ptr0tran;
(* 30*) size : integer;
(* 30*) empty : boolean;
(* 30*) stat : statistic;
(* 30*) end
(* 30*)
(* 31*)
t0begincheck = record
t : integer;
end;
t0generatran = record
s : integer;
end;
t0create = record
s : integer;
end;
t0endcheck = record
t : integer;
end;
t0dequeue = record
t : integer;
end;
t0departure = record
request : ptr0tran;
s : boolean;
end;
t0arrival = record
t : integer;
request : ptr0tran;
end;
t0main = record
status : integer;
end;
event0notice = record
next, prev, qhead : ptr0event;
inqueue, named : boolean;
evtime: real;
case eventtype: tOevent of
  noevent : ();
begonclick : (aObegonclick : tObegonclick);
generroran : (aOgeneratran : tOgeneratran);
create : (aOcreate : tOcreate);
endcheck : (aOendcheck : tOendcheck);
dequeue : (aOdequeue : tOdequeue);
deporture : (aOdeporture : tOdeporture);
arrival : (aOarrival : tOarrival);
main : (aOmain : tOmain);
end;
var
time : real;
gOnotice : ptrOevent;
evOset : record
  head : ptrOevent;
  size : integer;
  empty : boolean;
end;
current : ptrOevent;
iOi: integer;
seedOv: array[1..nOseed] of integer;

(* 31*)
(* 32*) mdelay : real;
(* 33*) reqper : real;
(* 34*) status : integer;
(* 35*) objectstatus : array [1..noob] of record
(* 36*) position : ptrOtran;
(* 37*) chst : checkstatus;
(* 38*) waited : integer;
(* 39*) end;
(* 40*) localtran: array [1..nosite] of integer;
(* 41*) tranno:array [1..nosite] of integer;
(* 42*) slotav : array [1..nosite] of
  array [1..maxtranno] of integer;
(* 43*) incmqueue : array[1..nosite] of tranqueue;
(* 44*) objects : array[1..noob] of tranqueue;
(* 45*) objectslock : array [1..noob] of boolean;
(* 46*) objectts : array [1..noob] of record
procedure c0notice(var ptr: ptroevent; tag: t0ev01);
begin (* c0notice *)
  case tag of
  begincheck  : new(ptr,begincheck);
  generatetrans : new(ptr,generatetrans);
  create : new(ptr,create);
  endcheck : new(ptr,endcheck);
  dequeue : new(ptr,dequeue);
  departure : new(ptr,departure);
  arrival : new(ptr,arrival);
  main : new(ptr,main);
end; (* with *)
end; (* c0notice *)

procedure error0x;
(* implementation dependent *)
(* berkeley pascal version *)
begin (* error0x *)
  halt;
end; (* error0x *)
procedure errorOp(errorOid, line : integer);
begin (* print execution error and stop *)
    writeln("**error at SIMPAS source line ",line:1);
    case errorOid of
    1 : writeln(' tried to reschedule a ',
               'non-existent event notice');
    2,6 : writeln(' tried to reschedule a ',
               'scheduled event notice');
    3 : writeln(' before/after phrase ',
               'refers to unscheduled event');
    4 : writeln(' tried to destroy a ',
               'scheduled event notice');
    5 : writeln(' tried to schedule event at ',
               'time already passed');
    7 : writeln(' tried to cancel ',
               'unscheduled event');
    8 : begin
        writeln(' tried to remove entity ',
                'from queue it is not in');
        writeln(' or tried to remove ',
                'first/last from an empty set');
    end;
    9 : writeln(' tried to insert entity ',
               'already in a queue');
    10 : writeln(' tried to insert before/ ',
                'after entity not in that queue');
    11 : writeln(' user removed the loop ',
                'variable in a forall loop');
    end; (* case *)
    errorOx;
end; (* errorOp *)

procedure s0control(var status: integer); forward;

function scheduled(name: ptrOevent):boolean;
begIn
    if name <> nil then
        scheduled:= name^.qhead=ev0set.head
    else
schedu := false;
end; (* scheduled *)

procedure schedOn(name: ptrOevent; line: integer);
begin (* schedOn *)
  if not scheduled(name) then
    begin
      errorOp(3, line);
    end;
end; (* schedOn *)

procedure dOnotice(var ptr: ptrOevent);
begin (* dOnotice *)
  if ptr<> nil then
    begin
      ptr := nil;
    end;
end; (* dOnotice *)

procedure eOinsert(ptr: ptrOevent; line: integer);
var
  gOnotice: ptrOevent;
begin (* eOinsert *)
  if (ptr^.evtime < time) then
    errorOp(5, line)
  else if scheduled(ptr) then
    errorOp(6, line)
  else
    with ev0set do begin
      size := size + 1;
      empty := false;
      gOnotice := head;
      while (gOnotice^.prev^.evtime > ptr^.evtime) do
        gOnotice := gOnotice^.prev;
      with ptr^ do begin
        next := gOnotice;
        prev := gOnotice^.prev;
        gOnotice^.prev^.next := ptr;
        gOnotice^.prev := ptr;
        inqueue := true;
function r0random(var seed: integer): real;
const
K = 16807;
P = 2147483647; (* 2^31 - 1 *)
Q = 32768; (* 2^15 *)
R = 65536; (* 2^16 *)
var x1, x2, y1, y2: integer;
begin
(* portable version of LLRANDOM *)
(* assumes 32 bit or larger word size *)
x1 := seed mod Q * K;
x2 := seed div Q * K;
y1 := x2 mod R * Q;
y2 := x2 div R;
if x1 <= P - y1 then
  y1 := x1 + y1
else
  begin
    y1 := x1 - P + y1 - 1;
y2 := y2 + 1;
  end;
if y1 < P - y2 then seed := y1 + y2
  else seed := y1 - P + y2;
r0random := seed / P;
end; (* r_random *)

function u0random(stream: integer): real;
(* uniform random number generator *)
(* it is assumed that u0random returns a *)
(* number in (0,1) *)
begin (* u0random *)
  if stream > 0 then
    u0random := r0random(seed0v[stream])
  else
u0random := 1.0 -
 )
end; (* u0random *)

procedure clear(var s : statistic;
    typ : stat0type);
begin
    with s do
        begin
            stype := typ;
            nobs := 0;
            max := 0.0;
            min := 0.0;
            case stype of
                tally, accumulate, accum :
                    begin
                        total := 0.0;
                        mk := 0.0;
                        tk := 0.0;
                        mean := 0.0;
                        variance := 0.0;
                        t0start := time;
                        t0last := time;
                    end;
                interval :
                    begin
                        ysum := 0.0;
                        asum := 0.0;
                        ysum2 := 0.0;
                        asum2 := 0.0;
                        sumprod := 0.0;
                    end;
            end; (* case *)
        end; (* with *)
end; (* clear *)

procedure r0observe(value : real;
    var s:statistic);
var
temp1, temp2 : real;
begin
  with s do
  begin
    nobs := nobs + 1;
    total := total + value;
    if nobs = 1 then
      begin
        max := value;
        min := value;
      end
    else
      begin
        if value > max then max := value;
        if value < min then min := value;
      end;
  case stype of
    no0stat : begin
      writeln(output, '*** tried to use ');
      'uninitialized statistic variable');
      error0x;
    end;
    tally : begin
      temp1 := (value - mk);
      mean := mk + 1/nobs * temp1;
      tk := tk + 1/nobs * temp1 * temp1
      * (nobs - 1);
      if nobs > 1 then variance :=
      tk/(nobs - 1);
      mk := mean;
    end;
    accum, accumulate : if time > t0start then
    begin
      temp1 := (time-t0last)/(time-t0start);
      temp2 := value - mk;
      mean := mk + temp1 * temp2;
      tk := tk + temp1 * temp2 * temp2
      * (t0last-t0start);
      if nobs>1 then
        variance := (tk * nobs) /
(nobs - 1) * (time - t0start));
    mk := mean;
    t0last := time;
  end;
end; (* case *)
end; (* with *)
end; (* r_ observe *)

procedure i0observe(val : integer; var s: statistic);
var
  temp : real;
begin
  temp := val;
  rOobserve( temp, s);
end (* i0observe *);

procedure b0observe(val : boolean; var s : statistic);
begin
  if val then rOobserve(1.0, s)
  else rOobserve(0.0, s);
end (* b0observe *);

procedure c0observe (var s, cs : statistic);
var yval, aval : real;
begin
  case s.stype of
    noOstat : begin
      writeln( output, ' *** tried to use ',
               'uninitialised statistic variable');
      error0x;
      end;
    tally : begin
      aval := s.nobs;
      yval := s.total;
      end;
    accum,
    accumulate : begin
      aval := s.t0last - s.t0start;
      yval := aval*s.mean;
      end;
    interval : begin

write ln(output, '*** tried to observe ', 'an interval statistic');
errorOx;
end;
end; (* case *)
if (s.max > cs.max) or (cs.nobs = 0)
then cs.max := s.max;
if (s.min < cs.min) or (cs.nobs = 0)
then cs.min := s.min;
clear(s, s.type);
if cs.type <> interval then
begin
write ln(output, '*** Interval statistic', 
' required by c_observe');
errorOx;
end;
with cs do
begin
nobs := nobs + 1;
ysum := ysum + yval;
asm := asum + aval;
ysum2 := ysum2 + yval*yval;
asm2 := asm2 + aval*aval;
sumpod := sumpod + yval*aval;
end;
end; (* c_observe *)

procedure c0calc(var cs : statistic;
sig : real; var point, hwidth : real);
var ybar, abar, yvar, avar, covar, 
sigma2, sigma : real;
begin
if cs.type <> interval then
begin
write ln(output, '*** Interval ', 
'statistic required by c_calc');
errorOx;
end;
with cs do
begin
if nobs <= 1 then

begin
  write(output, '*** c_calc: Too ',
        'few regeneration cycles to ');
  writeln(output, 'compute ',
           'confidence interval');
  error0x;
end;
ybar := ysum/nobs;
abar := asum/nobs;
yvar := (ysum2 - nobs*ybar*ybar)/(nobs - 1);
avar := (asum2 - nobs*abar*abar)/(nobs - 1);
covar := (sumprod - nobs*ybar*abar)/(nobs - 1);
if asum = 0.0 then
begin
  writeln(output, '*** c_calc: ',
           'Regeneration cycle of length zero');
  error0x;
end;
point := ysum/asum;
sigma2 := yvar - 2*point*covar + point*point*aovar;
if sigma2 < 0.0 then
begin
  writeln(output, '*** c_calc: ',
           'negative variance estimate ***');
  error0x;
end;
sigma := sqrt(sigma2);
hwidth := (sig*siga)/(sqrt(nobs)*abar);
end;
end;(* c_calc *)

function poisson(lambda: real; stream: integer): integer;
const
lowerbound = 0.000001;
upperbound = 0.999999;
var
  u, a, b: real;
  variate: integer;
begin
  if (lambda <= 0.0) then
begin
  writeln('*** error in function poisson - ');
  writeln(' called with parameter lambda=', lambda, ' ***');
  errorOx;
end;
a := exp(-lambda);
variate := 0;
b := a;
u := u0random(stream);
while (u > a) and (a <= upperbound) and
  (b >= lowerbound) do begin
  variate := variate + 1;
  b := (b * lambda) / variate;
  a := a + b;
end;
poisson := variate;
end; (* function poisson *)

procedure c0tran (var e : tr0tran );
begin
  new(e);
  with e do begin
    inqueue := false;
    qhead := nil;
    next := nil;
    prev := nil;
  end;
end;

procedure d0tran (var e:ptr0tran );
begin
  if e^.inqueue then begin
    writeln('*** tried to destroy an entity ',
      'of type tran ');
    writeln(' which is still in a queue ***');
    errorOx;
  end;
e := nil;
end;
procedure i0tranqueue (var q: tranqueue );
begin
   new(q.head);
   with q.head do
   begin
      next := q.head;
      prev := q.head;
      qhead := nil;
      inqueue := false;
   end;
   q.size := 0;
   q.empty := true;
   clear(q.stat, accumulate);
end;

(* 59*)
(* 60*)
(* 61*) procedure r0arrival
   (r0arg: t0arrival ); forward;
(* 62*) procedure r0departure
   (r0arg: t0departure ); forward;
(* 63*) procedure r0dequeue
   (r0arg: t0dequeue ); forward;
(* 64*) procedure r0endcheck
   (r0arg: t0endcheck ); forward;
(* 65*)
(* 66*) procedure r0create (r0arg: t0create );
(* 67*) var found: boolean; i : integer;
(* 68*) newtran : ptr0tran; x:real;
(* 69*) begin with r0arg do
(* 69*) begin
(* 70*) if not incmqueue[s].empty and
   (tranno[s] < maxtranno) then
(* 71*) begin
(* 72*)
(* 72*) begin
(* 72*) newtran := incmqueue[s].head^.next;
(* 72*) with incmqueue[s] do
(* 72*) begin
(* 72*) if newtran^.qhead <> head then error0p(8,72);
(* 72*) r0observe(size,stat); size := size - 1;
(* 72*) empty := size=0; end;
(* 72*) with newtran do
(* 72*) begin
(* 72*) inqueue := false; qhead := nil;
(* 72*) prev^.next := next; next^.prev := prev;
(* 72*) next:=nil; prev:=nil;
(* 72*) end; end
(* 72*) ;
(* 73*) tranno[s] := tranno[s] + 1;
(* 74*) newtran^.checked :=false;
(* 75*) newtran^.siteid := s;
(* 76*) totaltran := totaltran + 1;
(* 77*) if newtran^.tag=0 then begin
(* 78*) localtran[s] := localtran[s] + 1;
(* 79*) newtran^.tag := localtran[s]*100+s;
(* 80*) newtran^.stime := time;
(* 81*) end;
(* 82*) found := false; i:=1;
(* 83*) while (i<=maxtranno) and not found do
(* 84*) begin
(* 85*) if slotav[s][i]=0 then begin
(* 86*) slotav[s][i]:=1;
(* 87*) newtran^.slotid := i;
(* 88*) found := true
(* 89*) end;
(* 90*) i := i+1
(* 91*) end;
(* 92*) sitets[s] := sitets[s] + tsinterval;
(* 93*) newtran^.trid := sitets[s];
(* 94*) for i:=1 to noob do
(* 95*) begin newtran^.lockobtained[i]:=false;
(* 96*) newtran^.nextts[i]:=0
(* 97*) end;
(* 98*) newtran^.nowaits:=0;
(* 99*) newtran^.sd1:=newtran^.sd01;
newtran^.sd2:=newtran^.sd02;
(*100*) seed0v[50]:=newtran^.sd2;
(*101*) x := u0random(50) * noob;
(*102*) newtran^.sd2:=seed0v[50];
(*103*) if x <noob then i:= trunc(x) + 1
else i:= trunc(x);
begin
  c0notice( g0notice, arrival );
  with g0notice do begin
    a0arrival . t := i ;
    a0arrival . request := newtran ;
  end;
  g0notice . etime := time + poisson( servicerate, nosite+s ) * mdelay ;
  e0insert( g0notice, 105 );
  g0notice := nil;
end
end

procedure r0generatran( r0arg: t0generatran );
var
  newtran: ptr0tran; x: real;
begin with r0arg do
  begin
    c0tran(newtran);
    newtran . sd01 := seed0v[freeseed+s];
    newtran . sd02 := seed0v[freeseed+nosite+s];
    x := u0random(freeseed+s);
    x := u0random(freeseed+nosite+s);
    newtran . tag := 0;
    begin
      if newtran . inqueue then error0p(9, 121);
      with newtran do
      with incmqueue[s] do begin
        qhead := head . prev;
        next := qhead . next; prev := qhead;
        qhead . next := newtran;
        next . prev := newtran;
        inqueue := true; qhead := head;
r0observe(size, stat); size := size + 1;
(*121*) empty := false; end; end
(*121*)
(*122*) begin
(*122*) c0notice(g0notice, generatran);(*122*)
(*122*) with g0notice do begin
(*122*) a0generatran.s := s;
(*122*) end;
(*122*) g0notice.evtime := time + poisson(arrivalrate, s);
(*122*) e0insert(g0notice, 122);
(*122*) g0notice := nil;
(*122*) end
(*122*)
(*123*) if tranno[s] < maxtranno then
(*123*) begin
(*123*) c0notice(g0notice, create);
(*123*) with g0notice do begin
(*123*) a0create.s := s;
(*123*) end;
(*123*) g0notice.evtime := time + 0
(*124*)
(*124*) e0insert(g0notice, 124);
(*124*) g0notice := nil;
(*124*) end
(*124*)
(*125*)
(*126*)
(*127*)
(*128*)
(*129*) function greater(tr1, sitel, slot1, tr2, site2, slot2: integer): boolean;
(*130*) begin
(*131*) if tr1 > tr2 then greater := true
(*132*) else if tr1 < tr2 then greater := false
(*133*) else if sitel > site2 then greater := true
(*134*) else if sitel < site2 then greater := false
(*135*) else if slot1 > slot2 then greater := true
else greater := false
end;

procedure rObegincheck (rOarg: tObegincheck );
var item : ptrOtran; i: integer; found: boolean;
begin with rOarg do
  item := objectstatus[t].position;
  if greater(item^.trid, item^.siteid, item^.slotid, objectts[t].trnid, objectts[t].siteid, objectts[t].slotid) then
  begin
    with gOnotice* do begin
      aOendcheck .t := t;
      end;
    gOnotice^.evtime := time + cdelay;
    eOinsert ( gOnotice, 146 );
    gOnotice := nil;
    end
  else begin
    objectstatus[t].chst := wait;
    begin
      with gOnotice* do begin
        aOendcheck .t := t;
        end;
      gOnotice^.evtime := time + cdelay;
      eOinsert ( gOnotice, 149 );
      gOnotice := nil;
      end
    end
  end
end;

procedure rOendcheck;
variablenextOitem, item : ptrOtran;
i, j, max, min : integer;
a, b, c, d, k : integer; found : boolean;
begin with roarg do
begin
item := objectstatus[t].position;
nextOitem := item*.next;
if objectstatus[t].chst = normal then begin item*.checked := true;
objectsts[t].trandid := item*.trid;
objectsts[t].slotid := item*.slotid;
objectsts[t].siteid := item*.siteid;
if not objectslock[t] then begin
conotice(gOnotice, dequeue);
with gOnotice do begin
dequeuet := t;
end;
gOnotice.evtime := time + 0;
e0insert(gOnotice, 165);
gOnotice := nil;
end;
else begin
begin
with objectsts[t] do
begin
if item*.qhead <> head then error0p(8, 167);
roobserve(size, stat); size := size - 1;
empty := size = 0; end;
with item* do
begin
inqueue := false; qhead := nil;
prev*.next := next; next*.prev := prev;
next := nil; prev := nil;
end; end
end;
(*168*) begin
(*168*) c0notice( g0notice, departure );
(*168*) with g0notice* do begin
(*168*) a0departure .request := item ;
(*168*) a0departure .ss := false ;
(*168*) end;
(*168*) g0notice*.evtime := time + mdelay ;
(*168*) e0insert( g0notice, 168 );
(*168*) g0notice := nil;
(*168*) end
(*168*)
(*169*) objectstatus[t].chst := normal;
(*170*)
(*171*) if next0item <> objects[t].head then begin
(*172*) objectstatus[t].position := next0item;
(*173*)
(*173*) begin
(*173*) c0notice( g0notice, begincheck );
(*173*) with g0notice* do begin
(*173*) a0begincheck .t := t ;
(*173*) end;
(*173*) g0notice*.evtime := time + cdelay
(*174*) ;
(*174*) e0insert( g0notice, 174 );
(*174*) g0notice := nil;
(*174*) end
(*174*) end;
(*175*)
(*175*)
(*176*)
(*177*) procedure r0arrival;
(*178*) begin with r0arg do
(*178*) begin
(*179*) request*.checked := false;
(*180*)
(*180*) if request*.inqueue then error0p(9,180);
(*180*) with request* do
(*180*) with objects[t] do begin
(*180*) qhead := head*.prev;
(*180*) next:=qhead*.next; prev:=qhead;
(*180*) qhead^.next:=request;
(*180*) next^.prev:=request;
(*180*) inqueue:=true; qhead:=head;
(*180*) roobserve(sise,stat); sise:=sise+1;
(*180*) empty:=false; end; end

(*181*) if objectstatus[t].chst=normal then
(*182*) if ( objects[t].head = request^.prev ) then
(*183*) begin
(*184*) objectstatus[t].position := request;
(*185*)
(*185*) begin
(*185*) cononotice( gOnotice, begincheck );
(*185*) with gOnotice^ do begin
(*185*) aObegincheck .t := t ;
(*185*)
(*185*) gOnotice^.evtime:= time + cdelay
(*186*) ;
(*186*) e0insert( gOnotice, 186 );
(*186*) gOnotice:= nil;
(*186*) end
(*186*) end
(*187*) else if request^.prev^.checked then begin
(*188*) objectstatus[t].position := request;
(*189*)
(*189*) begin
(*189*) cononotice( gOnotice, begincheck );
(*189*) with gOnotice^ do begin
(*189*) aObegincheck .t := t ;
(*189*)
(*189*) gOnotice^.evtime:= time + cdelay
(*190*) ;
(*190*) e0insert( gOnotice, 190 );
(*190*) gOnotice:= nil;
(*190*) end
(*190*) end
(*191*) end
(*191*)
(*192*)
(*193*) procedure rodeparture;
(*194*) var i : integer;
(*195*) begin with r0arg do
(*195*) begin
(*196*)' trannorequest".siteid| :=
    trannorequest".siteid| - 1;
(*197*) slicev[request".siteid][request".slotid] := 0;
(*198*) if ss then completed:=completed+1
(*199*) else aborted:=aborted+1;
(*200*) for i:=1 to noob do
(*201*) if request".lockobtained[i] then
(*202*) begin
(*202*) c0notice(g0notice, dequeue);
(*202*) with g0notice do begin
(*202*) a0dequeue.t := i ;
(*202*) end;
(*202*) g0notice".evtime:= time + mdelay ;
(*202*) e0insert( g0notice, 202 );
(*202*) g0notice:= nil;
(*202*), end
(*202*);
(*203*) if time > 40 then
(*203*) begin
(*203*) c0notice( g0notice, main );
(*203*) with g0notice do begin
(*203*) a0main .status := normalterm ;
(*203*) end;
(*203*) with g0notice do begin
(*203*) evtime:= time;
(*203*) next:= ev0set.head".next;
(*203*) ev0set.head".next:= g0notice;
(*203*) prev:= ev0set.head;
(*203*) next".prev:= g0notice;
(*203*) inqueue:= true;
(*203*) qhead:=ev0set.head;
(*203*) end; (* with *)
(*203*) with ev0set do begin
(*203*) size := size + 1;
(*203*) empty := false;
(*203*) end; (* with *)
(*203*) g0notice:= nil;
(*203*) end

(*203*)

(*204*) else
(*204*) begin
(*204*) c0notice( g0notice, create );
(*204*) with g0notice* do begin
(*204*) a0create .s := request*.siteid ;
(*204*) end;
(*204*) g0notice*.evtime := time + 0 ;
(*204*) e0insert( g0notice, 204 );
(*204*) g0notice := nil;
(*204*) end
(*204*) end
(*204*) ;
(*205*) if ss then begin
(*206*) r0observe(time-request*.st ime, transtat);  
(*206*) d0tran(request) end
(*207*) else begin
(*208*) if request*.inqueue then errorOp(9,208);
(*208*) with request* do begin
(*208*) with incmqueue(request*.siteid)
(*208*) do begin
(*208*) qhead := head*.prev;
(*208*) next := qhead*.next; prev := qhead;
(*208*) qhead*.next := request;
(*208*) next*.prev := request;
(*208*) inqueue := true; qhead := head;
(*208*) r0observe(sise,stat); sise := sise+1;
(*208*) empty := false; end; end
(*208*) end
(*208*) end
(*208*) ;
(*209*)
(*210*) procedure r0dequeue;
(*211*) var request:ptr0tran;x :real;
(*212*) begin with r0arg do
(*212*) begin
(*213*) if objects[t].empty then begin
(*213*) objectslock[t] := false;
(*214*) b0observe(true, objbusy[t]) end
(*215*) else if objects[t].head*.next*.checked then begin
(*216*)
begin
request := objects[t].head^.next;
with objects[t] do
begin
if request^.qhead <> head then error0p(8,216);
request^.lockobtained[t] := true;
if not objectslock[t] then begin
objectslock[t] := true;
end;
seed0v[50] := request^.sdl;
x := u0random(50);
request^.sdl := seed0v[50];
if x > reqper then begin
begin
0notice(g0notice, departure);
with g0notice^ do begin
a0departure .request := request ;
a0departure .as := true ;
end;
end;
g0notice^.eventime := time +
poisson(servicerate,nosite+request^.siteid)+mdelay
end;
e0insert( g0notice, 227 );
g0notice := nil;
end
else begin
repeat

\[(\ast 229\ast)\] seed0v[50] := request^\_sd2;

\[(\ast 230\ast)\] x := u\_0\_random(50) * noob;

\[(\ast 231\ast)\] request^\_sd2 := seed0v[50];

\[(\ast 232\ast)\] if x < noob then i := trunc(x) + 1

\[(\ast 233\ast)\] else i := trunc(x);

\[(\ast 234\ast)\] until request^\_lockobtained[i] = false;

\[(\ast 235\ast)\] begin

\[(\ast 235\ast)\] c\_notice( g\_notice, arrival );

\[(\ast 235\ast)\] with g\_notice do begin

\[(\ast 235\ast)\] a\_arrival^\_t := i;

\[(\ast 235\ast)\] a\_arrival^\_request := request;

\[(\ast 235\ast)\] end;

\[(\ast 235\ast)\] end

\[(\ast 236\ast)\] g\_notice^\_evtime := time + poisson(servicerate, nosite+request^\_siteid)+ 2^mdelay

\[(\ast 237\ast)\] ;

\[(\ast 237\ast)\] e\_0\_insert( g\_notice, 237 );

\[(\ast 237\ast)\] g\_notice := nil;

\[(\ast 237\ast)\] end

\[(\ast 237\ast)\] end

\[(\ast 238\ast)\] end

\[(\ast 239\ast)\] else begin objects\_lock[t] := false;

\[(\ast 240\ast)\] bo\_observe(true, obj\_busy[t])

\[(\ast 240\ast)\] end

\[(\ast 240\ast)\] end

\[(\ast 240\ast)\] end

\[(\ast 241\ast)\] ;

\[(\ast 242\ast)\] procedure s\_0\_control;

\[(\ast 242\ast)\] label 9999;

\[(\ast 242\ast)\] begin (* s\_0\_control *)

\[(\ast 242\ast)\] with ev\_0\_set do begin

\[(\ast 242\ast)\] current := head^\_next;

\[(\ast 242\ast)\] while current <> head do begin

\[(\ast 242\ast)\] with current^\_do begin

\[(\ast 242\ast)\] time := ev\_time;

\[(\ast 242\ast)\] next^\_prev := prev;

\[(\ast 242\ast)\] end

\[(\ast 242\ast)\] end

\[(\ast 242\ast)\] end

\[(\ast 242\ast)\] end

\[(\ast 242\ast)\] end

\[(\ast 242\ast)\] end
prev^.next := next;
next := nil;
prev := nil;
qhead := nil;
inqueue := false;
end; (* with *)
size := size - 1;
empty := size = 0;
case current^.eventtype of
begincheck
  r0begincheck (current^.a0begincheck);
genetrans
  r0generetrans (current^.a0generetrans);
create
  r0create (current^.a0create);
endcheck
  r0endcheck (current^.a0endcheck);
dequeue
  r0dequeue (current^.a0dequeue);
departure
  r0departure (current^.a0departure);
arriaval
  r0arrivval (current^.a0arrivval);
main: begin
  status := current^.a0main.status;
goto 9999;
end;
end; (* case *)
if (current <> nil) then
  if (not scheduled(current)) and
      (not current^.named) then
d0notice(current);
current := head^.next;
end;
end;
status := 0;
9999:
end; (* s0control *)
begin

time := 0;
new(ev0set.head, no0event);
with ev0set.head^ do begin
  next := ev0set.head;
  prev := ev0set.head;
  qhead := ev0set.head;
evt ime := -1;
inqueue := false;
end; (* with *)
with ev0set do begin
empty := true;
size := 0;
end;

(* set default seeds for LLRANDOM *)
seed0v[1] := 1973272912;
seed0v[2] := 319340905;
seed0v[3] := 687514864;
seed0v[4] := 1638801628;
seed0v[5] := 1928201733;
seed0v[6] := 911320271;
seed0v[7] := 87021785;
seed0v[8] := 88473758;
seed0v[9] := 1868734899;
seed0v[10] := 26784697;
seed0v[11] := 1973272912;
seed0v[12] := 3140905;
seed0v[13] := 6875864;
seed0v[14] := 38801628;
seed0v[15] := 92201733;
seed0v[16] := 910271;
seed0v[17] := 821785;
seed0v[18] := 843758;
seed0v[19] := 18734899;
seed0v[20] := 2697;
seed0v[21] := 19272912;
seed0v[22] := 9340905;
seed0v[23] := 8754864;
seed0v[24] := 163801628;
seed0v[25] := 128201733;
seed0v[26] := 9120271;
seed0v[27] := 871785;
seed0v[28] := 8873758;
seed0v[29] := 18734899;
seed0v[30] := 2784697;
seed0v[31] := 45974321;
seed0v[32] := 3609825;
seed0v[33] := 456943214;
seed0v[34] := 268095438;
seed0v[35] := 8999933;
seed0v[36] := 62091771;
seed0v[37] := 34865425;
seed0v[38] := 98903678;
seed0v[39] := 11118899;
seed0v[40] := 529983188;

seed0v[41] := 67064912;
seed0v[42] := 12384205;
seed0v[43] := 5538104;
seed0v[44] := 17028;
seed0v[45] := 87392;
seed0v[46] := 470211;
seed0v[47] := 7895435;
seed0v[48] := 194297608;
seed0v[49] := 984327810;
seed0v[50] := 764319442;

seed0v[51] := 77064912;
seed0v[52] := 62384205;
seed0v[53] := 2538104;
seed0v[54] := 47028;
seed0v[55] := 57392;
seed0v[56] := 670211;
seed0v[57] := 9895435;
seed0v[58] := 394297608;
seed0v[59] := 584327810;
seed0v[60] := 164319442;

seed0v[61] := 8704912;
seed0v[62] := 1234205;
seed0v[63] := 953104;
seed0v[64] := 1708;
seed0v[65] := 8732;
seed0v[66] := 47011;
seed0v[67] := 789435;
seed0v[68] := 19497608;
seed0v[69] := 98427810;
seed0v[70] := 76419442;
seed0v[71] := 8764912;
seed0v[72] := 124205;
seed0v[73] := 958104;
seed0v[74] := 1728;
seed0v[75] := 8792;
seed0v[76] := 47211;
seed0v[77] := 785435;
seed0v[78] := 19297608;
seed0v[79] := 98327810;
seed0v[80] := 76319442;
seed0v[81] := 7064912;
seed0v[82] := 2384205;
seed0v[83] := 538104;
seed0v[84] := 7028;
seed0v[85] := 7392;
seed0v[86] := 70211;
seed0v[87] := 895435;
seed0v[88] := 94297608;
seed0v[89] := 84327810;
seed0v[90] := 64319442;
seed0v[91] := 6064912;
seed0v[92] := 1384205;
seed0v[93] := 938104;
seed0v[94] := 1028;
seed0v[95] := 8392;
seed0v[96] := 40211;
seed0v[97] := 795435;
seed0v[98] := 14297608;
seed0v[99] := 94327810;
seed0v[100] := 64319442;

(*243*) readln(mdelay,requer);
(*244*) totaltran := 0;
(*245*) completed := 0;
(*246*) aborted := 0;
(*247*) clear(transtat,tally);
(*248*) for i := 1 to noob do
clear(objbusy[i], accumulate);
for i := 1 to nosite do begin
  i0tranqueue(incmqueue[i]);
  localtran[i] := 0;
  tranno[i] := 0;
for j := 1 to maxtranno do
  slotav[i][j] := 0;
sitets[i] := 0
end;
for i := 1 to nosite do
  i0tranqueue(objects[i]);
objectstatus[i].chst := normal;
  objectstatus[i].waited := 0;
objectsluck[i] := false;
objectts[i].transid := 0;
objectts[i].slotid := 0;
objectts[i].siteid := 0
end;
for i := 1 to nosite do
begin
  c0notice( g0notice, generatran );
  with g0notice do begin
    a0generatran .s := i ;
    end;
  g0notice^.evtime := time +
      poisson(arrivalrate, i) ;
e0insert( g0notice, 268 );
g0notice := nil;
end;
end;
s0control( status);
writeln('Timestamping');
writeln('mdelay=',mdelay:6:2);
writeln('reqper=',reqper:6:2);
writeln('total no. of transactions generated:',
totaltran:5);
writeln('total completions:',completed:5);
writeln('total aborts:',aborted:5);
writeln('mean response time:',transtat.mean:10);
writeln('ratio:',completed * 100/totaltran :10 );
end.
REFERENCES


[2] Badal, D.
Correctness of Concurrency Control and Implications in Distributed Databases.

A Message-Based Approach to Discrete-Event Simulation.

Formal Aspects of Serializability in Database Concurrency Control.

[5] Bernstein, A.
Output Guards and Nondeterminism in "Communicating Sequential Processes".
ACM Transactions on Programming Languages and Systems 2(2):234-238, April, 1980.

Concurrency Control in Distributed Database Systems.
Concurrency Control for Multiversion Database Systems.  


Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems.  

[10] Bryant, R.  
*SIMPAS-A Simulation Language based on PASCAL*.  

An Effective Implementation for the Generalized Input-Output Construct of CSP.  

[12] Carey, M. J.  
An Abstract Model of Database Concurrency Control Algorithms.  

[14] Carey, M.
Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions.

*Distributed Databases: Principles & Systems.*

[16] Date, C.
*An Introduction to Database Systems.*
Addison-Wesley, 1983.

[17] Dijkstra, E. W.
Guarded Commands, Nondeterminacy and Formal Derivation of Programs.


[19] Elrad, T. and Maymir-Ducharme, F.
Distributed Language Design: Constructs for Controlling Preferences.

What is a "Distributed Data Processing System"?

The Notions of Consistency and Predicate Locks in a Database System.

[22] Filman, R. E. and Friedman, D. P.
*Coordinated Computing.*
[23] Garcia-Molina, H.
Using Semantic Knowledge for Transaction Processing in a DDB.

The Performance of a Concurrency Control Mechanism that Exploits Semantic Knowledge.

"Notes on Database Operating Systems", in *Operating System: An Advanced Course*.
Springer-Verlag, 1979.

[26] Hoare, C.A.R.
Communicating Sequential Processes.

[27] Kieburtz, R. B. and Silberschatz, A.
Comments on Communicating Sequential Processes.

[28] Kohler, W. H. and Jenq, B-C.

[29] Kohler, W., Towsley, D. and Jenq, B-C.
A Queueing Network Model for a Distributed Database Testbed System.

On Optimistic Methods for Concurrency Control.

[31] Lamport, L.
Time, Clocks and the Ordering of Events in a Distributed System.
[32] LeBlanc, R. and Wiles, C.
Systems Programming with Objects and Actions.

[33] Moss, J.
*Nested Transactions: An Approach to Reliable Distributed Computing.*

[34] Natarajan, N. and Sinha, M.
A Priority Based Distributed Deadlock Detection Algorithm.

[35] Natarajan, N.
A Distributed Scheme for Detecting Communication Deadlocks.

[36] Papadimitriou, C.
*Serializability of Concurrent Database Updates.*

[37] Rahm, Erhard.
Integrated Solutions to Concurrency Control and Buffer Invalidation in Database Sharing Systems.

[38] Reed, D.
Implementing Atomic Actions on Decentralized Data.

System Level Concurrency Control for Distributed Database Systems.
[40] Russel, D.
State Restoration in Systems of Communicating Processes.

[41] Sheth, A. P.
*Adaptive Concurrency Control for Distributed Database Systems.*

[42] Shum, A. and Spirakis, P.
Performance Analysis of Concurrency Control Methods in Database Systems.
*Performance '81.*

[43] Silberschatz, A.
A Multi-Version Concurrency Scheme With No Rollbacks.

[44] Silberschatz, A.
A Case for Non-two-phase Locking Protocols that Ensure Atomicity.

[45] Singhal, Mukesh.
*Concurrency Control Algorithms and their Performance in Replicated Database Systems.*

[46] Sinha, M.
Commutable Transactions and the Time_pad Synchronization Mechanism for Distributed Systems.

[47] Sugihara, K., Kikuno, T., Yoshida, N. and Ogata, M.
A Distributed Algorithm for Deadlock Detection and Resolution.
[48] Thomas, R.
A Solution to the Concurrency Control Problem for Multiple Copy Data Bases.

[49] Thomas, R.
A Majority Consensus to Concurrency Control for Multiple Copy Databases.

[50] Tripathi, A. and Wang, P.
An Object-Oriented Design Model for Reliable Distributed Systems.

[51] Ullman, J.
*Principles of Database Systems*.

[52] Verhofstad, J.
Recovery Techniques for Database Systems.