A STUDY OF THE AVAILABILITY AND SERIALIZABILITY IN A DISTRIBUTED DATABASE SYSTEM

by

David Wai-Lok Cheung

B.Sc., Chinese University of Hong Kong, 1971

M.Sc., Simon Fraser University, 1985

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in the School

of

Computing Science

© David Wai-Lok Cheung 1988

SIMON FRASER UNIVERSITY

January 1988

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without permission of the author.

APPROVAL

Name: David Wai-Lok Cheung

Degree: Doctor of Philosophy

Title of Thesis: A Study of the Availability and Serializability in a Distributed Database System

Examining Committee:

Chairperson: Dr. Binay Bhattacharya

Senior Supervisor: Dr. Tiko Kameda

Dr. Arthur Lee Liestman

Dr. Wo-Shun Luk

Dr. Jia-Wei Han

External Examiner: Toshihide Ibaraki
Department of Applied Mathematics and Physics
Kyoto University, Japan

Date Approved: January 15, 1988

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

A STUDY OF THE AVAILABILITY AND SERIALIZABILITY ~~OF~~ IN A DISTRIBUTED DATABASE SYSTEM

Author: ______________________

(signature)

David Wai-Lok CHEUNG

(name)

Jan 22, 88

(date)

ABSTRACT

Replication of data objects enhances the reliability and availability of a distributed database system. However, due to the inherent conflict between serializability and availability, if serializability is to be guaranteed in a partitioned database system, degradation of availability is inevitable. We first characterize serializable transaction executions in a partitioned database system, by means of a graph theoretical method. We then derive an upper bound on the availability of a database system with two partitions. This upper bound holds for a general class of transaction distributions that satisfy the "weak uniformity assumption".

Since it is impossible to simultaneously achieve serializability and high availability in a general database system, we investigate database systems in which constraints are imposed on the read/write activity of the transactions. In particular, we propose a fragmented database system, in which transactions are classified as either local or global. This model can be used to realize wide-area distributed database systems, in which messages encounter substantial communication delays. We argue that a transaction scheduling policy that favors local transactions over global transactions should be adopted in wide-area distributed database systems. Two schemes are proposed for concurrency control in this kind of system.

The first scheme, Global Timestamp Order Certification, uses an "active" approach, and sends out requests for global transactions to be certified by remote sites. The second scheme, Global Timestamp Order Synchronization, adopts a "passive" approach in which a global transaction is made to wait until it is known that the transaction has read consistent data object values from other sites. In both approaches, no global transaction blocks a local transaction. Moreover, local transactions are executed as if they were in the single-site environment, in which communication delays are negligible.

ACKNOWLEDGEMENTS

I wish to express special appreciation and gratitude to Professor Tiko Kameda, my senior supervisor, for his constant guidance, advice, support and availability. Without his support and supervision which I enjoyed during the last four years of my study at Simon Fraser, the completion of this thesis would not have been possible. His insistence on precision and clarity in writing has been immensely helpful in the preparation of this thesis.

I am also indebted to Dr. Wo-shun Luk, Dr. Arthur Liestman and Dr. Jia-wei Han, not only for their reading and commenting on this thesis, but also for their constant advice and encouragement. Thanks are also due to Professor Toshihide Ibaraki, the external examiner of this thesis, for his valuable and stimulating suggestions and comments.

My thanks go also to many of the graduate students in the School of Computing Science at Simon Fraser, who either read and commented parts of my thesis, or provided me with various kinds of help when they were most needed: Ada Fu, Steven Yap, Mimi Kao, Frank Tong, Patta Pattabhiraman, Siu-Cheung Chau, Wuyi Wu and many others.

Finally, this work is dedicated to my wife, Juana. She gave me love and support, and encouraged my endeavour to study at an age when most men have to work. This thesis is also dedicated to my parents, brothers and sisters. They have been caring so much about me, my wife and my children.

TABLE OF CONTENTS

Approval ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Figures viii

Chapter One. INTRODUCTION 1

1.1. Replicated Database Management 1

1.2. Goals of the thesis 2

1.3. Overview of the thesis 3

Chapter Two. CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEM 6

2.1. Concurrency Control 6

2.2. Serializability in a Single Site Database System 7

2.3. Disjoint-Interval Topological Sort 11

2.4. Serializability in a Distributed Database System 13

Chapter Three. COPING WITH PARTITION FAILURES 18

3.1. Partition Failure 18

3.2. Pessimistic Approach to Dealing with Partition Failures 20

3.3. Optimistic Approach to Dealing with Partition Failures 24

3.4. Transaction-Cluster Log 26

3.5. Characterization of Executions Admissible under Partition Failures 29

Chapter Four. LIMITATION ON AVAILABILITY 38

4.1. Availability in Partitioned Database 38

4.2. Uniform Transaction Distribution and Availability 42

4.3. A General Upper Bound on Availability 44

Chapter Five. TECHNIQUES FOR ACHIEVING HIGH AVAILABILITY 49

5.1. Trade-off between Serializability and Availability 49

5.2. A Highly Available Distributed Database System 50

5.3. Fragmented Distributed Database System 52

5.4. Transaction Processing with a Static Read Access Graph 56

Chapter Six. A CONCURRENCY CONTROL SCHEME FOR WIDE-AREA DISTRIBUTED DATABASE SYSTEM 63

6.1. A Model for Wide-Area Distributed Database Systems 63

6.2. Architecture for Wide-Area Distributed Database System (WADDS) 69

6.3. Correctness of Fragmented Executions 74

6.4. An Algorithm to Control Fragmented Execution 86

6.5. Performance Analysis 102

6.6. Partition Failures in a Fragmented Database System 104

Chapter Seven. ANOTHER CONCURRENCY CONTROL SCHEME FOR FRAGMENTED EXECUTION 109

7.1. Another Scheme to Control Fragmented Execution 109

7.2. An Architecture for GTOS 111

7.3. Timestamp and Virtual Clock Management 116

7.4. Global Timestamp Ordering Synchronization Algorithm 125

7.5. Correctness of GTOS 128

7.6. Performance of GTOS 130

Conclusion 134

References 136

LIST OF FIGURES

Figure 2.1.1 Illustration for Example 2.1.1. 140

Figure 2.2.1 Transaction Read-from graphs. 141

Figure 2.3.1 TIO graph and DITS. 142

Figure 2.4.1 Translation of transactions into posets of physical operations. 143

Figure 2.4.2 A rp log. 144

Figure 2.4.3 TIO graph of a rp log L and DITS. 144

Figure 3.1.1 Rp log L and TIO(L). 145

Figure 3.2.1 Illustration for Example 3.2.1. 145

Figure 3.3.1(a-d) Illustration for Example 3.3.1. 146

Figure 3.3.1(e-f) Illustration for Example 3.3.1. 147

Figure 3.4.1 Illustration for Example 3.4.1. 148

Figure 3.4.2 Illustration for Example 3.4.2. 149

Figure 3.5.1 Illustration for Examples 3.5.1 and 3.5.2. 150

Figure 3.5.2 An illustration for the Non-Selective Assumption. 151

Figure 4.1.1 Acceptance Ratio. 152

Figure 4.2.1 Two PIO graphs. 153

Figure 4.3.1 Two DITS's for PIO(L). 154

Figure 4.3.2 Inclusion relationships. 155

Figure 5.2.1 Architecture of SHARD. 156

Figure 5.3.1 A fragmented database system. 157

Figure 5.4.1 Illustration for Example 5.4.1. 158

Figure 5.4.2(a-c) Three RAG's. 159

Figure 5.4.2(d) A RAG. 160

Figure 5.4.3 RAG of Example 5.4.3. 160

Figure 5.4.4(a-b) Illustration for Example 5.4.4. 161

Figure 5.4.4(c-d) Illustration for Example 5.4.4. 162

Figure 5.4.5 RAG for a fragmented database. 163

Figure 6.1.1 Illustration for Example 6.1.1. 164

Figure 6.1.2 (a-b) Illustration for Example 6.1.2. 165

Figure 6.1.2 (c-d) Illustration for Example 6.1.2. 166

Figure 6.2.1 An architecture for a single-site database. 167

Figure 6.2.2 An architecture for GTOC. 168

Figure 6.3.1 (a-b) Precedence edge and global-read edge. 169

Figure 6.3.1 (c-d) Two kinds of induced edges. 170

Figure 6.3.2 A GOS graph. 171

Figure 6.4.1 A GOS graph. 172

Figure 6.4.2 A GOS graph. 172

Figure 6.4.3 Deadlock. 173

Figure 7.2.1 An architecture for GTOS. 174

Figure 7.2.2 Compatibility among h-locks and l-locks. 175

Figure 7.3.1 Update and timeout messages. 176

Figure 7.3.2 Deadlock and a remedy. 177

Figure 7.3.3 Global-start messages and global-completion messages. 178

Figure 7.4.1 Time Chart of Transaction Execution. 179

CHAPTER 1

INTRODUCTION

1.1. Replicated Database Management

Replication of a database in terms of a backup copy has long been used as a means to achieve higher reliability. The backup copy is not normally used when the master copy is accessible; it is used only when a disastrous failure has destroyed the master copy.

In a distributed database system, replication provides an additional advantage besides reliability, i.e., availability. In this thesis, we define availability as 1 minus the fraction of transactions that cannot be executed due to inaccessibility of some data objects, caused by communication failure. (Availability will be formally defined in Chapter 4.) If there are multiple copies of a data object located at different sites, this data object can be quickly accessed locally, though it cannot always be updated.

Unfortunately, replication is not cheap. Primarily, there are two kinds of costs associated with replication. The first is incurred in maintaining (storage) and updating multiple copies. This cost is intrinsic and cannot be avoided. The second are delays incurred in the concurrency control of distributed transaction execution, in other words, in synchronizing read/write operations at different sites to ensure some kind of correctness.

If the possibility of failure is taken into account, the issue of concurrency control becomes much more complicated. The worst kind of failure, as far as concurrency control is concerned, is

network partitioning. When a network partitions, sites in different partitions cannot communicate with each other. A site in a partition has no knowledge about the activity in the other partitions. Therefore, it is no longer possible to synchronize the operations at all the sites through communication. As a result, the activity within a partition must be restricted in order to maintain global serializability for an execution which consists of operations executed in more than one partition. This restriction degrades availability.

1.2. Goals of the thesis

The first goal of this thesis is to study the availability of a replicated database under partition failures. Much work has been done in designing concurrency control schemes that achieve good availability for distributed database systems which are prone to partitioning failures [ASC85, AbT86, Dav84, Gif79, MiW82, SkW84]. However, to the best of our knowledge, no work has been done to characterize transaction executions admissible under a partition failure. In this thesis, we first attempt a graph theoretical characterization for such executions. Then we will characterize the executions over a partitioned database system generated by an important class of protocols, called **prevention protocols**. We shall also derive a theoretical upper bound on the availability of a replicated database system under partition failures. It will be demonstrated that the upper bound is achievable in some cases.

In a replicated database system, availability and serializability appear to be two conflicting goals. An approach taken by several researchers is to increase availability by abandoning the correctness criterion of serializability [BGR83, BlK85, GAB83, LBS86, SBK85, Sar86]. However, this approach introduces a new difficulty. It becomes difficult to specify the correctness of an execution. In some cases, a compensation of incorrect execution necessitates cascading rollbacks.

Without giving up the generality of the database system model under consideration, it is unlikely that high availability and serializability can be achieved simultaneously. For this purpose, a restricted model, the **fragmented database system**, has been proposed by Garcia-Molina and Kogan [GaK87]. This model, though not general, is applicable to many real-life situations. The significance of their work is that it demonstrates a possible way to achieve both high availability and serializability by restricting the activity allowed in a transaction. They have designed some concurrency control algorithms which use a fixed "access pattern". In particular, they do not allow any cycle in the directed graph representing the access pattern. In this thesis, we shall propose algorithms to solve a more general case where the access pattern can be any directed graph. Also, the algorithms proposed in this thesis are dynamic in the sense that it is not necessary to fix the access pattern prior to an execution of the algorithms. These algorithms are particularly useful when applied to a wide-area distributed database, in which communication delays are significant. (Chapter 5).

1.3. Overview of the thesis

In Chapter 2, we will first illustrate an anomaly in a database system, if concurrency control is not provided. Then a correctness criterion, **serializability**, for an execution of transactions in a single-site database system will be defined. We will show that the serializability of an execution can be characterized by its **transaction IO graph**, which is a directed graph containing information on the read-from relation among transactions in the execution. We will also generalize this characterization to a distributed database system.

In Chapter 3, we will study the characteristics of an execution generated in a partitioned database system. We will show that an execution generated by a **prevention protocol** in a

partitioned database system can be characterized by its **partition IO graph**, which is a modification of its transaction IO graph.

In Chapter 4, we will formally define the availability for a **transaction distribution**. Then we will discuss an upper bound on the availability for a transaction distribution found in [COK86]. The transaction distributions studied in [COK86] have to satisfy the **uniformity assumption**. We propose a more general assumption, the **weak uniformity assumption**. We will also derive an upper bound on the availability of transaction distributions which satisfy this assumption. In the derivation of this upper bound, the characteristics of the partition IO graph found in Chapter 3 are used.

Since there is an upper bound on the availability of a transaction distribution, it is not possible to achieve both serializability and full (100%) availability in a distributed database system which is susceptible to partition failures. A new approach to solve this dilemma is to restrict the behaviour of the transactions submitted to a distributed database system. In Chapter 5, a new model, **fragmented database system**, will be described along this line. We will discuss two schemes proposed in [KoG87], which achieve both serializability and high availability in a fragmented database system. These two schemes adopt fixed "access patterns" which are rather restrictive.

In Chapter 6, we will propose a concurrency control scheme for a fragmented database system, in which the access pattern can be any directed graph. Since the condition on the access pattern is relaxed, this scheme is more general then those proposed in [KoG87]. We will show that this scheme can be applied to wide-area distributed database systems, for which conventional concurrency control schemes are not feasible because of large communication delays. This concurrency control scheme is called "Global Timestamp Order Certification". This scheme is an "active" scheme in the sense that timestamps of data objects read by a transaction at a site are sent to other sites for certification. It is basically a distributive protocol in which the values read by a transaction are certified by remote sites.

In Chapter 7, another scheme for concurrency control for a fragmented database is proposed. This scheme is called "Global Timestamp Order Synchronization". In this scheme, transactions are serialized by their timestamps. It is a "passive" scheme in the sense that a transaction does not send out requests to other sites but just waits until it is known that all the values the transaction has read are "correct".

CHAPTER 2

CONCURRENCY CONTROL IN DISTRIBUTED DATABASE SYSTEMS

2.1. Concurrency Control

Concurrency control is the activity of coordinating concurrent accesses to a database. While several users access a database concurrently, each should have the illusion that he or she is executing alone on a dedicated system. The main technical difficulty in attaining this goal is how to prevent database updates performed by one user from "interfering" with the database retrievals and updates performed by others.

Let us use an example to illustrate "interference", which happens because concurrent accesses to a database are not properly controlled.

Example 2.1.1. We consider as an example an on-line electronic funds transfer system. Suppose that two customers A and B are simultaneously accessing their joint accounts by executing the following two transactions.

Customer A (who wants to transfer $1000 from the savings account to the checking account) : reads the balance in the savings account, subtracts one thousand dollars from it, writes the updated balance back to the savings account, reads the balance in the checking account, adds one thousand dollars to it, and writes the updated balance back to the checking account.

Customer B (who wants to print the total balance in the savings and checking accounts) : reads the savings account, reads the checking account, and prints the total of the two readings.

In the absence of concurrency control, these two transactions could interfere with each other as shown in Figure 2.1.1. A's transaction reads the savings account balance, subtracts $1000 and writes the update back into the database. Next, B's transaction reads the balances in the savings and checking accounts and prints out the total. Then A's transaction finishes the funds transfer by reading the checking account, adding $1000, and finally storing the update into the database. Even though the final values in the two accounts are correct, B's transaction prints an incorrect total, which is $1000 short. This is surely not acceptable. □

The above example does not exhaust all possible ways in which concurrent users can interfere with each other. However, it explains why concurrency control is necessary in database management.

In Section 2.2, a correctness criterion, **serializability**, for concurrency control in a single-site database system is introduced. In Section 2.3, a **transaction IO graph (TIO graph)** is used to characterize the serializability of an execution. It is shown in [IKM87] that the transaction IO graph of a serializable execution must have a **disjoint interval topological sort (DITS)**. In Section 2.4, the notions of TIO graph and DITS are extended to distributed database systems and a result similar to the one in [IKM87] is derived.

2.2. Serializability in a Single Site Database System

Serializability theory deals with the correctness of a set of concurrently executing transactions. It also provides a guiding principle for designing concurrency control algorithms [BSW79, BeG81, Cas81, IKM87, Pap79]. In this section, we briefly review serializability theory, based on the work contained in [IKM87].

A database system consists of a set D of **data objects** and a set of transactions $\mathbf{T} = \{T_0, T_1, T_2, \ldots, T_f\}$. T_0 and T_f are two fictitious transactions called the **initial transaction** and the **final transaction**, respectively. Transaction T_0 is a write only transaction that "writes" all the data objects in D before any other transaction starts, and T_f is a read-only transaction which "reads" all data objects after all the other transactions have completed. A **read operation** $R_i[X]$ of transaction T_i returns a value of data object X, and a **write operation** $W_i[X]$ of transaction T_i updates the value of X. Sometimes, we use $R_i[A]$, where $A \subseteq D$, to represent a set of read operations $\{R_i[X] \mid X \in A\}$. Similarly, $W_i[A]$ is used to represent a set of write operations $\{W_i[X] \mid X \in A\}$. Abbreviations such as $R_i[X, Y]$ for $R_i[\{X, Y\}]$ and $W_i[X, Y]$ for $W_i[\{X, Y\}]$ are used in the following.

The execution of a transaction $T_i \in \mathbf{T}$ is modeled by a totally ordered set $T_i = (\Sigma_i, <_i)$, where Σ_i is the set of read and write operations issued by transaction T_i, and $<_i$ is a total order on Σ_i, representing the order in which these operations are executed.

For a set of transactions $\mathbf{T} = \{T_0, T_1, T_2, \ldots, T_f\}$, where $T_i = (\Sigma_i, <i)$, $i = 0, 1, \ldots, f$, let $\Sigma(\mathrm{T})$ denote the set of all the read and write operations of the transactions in **T**. A **dbs log** (or simply a **log, history** or **execution**) over **T** represents an execution of $T_0, \ldots, T_f$. More formally, a log over **T** is a totally ordered set $L = (\Sigma(\mathrm{T}), <)$, where (1) $\Sigma(\mathrm{T}) = \bigcup_{i=0}^{f} \Sigma_i$; (2)$\bigcup_{i=0}^{f} <_i \subseteq <$; (3) for every $A[X] \in \Sigma_0$ and every $B[Y] \in \Sigma(\mathrm{T}) - \Sigma_0$, $A[X] < B[Y]$ holds; and (4) all pairs of conflicting operations in $\Sigma(\mathrm{T})$ are $<$ related. (Two operations on the same data objects are said to **conflict**, if one of them is a write operation). In order to represent the total order $<$, we simply write the operations from left to right in the order of $<$. For two operations A and B in $\Sigma(\mathrm{T})$, we say that A **precedes** B in L if $A < B$.

Let L be a log over T. Transaction T_j **reads X from** transaction T_i if (1) $W_i[X]$ and $R_j[X]$ are in $\Sigma(\mathrm{T})$; (2) $W_i[X] < R_j[X]$; and (3) no $W_k[X]$ falls between these two operations with respect to $<$. Two logs L and L' are said to be **equivalent**, written $L \equiv L'$, if for each data object X and indices i

and j, transaction T_j reads X from transaction T_i in L if and only if T_j reads X from T_i in L'.

A **serial log** is a log such that for every pair of transactions T_i and T_j, either all of T_i's operations precede all of T_j's, or vice versa. A log L is **serializable**, if there exists a serial log L' such that $L \equiv L'$. **SR** denotes the set of all serializable logs. We use $[T_1 T_2 \cdots T_n]$ to denote the serial log in which all the operations of transaction T_i are clustered before all the operations of T_{i+1}, for $i = 1, 2, \cdots, n-1$.

Example 2.2.1. The following log L_1 is clearly serial.

$$L_1 = W_0[X,Y]R_1[X]W_1[Y]R_2[X]W_2[X,Y]R_3[X]W_3[Y]R_f[X,Y].$$

Consider a non-serial log L_2:

$$L_2 = W_0[X,Y]R_1[X]R_2[X]W_2[X,Y]R_3[X]W_1[Y]W_3[Y]R_f[X,Y].$$

In L_2, operation $W_1[Y]$ is executed after an operation, i.e., $R_3[X]$, of T_3. However, the delayed execution of $W_1[Y]$ in L_2 compared with L_1 does not affect the read-from relation between any two transactions in the log. It is easy to see that the two logs L_1 and L_2 are equivalent and therefore L_2 is serializable. □

In order to represent the read-from relation implied by a log, a directed graph is constructed. Let $L = (\Sigma(\mathbf{T}), <)$ be a log over a set **T** of transactions. The **transaction read-from graph (TRF graph,** for short) [IKM87] for L, denoted by $TRF(L)$, has a node set **T** and an arc set **A**. If a transaction T_j reads X from T_i in L, an arc $(T_i, T_j) \in \mathbf{A}$ labeled by X is introduced. This arc is denoted by $(T_i, T_j){:}X$. There are no other nodes or arcs in $TRF(L)$. If follows immediately from definition that two logs L_1 and L_2 are equivalent if and only if $TRF(L_1) = TRF(L_2)$.

Example 2.2.2. Consider a serial log L:

$$L = W_0[X, Y]R_1[X]W_1[Y]R_2[X]R_3[Y]W_3[Y]R_f[X, Y].$$

Figure 2.2.1(a) shows $TRF(L)$. □

An **interval** of a TRF graph is a set of all arcs that have the same label and originate from the same node. For example, the three arcs, $(T_0, T_1){:}X$, $(T_0, T_2){:}X$ and $(T_0, T_f){:}X$ form an interval in Figure 2.2.1(a). The arc $(T_1, T_3){:}Y$ by itself is an interval.

In Figure 2.2.1(a), we have intentionally placed the nodes in the serial order implied by L. It is observed that the intervals in $TRF(L)$ which have the same label do not "overlap". For example, the interval labeled by X spans from T_0 to T_f, (i.e., the longest arc in this interval goes from T_0 to T_f), and it is the only interval labeled by X. There are two intervals labeled by Y. The first one spans from T_1 to T_3, the second one spans from T_3 to T_f, and these two intervals are disjoint. (Two intervals are disjoint, if the sets of nodes ordered between the first and the last nodes, inclusively, of the two intervals contain at most one node in common). It is generally true that the TRF graph of any serial log, with their nodes ordered according to the order of the transactions in the log, has no overlapping interval with the same label [IKM87].

However, it is noted that the converse is not true in general. For example, log L_3 in the following example is not serializable, although the nodes in $TRF(L_3)$ can be arranged linearly so that there is no overlapping interval with the same label in the resulting arrangement.

Example 2.2.3. The TRF graph, $TRF(L_3)$, for

$$L_3 = W_0[X]R_1[X]W_3[X]W_1[X]R_2[X]W_2[X]R_f[X],$$

is shown in Figure 2.2.1(b). With the order of the nodes given in Figure 2.2.1(b), it is observed that no two intervals with the same label overlap. However, from the following observation, it can be seen that L_3 is not serializable. T_2 reads X from T_1; therefore, T_1 must be serialized before T_2. Hence, only the serial logs $[T_0T_3T_1T_2T_f]$, $[T_0T_1T_3T_2T_f]$ and $[T_0T_1T_2T_3T_f]$ need be considered. It

can be checked that none of these three logs is equivalent to L_3. Hence L_3 is not serializable. □

It turns out that the non-serializability of L_3 can be detected if its TRF graph is slightly modified. In a log, a write operation which creates a value that is never read by any other transaction is said to be **useless**. For the useless write $W_3[X]$ in Example 2.2.3, if a dummy node T_3' and a dummy arc $(T_3, T_3'):X$ are added to the $TRF(L_3)$, then there always exist two overlapping intervals labeled by X, no matter what serial order is used to arrange the nodes. (See Figure 2.2.1(c)). This observation is formalized and used in the next section to state a necessary and sufficient condition for an execution to be serializable.

2.3. Disjoint-Interval Topological Sort

As was explained in the last section, the notion of TRF graph has to be extended in order to define a necessary and sufficient condition for a log to be serializable.

Definition 2.3.1 [IKM87]. Let $L = (\Sigma(\mathbf{T}), <)$ be a log over a set **T** of transactions. The **transaction IO graph** (**TIO graph**, for short) for L, denoted by $TIO(L)$, is an arc-labeled directed graph with the node set $\mathbf{T} \cup \mathbf{T}'$, where $\mathbf{T}'$ consists of **dummy nodes** as defined below, and the arc set **A**. If T_j reads X from T_i, there is an arc $(T_i, T_j) \in \mathbf{A}$ labeled by X. If $W_i[Y]$ is a useless write, then we introduce a dummy node $T_i' \in \mathbf{T}'$ together with a dummy arc from T_i to T_i' labeled by Y. There is no other node or arc in $TIO(L)$. □

Example 2.3.1. Consider log L_2 in Example 2.2.1 again. The TIO graph of L_2, $TIO(L_2)$, is shown in Figure 2.3.1(a). Note that there are three useless writes, $W_0[Y]$, $W_1[Y]$ and $W_2[Y]$. Because of these useless writes, three dummy nodes T_1', T_2' and T_3' have been introduced. □

For any two logs L_1 and L_2, $TRF(L_1) = TRF(L_2)$ if and only if $TIO(L_1) = TIO(L_2)$. Hence, the equivalence of logs can also be tested by their TIO graphs.

Example 2.3.2. The TIO graph of L_2 in Figure 2.3.1(a) can be rearranged as in Figure 2.3.1(b), in which no two intervals with the same label overlap. In fact, the first interval of X spans from T_0 to T_2 and the second interval of X spans from T_2 to T_f. So, these two intervals do not overlap. Similarly, the intervals labeled by Y are all disjoint. □

The property of disjoint intervals is used below in a characterization of serializable executions.

Definition 2.3.2 [IKM87]. Let L be a log. A total ordered $\ll$ on the set of nodes of $TIO(L)$ is a **disjoint-interval topological sort** (**DITS**, for short), if it satisfies the following two conditions:

(1) if $T_i \ll T_j$, then there is no path from T_j to T_i in $TIO(L)$, and

(2) if $T_h \ll T_k$ and there are two arcs labeled by X from T_h to T_i and T_j to T_k in $TIO(L)$ $(j \neq h)$, then $T_i \ll T_j$. □

Intuitively, if $TIO(L)$ has a DITS, then it can be ordered linearly from left to right by the order $\ll$, so that all the arcs are directed from left to right (condition (1)), and no two intervals labeled with the same data object overlap (condition (2)). The following theorem characterizes a serializable execution.

Theorem 2.3.1 [IKM87]. *A log L is serializable if and only if $TIO(L)$ has a DITS which orders T_0 first and T_f last.* □

Example 2.3.3. Consider log L_2 of Example 2.2.1 again. $TIO(L_2)$ has a DITS as shown in Figure 2.3.1(b). Therefore L_2 is serializable and the order of the transactions in the serial log implied by the DITS is T_0, T_1, T_2, T_3, T_f. □

Theorem 2.3.1 characterizes a serializable log in a single-site database system (sometime called centralized database system). This result will be generalized to the distributed case in the next section.

2.4. Serializability in a Distributed Database System

In a distributed database system, data objects are often replicated at different sites. The copy of a data object X at site i is denoted by X_i. A data object and its copies are called **logical data object** and **physical data objects,** respectively. In general, when we use the term "data object", we refer to a logical data object. When we use the term "copy", we refer to a physical data object. When a transaction T_i executes, the system uses a translation function τ_i to translate logical operations into **physical operations**, i.e., $W_i[X]$ is translated to $W_i[X_a]$, $W_i[X_b]$, $\cdots$, $W_i[X_l]$, where X_a, $\cdots$, X_l are *some* copies of X and $R_i[X]$ is translated into $R_i[X_e]$ for some copy X_e of X. (Normally, $W_i[X]$ is translated to physical write operations on all the copies of X. However, in some cases, only a subset of all the copies are written. For example, under a partition failure, a write operation $W_i[X]$ may be translated to physical write operations on the subset of accessible physical data objects representing X). Under translation τ_i, transaction T_i is mapped to a partially order set (poset, for short) of physical operations in $\tau_i(\Sigma_i)$, where Σ_i is the set of logical operations in T_i. For a logical operation A in Σ_i, the operations in $\tau_i(A)$ are said to be **associated** with each other as well as to A.

Example 2.4.1 In Figure 2.4.1(a), a transaction $T_1 = R_1[X]R_1[Z]W_1[X]$ is translated into a poset of operations executed distributively at three different sites S_a, S_b and S_e. Similarly, transactions $T_2 = R_2[X]W_2[Y]$ and $T_3 = R_3[Z]W_3[Y]W_3[Z]$ are translated into two posets of operations shown in Figure 2.4.1(b) and (c), respectively. □

The execution of a set of transactions in a distributed database system with replicated data objects can be modeled by a **replicated log** [BeG81]. A **replicated log** (or a **rp log**, for short) over a set **T** of transactions $\{T_i = (\Sigma_i, <_i)\}$ is a poset $L = (\Sigma(\mathbf{T}), <)$ such that (1) $\Sigma(\mathbf{T}) = \bigcup_{i=0}^{f} \tau_i(\Sigma_i)$, where τ_i is the translation function for T_i; (2) for each i and any two operations p_i and q_i in Σ_i, if $a \in \tau_i(p_i)$, $b \in \tau_i(q_i)$, $p_i <_i q_i$ and the physical copies accessed by a and b belong to the same site, then $a < b$;

(3) all pairs of conflicting physical operations are < related (two physical operations conflict if they operate on the same physical copy of a data object and at least one of them is a write operation); and (4) as in a centralized database, **T** contains two fictitious transactions T_0 and T_f. T_0 is translated to a set of write operations, one for each copy of each object, and it precedes all other operations on that copy with respect to <. Similarly, T_f is translated to a set of physical read operations, one for each logical data object. Such a read operation is preceded by all physical write operations on the copies of the corresponding logical data object.

We now give intuitive meanings of the above four conditions. Condition (1) states that each physical operation be a result of translation from a logical operation submitted by a transaction. Note that in a rp log, Σ(**T**) represents a set of physical operations, not a set of logical operations as in the case of a log in a single-site database. Condition (2) states that the rp log preserves the order among the operations of each transaction. This condition is slightly more general than the condition used for defining a rp log in [BeG81]. Condition (3) states that the order among conflicting operations must be specified. Condition (4) states that each copy read in L should have been given some value, and that, after the completion of the execution, only one value for each logical data object updated in L can be read by any subsequent read operation.

Example 2.4.2. Figure 2.4.2 shows a rp log L for the three transactions given in Example 2.4.1. The arcs in the graph indicate the partial order among the operations in the obvious way. Note that even if A < B for physical operations A and B, if there is a path from A to B, no arc is drawn from A to B in the figure. □

A transaction T_j **reads X from** another transaction T_i in a rp log $L = (\Sigma(\mathbf{T}), <)$ if there exists a copy X_a such that (1) $W_i[X_a]$ and $R_j[X_a]$ are operations in Σ(**T**); (2) $W_i[X_a] < R_j[X_a]$; and (3) there is no $W_k[X_a]$ such that $W_i[X_a] < W_k[X_a] < R_j[X_a]$. In the above case, transaction T_j is also said to have **read** a physical data object (or a copy) X_a **from** T_j.

Serial logs we use in connection with rp logs are over logical operations. Thus in a serial log, it is as if the transactions were executed in a single-site database, which maintains only one copy of each data object. For example, $W_0[X,Y,Z]R_1[X,Z]W_1[X]R_2[X]W_2[Y]R_3[Z]W_3[Y,Z]R_f[X,Y,Z]$ is a serial log for the transactions in Example 2.4.2. This kind of serial log, in which data objects accessed by an operation no longer have location indices, is called **1 copy serial log** (or **1C serial log**, for short).

A rp log L_1 is **equivalent** to another log L_2, which is either a rp log or a 1C serial log, if both L_1 and L_2 have the same read-from relation. A rp log is **serializable** if it is equivalent to a 1C serial log over the same set of transactions. As in the case of a single site database system, in a distributed database system, we use $[T_1T_2 \cdots T_n]$ to denote a 1C serial log in which the operations from the transactions are clustered in the order $T_1, T_2, \cdots, T_n$.

Note that, a rp log in which transactions are executed serially, as shown in the following example, may not be equivalent to a 1C serial log.

Example 2.4.3. Consider a rp log

$$L_1 = W_0[X_a,X_b,Y_c,Y_d]R_1[X_a]W_1[Y_d]R_2[Y_c]W_2[X_b]R_f[X_a,X_b,Y_c,Y_d],$$

over transactions $T_1 = R_1[X]W_1[Y]$ and $T_2 = R_2[Y]W_2[X]$. Note that L_1 is a rp log in which the transactions are executed serially. However, it is not equivalent to the 1C serial log $[T_0T_1T_2T_f]$, because in L_1, transaction T_2 reads Y from T_0 instead of T_1. □

In the above example, L_1 is not equivalent to a serial execution in which each transaction sees all the changes made by the transactions executed before it. This is why 1C serial log is used in defining serializability for rp logs.

In order to generalize Theorem 2.3.1 to rp logs, we must distinguish between logical and physical data objects in defining the transaction IO graph in the distributed context. If in a rp log L =

(Σ(T), <) transaction T_j reads X from T_i, the arc from T_i to T_j in $TIO(L)$ should be labeled by X, i.e., the arc is labeled by a logical data object. In the distributed context, a logical write operation of a transaction is said to be **useless** if no physical write operation associated with it writes a value read by another transaction. With these interpretations, the TIO graph of a rp log is defined in the same way as the TIO graph of a log in a single-site database. The TIO graph, $TIO(L)$, of rp log L in Example 2.4.2 is illustrated in Figure 2.4.3(a). The following theorem characterizes a serializable rp log.

Theorem 2.4.1. *A rp log L is serializable if and only if* $TIO(L)$ *has a DITS which orders* T_0 *first and* T_f *last.*

Proof. Similar to the proof of Theorem 2.3.1 given in [IKM87].

To prove the if part, assume that $TIO(L)$ has a DITS that orders T_0 first and T_f last. (Assume that the nodes in the DITS are arranged from left to right.) We have to show that there exists a 1C serial log L_1 equivalent to L. Let σ be the sequence of transactions corresponding to the DITS, and let L_1 be the 1C serial log generated from σ by removing all the dummy transactions in σ. Note that σ has the property that if T_j reads a data object from T_i in L, then T_i is the last transaction preceding T_j in σ that writes X. If this is not the case, then either T_i follows T_j in σ, or another transaction T_k which writes X is located between T_i and T_j in σ. In the first case, the arc (T_i,T_j):X would be directed from right to left, which contradicts the definition of DITS. In the second case, there would be two overlapping intervals starting from T_i and T_k, respectively, which violates a condition of DITS.

We now show that $TIO(L) = TIO(L_1)$. Let T_j read X from T_i in L. Since T_i is the last transaction that writes X before T_j in σ, T_j also reads X from T_i in L_1. On the other hand, suppose T_b reads X from T_a in L_1. Let T_b reads X from T_m in L. Then T_b reads X from T_m in L_1. This

implies that T_a and T_m are identical. Hence T_b reads X from T_a in L. Therefore, we have $TIO(L) = TIO(L_1)$.

To prove the only-if part, we assume that L is serializable and equivalent to a 1C serial log L_1. Since L is equivalent to L_1, $TIO(L) = TIO(L_1)$. It follows from Theorem 2.3.1 that $TIO(L_1)$ has a DITS which orders T_0 first and T_f last. Hence $TIO(L)$ also has a DITS which orders T_0 first and T_f last. □

Example 2.4.3. The TIO graph, $TIO(L)$, of the rp log L in Example 2.4.2 is illustrated in Figure 2.4.3(a). $TIO(L)$ has a DITS as illustrated in Figure 2.4.3(b). Hence L is serializable and it is equivalent to the 1C serial log $T_0 T_2 T_1 T_3 T_f$. □

CHAPTER 3

COPING WITH PARTITION FAILURES

3.1. Partition Failure

In a distributed database system, different kinds of failures can occur. One of them is a site failure, which can be either fail-stop or a Byzantine failure.

Fail-stop is a "clean" type of failure and is relatively easy to handle. A site or a component just crashes, losing all the information it had in volatile memory before the crash; thereafter no activity takes place at the site until it is repaired. In general, fail-stop can be handled by using checkpoints and a transaction log to recover a **consistent state** of the database [BeG83, Gra78]. A database is in a **consistent** state, if the values of all its data objects are the same as the results of serially executing a set of transactions completely and before starting a new transaction.

As for a Byzantine failure, a site suffering from it may deliberately send incorrect messages to other sites. It is very costly in terms of the number of messages for the non-faulty sites to come to an agreement and to take coordinated actions against this kind of malicious activity by failing sites. The problem of reaching agreement despite Byzantine failures has been well studied and is known as the Byzantine Generals Problem [Dol82, FLP85, PSL80].

Besides site failures, we must face partition failures in designing distributed database systems. This kind of failure results from the breakdown of communication links among sites, causing the sites in a network to be separated into two or more groups. Sites in each group can still communicate

with each other, but sites in different groups can no longer talk to each other. The activity in one group is completely unknown to the sites in the other groups. These groups are called **partitions.**

A distributed database system that has undergone a partition failure is called a **partitioned database system.** Thus the topology of the underlying network changes dynamically. In this chapter, we discuss the problems associated with transaction processing in a partitioned distributed database system. In particular, we are interested in the characterization of rp logs generated by an important class of protocols, called **prevention protocols.** The property of prevention protocols will be discussed in Section 3.2. In Sections 3.4 and 3.5, we will show that the **PIO graph** of an execution generated by a prevention protocol in a partitioned database system must have a DITS. The PIO graph of an execution in a partitioned database system is a modification of the TIO graph of the execution over the partitions. The characteristic of the PIO graph of an execution will be used in Chapter 4 in the discussion of an upper bound on the availability of a partitioned distributed database system.

Example 3.1.1. Suppose a distributed database system consists of two sites S_1 and S_2. Both of these sites have copies of two data objects X and Y. The copies of these objects at site S_1 (S_2) are denoted by X_1 and Y_1 (X_2 and Y_2), respectively. A transaction T_1 submitted at S_1 needs to read X and write Y, and a transaction T_2 submitted at S_2 needs to read Y and write X.

Suppose the system is partitioned into two partitions, each containing one site. Since there is no communication between the two sites, T_1 can only read X_1 and update Y_1 and T_2 can only read Y_2 and update X_2. In Figure 3.1.1(a), a rp log L representing this execution is illustrated. The TIO graph, $TIO(L)$, is shown in Figure 3.1.1(b). It is clear that $TIO(L)$ does not have a DITS and hence L is not serializable. □

Example 3.1.1 indicates clearly a problem which crops up in a partitioned database system; if a data object is updated in one partition by a transaction and is read by a transaction in another partition, then the resulting execution may not be serializable.

In a partitioned database system, a set of executions, one for each partition, is called a **global execution**. A great deal of work have been done in trying to control the activity allowed in a transaction when it is executing in a partitioned database system. The goal is to ensure that the global execution is serializable. In general, there are two different approaches to achieving this goal: **pessimistic** approach and **optimistic** approach.

3.2. Pessimistic Approach to Dealing with Partition Failures

A protocol *PT* that always generates a serializable execution in any partitioned database system is said to be a **partition-tolerant** protocol. The **pessimistic** approach is based on the assumption that, if individual partitions exercise concurrency control autonomously, then there is a high probability of generating a non-serializable global execution. Therefore, a partition-tolerant protocol designed with this assumption has to make sure that the global execution consisting of all the operations granted in individual partitions is serializable, even though it must decide to accept or reject an operation submitted in a partition, based solely on the information available within the partition. Partition-tolerant protocols based on this approach are called **inconsistency prevention protocols** (**prevention protocols**, for short) and they are the major objects of investigation in this thesis. This kind of protocol is also referred to as an **on-line protocol** in [COK86], because, once an operation is granted in one partition, it can be committed and will never be rolled back. Hence, it can be used for on-line processing.

The general strategy used in this approach is to define a mutually exclusive condition for read and write operations on the copies of the same logical data object, so that, if a write operation on a data object is allowed in one partition, then any read or write operation on any other copy of the same logical data object is not permitted in any other partition.

Alsberg and Day used the notion of "primary site" [AlD76] to implement read-write exclusion. In their **primary site** model, a single site is designated as the primary site and every read/write access to any data object must first be granted by the scheduler at that site. In the original proposal, locking was used by the scheduler. However, this scheme is too centralized, causing a bottleneck at the primary site. Also, a failure of the primary site will jeopardize the whole system. In the case of a partition failure, only the transactions submitted in the partition which contains the primary site can be executed.

Stonebraker modified the idea of primary site by "distributing" the primary site. Instead of one primary site, one copy of each data object is designated as the **primary copy** [Sto79] of that data object and these primary copies are distributed at more than one site. Any access to a data object must be preceded by the locking of its primary copy. In this scheme, there are no longer severe bottlenecks. Moreover, in the case of a partition failure, more than one partition might be able to execute transactions. However, this scheme also has shortcomings. Firstly, the system has to be equipped with the ability to detect distributed deadlocks. Secondly, if partitioning has occurred, it is not clear how to deal with the locks in a partition that were requested (before the failure) by transactions in other partitions. Thirdly, if the access demand on a primary copy within the partition in which it resides is relatively low in comparison with that from other partitions, then availability degrades.

Gifford [Gif79] presents a simple and elegant "voting scheme" to enforce read-write exclusion. The basic idea is to use a **read quorum** q_r and a **write quorum** q_w. To read a data object, a

transaction must be able to access q_r copies of that object. In other words, a transaction at a site S_i can execute a read operation on a logical data object X by reading a local copy only if it can be sure that there are at least q_r copies of X located in the partition to which S_i belongs. We assume that a site can determine the set of sites in the partition it belongs to, upon detecting a partition failure. With the help of a global directory, a site can thus determine whether there are q_r copies in its partition. As for writing, a transaction must be able to access q_w copies of a data object before it can update it. Updating is performed on every copy that is accessible. In order to achieve mutual exclusion, $q_r + q_w$ must be larger than n, the total number of copies of the data object, and q_w must be larger than $n/2$. The first condition ensures that read and write operations on the same data object are not performed in two different partitions. The second condition guarantees that a write operation on a data object done in one partition will exclude any write operation on the same logical data object in other partitions.

Example 3.2.1. In Figure 3.2.1, a database system consisting of five sites, $S_1, \cdots, S_5$, is divided into two partitions P_1 and P_2. Data objects X and Y are partially replicated as shown in the figure. Let $q_r(X)$ and $q_w(X)$ denote, respectively, the read and write quorums of data object X.

Suppose $q_r(X) = q_w(X) = 3$ and $q_r(Y) = q_w(Y) = 2$. Then X can be read and written in P_1 but not in P_2. Similarly, Y can be read and written in P_2 but not in P_1.

If $q_r(X) = 2$ and $q_w(X) = 4$, then X is now readable in both P_1 and P_2, but X is no longer writable in P_1 or P_2. □

An interesting feature of this approach is that the quorums can be altered to change the accessibility of data objects. Suppose that the quorums of X has been changed in Example 3.2.1 from the first set of quorums to the second set. Before the change, X was accessible only in partition P_1. After the change, it becomes read-accessible in both P_1 and P_2. The problem with this scheme,

however, is that there may not exist any partition that has a quorum required for an operation. As a matter of fact, some work has been done [BGS86, Jaj87], which has tried to ensure that, for any data object X, at least one partition has a quorum in most cases.

Minoura and Wiederhold [MiW82] have proposed an "extended true-copy token" scheme, in which primary copies are marked by tokens which can migrate. This scheme allows more than one partition to execute read operation on the same data object.

Eager and Sevcik [EaS83] have proposed the "missing write algorithm", which is a variant of Gifford's voting scheme. In this scheme, a transaction considers a read operation as the reading of any copy and a write operation as the writing to all the copies. However, this is only possible when there is no partition failure. Once a partition failure is detected, the system goes into the "partition mode", in which Gifford's scheme of mutually exclusive quorums is used. The advantage of this scheme is a reduction in the overhead of reading when there is no failure.

Abbadi, Skeen and Christine [ASC85], and later Abbadi and Toueg [AbT86], modified Gifford's scheme to the "virtual partition scheme". They attempt to track changes in the network topology as closely as possible without being constrained by the need to cope with the changes instantaneously. A **virtual partition** is a set of nodes that have agreed that they can communicate with each other and further that they will not communicate with any other site outside the virtual partition. A transaction can interpret a read operation as the reading of any **accessible** copy, as long as the data object was announced accessible when the virtual partition was formed. A data object is **accessible** in a virtual partition, if the partition has a majority of its copies. This scheme permits cheaper read operations. In return, it must be made sure that all the copies have the most up-to-date value when a virtual partition is formed. This bookkeeping incurs a lot of overhead, whenever there is any change in the network's topology.

In all of the above schemes, no restriction is imposed on the transactions submitted for execution. Wright and Skeen [SkW84] adopt the notion of **transaction class** and propose an interesting scheme to handle partition failures. A class of transactions is defined by its readset and writeset, so that any transaction in the same class reads and writes the same set of data objects. In this model, only a predefined set of classes of transactions can be submitted to a site. The complete information on the classes of transactions that can be submitted to each site is known to every site. Therefore, a partition can use this information, to find out, with the help of a "class conflict graph", a possible conflict between its transactions and those in the other partitions that may lead to a nonserializable execution. This conflict is then resolved by removing some transactions which contribute to the conflict. In this scheme, it is not clear how to avoid unnecessary removal of transactions in different partitions. Unnecessary removal will of course degrade the availability.

As can be seen from the above schemes, the key idea used in designing prevention protocols is to limit the access to data objects in different partitions independently to make sure that the rp log of the global execution is serializable. Note that these schemes can decide on the fly whether to grant or to reject a request for an operation. Once they have granted an operation, they will not rollback the operation.

3.3. Optimistic Approach to Dealing with Partition Failures

The **optimistic** approach is based on the assumption that, even if individual partitions exercise concurrency control autonomously, there is only a small probability that the global execution generated is non-serializable. A transaction submitted in a partition is allowed to execute so long as it can be serialized together with other transactions in the same partition. Any non-serializable global execution is detected when partitions are merged together after the partition failure has been

repaired. Once nonserializability is detected, some operations in the global execution must be rolled back to rebuild a serializable global execution. This involves undoing and redoing some transaction(s). Partition-tolerant protocols using this approach are called **nonserializability detection protocols with rollback** (**detection protocols**, for short).

Example 3.3.1. Assume that two sites S_1 and S_2 belong to two different partitions P_1 and P_2, respectively. Suppose two transactions $T_1 = R_1[X_1]W_1[Z_1]$ and $T_2 = R_2[Z_1]W_2[Y_1]$ are executed in P_1 in such a way that the execution is equivalent to the 1C serial log $[T_1T_2]$. Suppose also that a transaction $T_3 = R_3[Y_2]W_3[X_2]$ is executed alone in P_2. If the two partitions are merged later, then the rp log L in Figure 3.3.1(a) represents the result of merging. (In this example, T_f reads X from X_2 instead of from X_1. If T_f had read X from X_1, the rp log L would be different from the one in Figure 3.3.1(a). Whether T_f should read from X_1 or X_2 depends on the protocol used to control read operation in the final partition.) By looking at the TIO graph, $TIO(L)$, in Figure 3.3.1(b), it is clear that the execution is serializable. Such an execution will be happily accepted by a scheduler that uses the optimistic approach.

Suppose now that T_2 is changed to $T_2 = R_2[X_1]W_2[Y_1]$. Then the result of merging is the rp log L_1 shown in Figure 3.3.1(c). It can be seen that $TIO(L_1)$, shown in Figure 3.3.1(d), has no DITS and hence the global execution is nonserializable.

If the scheduler discovers that the global execution is not serializable, it can eliminate the effect of some of the transactions in order to modify the execution into a serializable one. For example, T_2 can be eliminated from L_1 by rolling back its update on Y_1. Then the rp log now looks like L_3 in Figure 3.3.1(e), and $TIO(L_3)$ has a DITS as shown in Figure 3.3.1(f). Hence the resulting execution is serializable. □

However, in general, testing whether a rp log is serializable and, if not, selecting the minimum number of transactions to be rolled back are both NP-complete [Dav84]. Davison [Dav84] has proposed an "optimistic protocol" in which a "precedence graph" is used to represent a global execution over two partitions. In her model, the transactions are restricted to those whose writesets are contained in their readsets. She proves that the precedence graph is acyclic if and only if the execution is serializable. Hence, the serializability of an execution resulting from merging two partitions can be tested efficiently. However, selecting the minimum number of transactions for rollback is still very costly. A number of heuristics are suggested to solve this selection problem in [Dav84].

When a transaction is rolled back, those transactions which have read data written by it must also be rolled back. This cascading effect could incur a lot of overhead. Finally, all the transactions that are rolled back in this process have to be redone. In addition, if any rolled back transaction had an external action (e.g., output money to a user), then a compensating action may have to be carried out. Because of such limitations, transactions cannot be committed as they are completed. Instead, they must be committed only after nonserializability detection and resolution have completed. Therefore, protocols based on the optimistic approach are also referred to as **off-line** protocols [COK86], indicating that they are only suitable for off-line processing.

3.4. Transaction-Cluster Log

In this and the next sections, we study the characteristics of global executions which are realizable in a partitioned distributed database system. In particular, we characterize the rp logs of executions generated by any prevention protocol. This characterization will form the basis of our study of an upper bound on the availability of a partitioned database system to be presented in

chapter 4.

As time goes on, partitions may be reconnected to form a larger partition, or further split into smaller partitions. If partition A merges with some other partition to form partition B, or if A is split into a set of partitions including B, we say that A **happens before** B. With the relation "happens-before", the set of all the partitions corresponding to a "partition history" is partially ordered. If the same partition occurs more than once in a "partition history", each instance is considered as a different partition. We introduce two special partitions P_0 and P_f, where P_0 is the **initial partition** consisting of all the sites such that T_0 is executed in it, and P_f is the **final partition**, again, consisting of all the sites such that T_f is executed in it. We assume that P_0 always happens before every other partition and every partition (except for P_f) happens before P_f.

Given a rp log $L = (\Sigma(\mathbf{T}), <)$ over a set of transactions **T**, its **sublog** in a partition P_i is a rp log $L_i = (\Sigma_i, <)$, where Σ_i consists of those operations in $\Sigma(\mathbf{T})$ which belong to the transactions executed in P_i, and the order among the operations in Σ_i is inherited from L. Here, we assume that a transaction can execute in only one partition. A transaction T is said to **belong** to a partition P_i, if T is executed in P_i. We use $Trans(P_i)$ to represent the set of all the transactions belonging to a partition P_i.

We now associate a class of 1C serial logs with a poset of partitions. Each 1C serial log in this class has the property that all the operations of the transactions belonging to a partition appear consecutively. More formally, a 1C serial log $L = [T_1 \cdots T_n]$ is a **transaction-cluster 1C serial log** (**TC-serial log**, for short) with respect to a poset of partitions $\mathbf{P} = \{P_0, \ldots, P_m, P_f\}$, if (1) for every partition P_m in **P**, there exists an interval $[i_m, j_m] \subseteq [1, n]$ such that a transaction $T_k \in Trans(P_m)$ if and only if $i_m \le k \le j_m$, (2) all the intervals $[i_m, j_m]$, $m = 0, \ldots, f$, are disjoint and (3) for any two transactions $T_a \in Trans(P_i)$ and $T_b \in Trans(P_j)$, T_a precedes T_b in L if P_i happens before P_j in **P**. Intuitively, the transactions in a TC-serial log are grouped into several disjoint clusters such that each

cluster contains all the transactions executed in one partition and the order of the clusters is compatible with the partial order on **P**. (Thus the sites running the transactions in a cluster can communicate with each other without disruption.) A rp log L over a poset of partitions **P** is **transaction-cluster serializable** (**TC-serializable**, for short), if it is equivalent to a TC-serial log. The set of all TC-serializable rp logs, denoted by **TC**, is a proper subset of the set of all serializable rp logs, **SR**. (See Example 3.4.1 below.)

Example 3.4.1. In Figure 3.4.1(a), a rp log L over two partitions P_1 and P_2 is illustrated. Each arrow in the figure represents the happens-before relationship between its two end partitions. The sublog L_1 of L in P_1 is

$$R_1[X_1]W_1[Z_1]R_2[Z_1]W_2[X_1],$$

and the sublog L_2 of L in P_2 is

$$R_3[Y_2]W_3[Y_2].$$

The TIO graph of L is illustrated in Figure 3.4.1(b), and it has a DITS, T_0, T'_0, T_1, T_2, T_3, T_f. Therefore, L is equivalent to the 1C serial log $L_1 = [T_0 T_1 T_2 T_3 T_f]$. Since both transactions T_1 and T_2 belonging to P_1 are ordered before transaction T_3 which belongs to P_2 in L_1, L_1 is a TC-serial log. Hence, the rp log L is TC-serializable. □

Example 3.4.2. In Figure 3.4.2(a), a rp log L' over two partitions P_1 and P_2 is illustrated. The sublog L'_1 of L' in P_1 is

$$R_1[X_1]W_2[Y_1],$$

and the sublog L'_2 of L' in P_2 is

$$R_3[Y_2]W_3[Y_2].$$

The TIO graph of L is illustrated in Figure 3.4.2(b), and it has a DITS. It is clear from the graph that

T_0, T_1, T_3, T_2, T_f is the only DITS for $TIO(L')$ and $[T_0 T_1 T_3 T_2 T_f]$ is not a TC-serial log. Therefore, L' is serializable, but not TC-serializable. □

3.5. Characterization of Executions Admissible under Partition Failures

In this section, we try to characterize the rp logs of executions generated by any prevention protocol. Here again, the notion of DITS plays a useful role.

Given a rp log L over a poset of partitions $\mathbf{P} = \{P_0, \ldots, P_m, P_f\}$, if the sublog of L in each partition $P_i \in \mathbf{P}$ is serializable, then under what condition is L serializable? This question can be answered by regarding all the operations from the transactions in a partition as coming from one "super transaction". First of all, we have to clarify the meaning of a "serializable sublog". In Section 2.4, we defined a serializable rp log by adding to it two fictitious transactions T_0 and T_f. In order to define the serializability of sublog L_i in a partition P_i, we introduce two fictitious transactions T_{i0} and T_{if}. Transactions in a partition P_i may read the values of different copies of a data object written by different transactions belonging to a partition or partitions that happen before P_i. However, in a rp log generated by any prevention protocols defined in Section 3.2, the following two conditions always hold. The first condition is about the values "imported" by a partition from partitions which happen before it. Initially, the copies of a data object X in a partition P_i may have different values. That is, these copies may have been written by different transactions belonging to a partition or partitions that happen before P_i. However, only one of these values is read by transactions in P_i. The second condition is about the values "exported" by a partition to partitions which happen after it. (A partition P_j **happens after** a partition P_i if P_i happens before P_j.) If Y is a data object written by some transactions in a partition P_i, the copies of Y in P_i may have different final values. However, only one of these values can be read by transactions in partitions which happen after P_i.

For partition $P_i \in \mathbf{P}$, let $PB_i = \{P_j \in \mathbf{P} : P_j$ happens before $P_i\}$ and $RS_i = \{X \in D$: there exists a transaction in $Trans(P_i)$ that reads X from a transaction in $Trans(P_j)$, $P_j \in PB_i\}$. Let $PA_i = \{P_j \in \mathbf{P} : P_j$ happens after $P_i\}$ and $WS_i = \{Y \in D : Y$ is updated by some transaction(s) in $Trans(P_i)\}$. We now formally state the **Unique IO Assumption** on a rp log L over a poset of partitions $\mathbf{P}$.

Unique IO Assumption

(1) For each $X \in RS_i$, every transaction in $Trans(P_i)$ that reads X from some transaction belonging to a partition in PB_i, reads X from the same transaction $T_b(X)$.

(2) For each $Y \in WS_i$, there exists a transaction $T_a(Y) \in Trans(P_i)$ such that, for any $P_j \in PA_i$, if a transaction $T_k \in Trans(P_j)$ reads Y from a transaction in $Trans(P_i)$, T_k reads Y from $T_a(Y)$. □

In condition (2) above, even if no transaction belonging to a partition in PA_i reads Y, $T_a(Y)$ must still exist.

Note that a rp log generated by any one of the prevention protocols described in Section 3.2 satisfies the unique IO assumption. For example, in the primary copy protocol, if the primary copy of a data object X is located at a site in a partition P_i, then the last transaction that wrote the primary copy of X before P_i is formed is $T_b(X)$. Any transaction in $Trans(P_i)$, that reads X from a transaction belonging to a partition happening before P_i, reads the primary copy of X thus from $T_b(X)$. Also, the last transaction in $Trans(P_i)$ that writes the primary copy of X is $T_a(X)$. Therefore, the primary copy protocol satisfies the unique IO assumption.

To see that the quorum protocol also satisfies the unique IO assumption, suppose that the read and write quorums of a data object X are q_r and q_w, respectively. As stated earlier, the sum of q_r and q_w is larger than n and $q_w > n/2$, where n is the total number of copies of X. Every copy of a data

object has a **version number** [ASC85, AbT86]. When a write operation updates X in a partition P_i, it must be able to write at least q_w copies of X. Also, it must update the version numbers of all the copies that its has written to a number larger than all the version numbers that these copies had. When an operation reads X in P_i, it must be able to access q_r copies of X and it reads the copy that has the highest version number. Suppose X can be read in P_i. The copies of X in P_i may initially have different values and version numbers. Suppose a copy X_i initially has the highest version number. Then the transaction that wrote the initial value on X_i is $T_b(Y)$. On the other hand, suppose X can be written in P_i. Before the copies of X migrate to the partitions that happens after P_i, they may have different version numbers. If the version number of a copy X_j is the largest among all these version numbers, then the transaction in $Trans(P_i)$, which wrote this version number, is $T_a(X)$. Since $T_a(X)$ must have written q_w copies of X in P_i, any transaction belonging to a partition happening after P_i, that reads X from a transaction in $Trans(P_i)$, reads X from $T_a(X)$. Hence, quorum protocols satisfy the unique IO assumption. In the following, we assume that the unique IO assumption holds for all rp logs generated by a prevention protocol.

For a rp log over a poset **P** of partitions, let $P_i \in$ **P**, and RS_i and WS_i be as defined earlier in this section. For a data object $X \in RS_i$, let $Copies_i(X)$ be the set of copies of X in P_i that are read by some transaction(s) in $Trans(P_i)$. By condition (1) of the unique IO assumption, all the copies in $Copies_i(X)$ can be considered as initially written by the same transaction $T_b(X)$ defined in the assumption. We thus introduce a fictitious write-only transaction T_{i0} in P_i, which writes the value of X written by $T_b(X)$ into all the copies in $Copies_i(X)$, for every data object $X \in RS_i$. We use $\hat{RS}_i$ to represent the set $\{X_k \in Copies_i(X) : X \in RS_i\}$, i.e., $\hat{RS}_i$ is the set of physical copies written by T_{i0}.

We introduce another fictitious read-only transaction T_{if} in P_i, which reads Y from $T_a(Y)$ defined in condition (2) of the unique IO assumption for every data object $Y \in WS_i$. We use $\hat{WS}_i$ to represent the set of copies, $\{Y_k : Y_k$ is written by $T_a(Y), Y \in WS_i\}$, i.e., $\hat{WS}_i$ is the set of copies at the

sites in P_i whose values are written by $T_a(Y)$. Therefore if $P_j \in PA_i$ and a copy $Y_k \in \hat{WS}_i$ is inherited by P_j from P_i, then the initial value of Y_k is considered to have been written by $T_a(Y)$.

Now we can define the serializability of the sublogs of a log L generated by a prevention protocol over a poset **P** of partitions. For a sublog $L_i = (\Sigma_i, <)$ of L in a partition $P_i \in$ **P**, its **extension** is a poset $(\Sigma_i \cup \overline{\Sigma}_i \cup \hat{\Sigma}_i, <')$, where $\overline{\Sigma}_i$ and $\hat{\Sigma}_i$ are the sets of operations in T_{i0} and T_{if}, respectively, and $<'$ is defined as follows. (1) For any two operations $p \in \Sigma_i$ and $q \in \Sigma_i$, $p <' q$ if and only if $p < q$; (2) $p <' q$ for any two operations $p \in \overline{\Sigma}_i$ and $q \in \Sigma_i$ on the same copy; and (3) $p <' q$ for any two operations $p \in \Sigma_i$ and $q \in \hat{\Sigma}_i$ on the same copy. A sublog L_i of L is **serializable** if its extension L'_i is serializable.

Given a serializable sublog L_i in a partition P_i, for every object X updated in L_i, the transaction $T_a(X)$ (recall (2) in the unique IO assumption) in L_i is called the **output transaction** for X in L_i, and the write operation on X of $T_a(X)$ is the **external write** on X in L_i.

Given a rp log L generated by a prevention protocol over a poset of partitions $\mathbf{P} = \{P_0, \ldots, P_m, P_f\}$, if all its sublogs are serializable, then we can construct a **partition IO graph** for L. As before, let L_i denote the sublog of L in P_i, for $i = 0, ..., m, f$. The **partition IO graph** for L, denoted by $PIO(L)$, is a arc-labeled directed graph with a node set $\mathbf{PN} \cup \mathbf{PN}'$ and an arc set **A**, where $\mathbf{PN} = \mathbf{P}$. **A** has an arc from P_i to P_j labeled by X, denoted by $(P_i, P_j){:}X$, if L_j has a read operation which reads a physical copy updated by the external write of X in L_i. For any partition P_i and any data item X, if the external write on X in L_i is not read by any transaction in other partitions, then $\mathbf{PN}'$ contains a dummy node P'_i and **A** has an arc $(P_i, P'_i){:}X$. There is no other nodes or arcs in $PIO(L)$. Since, for every data object X, transactions in one partition can read the update of X inherited from only one of the partitions that happens before it (recall condition (2) in the unique IO assumption), no two incident arcs to a node in $PIO(L)$ are labeled by the same data object.

Example 3.5.1. In Figure 3.5.1(a), there is a poset of partitions $\{P_0, P_1, P_2, P_3, P_4, P_f\}$. The execution in each partition is illustrated in an oval. The PIO graph of the rp log L representing the global execution is shown in Figure 3.5.1(b). □

A PIO graph was defined above with respect to a rp log generated by a prevention protocol over a poset of partitions. However, it can also be defined for a TC-serial log L over a poset of partitions $\mathbf{P}$ in a similar way. For each partition $P_i \in \mathbf{P}$, there is a segment of transactions in L in which all the transactions are executed in P_i and the last transaction in the segment that writes a data object X is the output transaction for X. $PIO(L)$ can be defined in the same way.

Lemma 3.5.1. *The PIO graph of any TC-serial log L over a poset of partitions* $\mathbf{P} = \{P_0, \ldots, P_m, P_f\}$ *has a DITS which orders* P_0 *first and* P_f *last.*

Proof. Since L is a TC-serial log, it consists of a set of clusters of transactions, one for each partition. Let the nodes in **PN** be ordered from left to right by the order of their corresponding clusters in L. As for a node $P'_i \in$ **PN**′, insert P'_i immediately after P_i in the above ordering. We claim that the resulting order is a DITS for $PIO(L)$ in which P_0 and P_f are ordered first and last, respectively. First of all, all the edges in $PIO(L)$ are directed from left to right. Also, since L is serial, there is no overlapping intervals with the same label, if the nodes of $PIO(L)$ are arranged in the above order. Since the order of the partitions in $\mathbf{P}$ is preserved in the order of the corresponding clusters in L, P_0 and P_f are ordered first and last, respectively, in the DITS constructed above. □

Theorem 3.5.1. *A rp log L generated by a prevention protocol over a poset* $\{P_0, \ldots, P_m, P_f\}$ *of partitions is TC-serializable if and only if the sublogs of L in all the partitions are serializable and* $PIO(L)$ *has a DITS which orders* P_0 *first and* P_f *last.*

Proof. We first prove the if part. Let $\sigma = P_0, P'_1, \cdots, P'_m, P_f$ be the sequence of nodes, except for dummy nodes, corresponding to a DITS of $PIO(L)$. For each P'_i, let $L_s(P'_i)$ be the 1C

serial log equivalent to L'_i, the sublog of L in P'_i. Let L_s be the 1C serial log $L_s(P_0)L_s(P'_1) \cdots L_s(P'_n)L_s(P_f)$, with all the fictitious transactions removed, except for T_0 and T_f. We will show that L is equivalent to L_s and hence L is TC-serializable. Suppose T_j reads X from T_i in L. If both T_i and T_j belong to the same partition P'_k, T_j reads X from T_i in both L'_k and $L_s(P'_k)$, and hence in L_s. If T_i and T_j belong to two different partitions P'_k and P'_l, respectively, then there is an arc (P'_k, P'_l):X in $PIO(L)$. Since σ is a DITS for $PIO(L)$, no node between P'_k and P'_l in σ has an outgoing edge labeled with X. This implies that no transaction executed in the partitions P'_{k+1} to P'_{l-1} writes X. Furthermore, since T_i and T_j belong to two different partitions, T_i must be the output transaction in $L_s(P'_k)$ that writes X. Therefore, there is no write operation on X between T_i and T_j in L_s. Hence, T_j reads X from T_i in L_s.

On the other hand, assume that T_j reads X from T_i in L_s. We want to show that T_j also reads X from T_i in L. Suppose T_j reads X from T_k in L. This implies that T_j reads X from T_k in L_s. Hence T_k and T_i must be identical. Therefore, T_j reads X from T_i in L. It is now clear that L and L_s are equivalent.

Let us next prove the only if part. Suppose that L is TC-serializable and let L_s be a TC-serial log equivalent to L. For every partition P_i, there is a segment L'_i of transactions in L_s which is equivalent to L_i. Since L_s is a 1C serial log, for any data object X, all the transactions (if any) in L'_i that read X from some transaction(s) ordered before L'_i read X from the same transaction T. Since L_s and L are equivalent, for any data object X, all the transactions (if any) in L_i that read X from some transaction(s) not in L_i read X from T. Similarly, for any data object Y, all the transactions (if any) in L but not in L_i, that read Y from a transaction in L_i, read Y from the same transaction. It is clear that the extension of L_i is equivalent to the serial log $[T_{i0}L'_iT_{if}]$, where $[T_{i0}L'_iT_{if}]$ is the 1C serial log beginning with the operations of T_{i0}, followed by all the transactions in L'_i, ordered serially as in L'_i, and ending with the operations of T_{if}. Hence the sublog L_i is serializable. It follows that

all the sublogs of L are serializable. Because of the equivalence of L and L_s, $PIO(L)$ is identical to $PIO(L_s)$. Since L_s is TC-serial, by Lemma 3.5.1, $PIO(L_s)$ has a DITS which orders P_0 first and P_f last. Hence, $PIO(L)$ also has a DITS which orders P_0 first and P_f last. □

Example 3.5.2. The PIO graph, $PIO(L)$, of the rp log L over the partitions $\{P_1, P_2, P_3, P_4\}$ in Figure 3.5.1(b) has a DITS illustrated in Figure 3.5.1(c). Hence, L is TC-serializable and the serialization order is given by

$$T_0 T_1 T_2 T_3 T_5 T_4 T_f. \square$$

In the following, we will characterize the executions generated by prevention protocols. We assume that these protocols satisfy the **non-selective assumption** defined below. Recall the definitions of $\hat{RS}_i$ and $\hat{WS}_i$ given earlier in this section; $\hat{RS}_i$ contains all copies read by transactions in L_i, whose values were inherited from some other partitions which happen before P_i, and $\hat{WS}_i$ contains those copies whose updated values may be read if they migrate to another partition.

Non-selective Assumption: *Let $L = (\Sigma(T), <)$ be the rp log of an execution generated by a prevention protocol PT over a poset of partitions* $\mathbf{P}$. *For every partition $P_i \in \mathbf{P}$, let $L_i = (\Sigma_i, <)$ be the sublog of L in P_i. Then PT must grant all the operations of transaction $T = R[\hat{RS}_i]W[\hat{WS}_i]$ when it is submitted alone in P_i.* □

The non-selective assumption holds for almost all the prevention protocols, except for those which restrict admissible transactions to a fixed set of transaction classes. For example, the "class-conflict protocol" proposed by Skeen and Wright does not satisfy this assumption [SkW84]. The following two results give a general characterization for serializable executions generated by any prevention protocol under partition failures.

Theorem 3.5.2 *If the non-selective assumption holds for a prevention protocol PT, then all the rp logs generated by PT are TC-serializable.*

Before giving a proof for Theorem 3.5.2, we use an example to illustrate the idea of the proof.

Example 3.5.3. Suppose that PT is a partition-tolerant protocol and that the rp log L in Figure 3.5.2(a) is generated by PT in two partitions P_1 and P_2. L is serializable and equivalent to the serial log $[T_0 T_1 T_3 T_2 T_f]$. It is not difficult to see that L is not TC-serializable. With respect to L, we have $\hat{RS}_1 = \{X_1, Z_1\}$, $\hat{WS}_1 = \{Y_1, Z_1\}$, $\hat{RS}_2 = \{Y_2\}$, and $\hat{WS}_2 = \{X_2\}$. If PT is a prevention protocol satisfying the non-selective assumption, then the rp log L_1 illustrated in Figure 3.5.2(b) must also be accepted by PT. However, L_1 is not serializable and this violates the property of PT that it generates only serializable rp logs. Hence, it is not possible for PT to generate L, which is serializable but not TC-serializable. This is because PT cannot know what takes place in P_1, based on the information available in P_2. However, a detection protocol is able to do so and can tell that L is, but L_1 is not, serializable, when the partitions are merged. □

Proof of Theorem 3.5.2. Suppose that L is a serializable rp log generated by a prevention protocol PT over a poset of partition $\{P_0, \ldots, P_m, P_f\}$, but that L is not TC-serializable. It follows from Theorem 3.5.1 that $PIO(L)$ does not have a DITS. For each partition P_i, $(i \neq 0, f)$, construct a transaction $T_i = R_i[\hat{RS}_i] W_i[\hat{WS}_i]$, where $\hat{RS}_i$ and $\hat{WS}_i$ are two sets of copies defined earlier in this section. According to the non-selective assumption, PT would also accept the log L' whose sublog in each partition P_i consisted only of transaction T_i. From the way that T_i's are constructed, it can be seen that $PIO(L')$ is the same as $PIO(L)$. This implies that $PIO(L')$ does not have a DITS. In L', there is only one transaction executed in every partition. Therefore, $PIO(L')$ is identical to $TIO(L')$, if P_i is replaced by T_i for each node P_i in $PIO(L')$. Hence $TIO(L')$ does not have a DITS, and L' is not serializable. This contradicts the property of PT that it generates only serializable rp logs. Therefore, every rp log generated by PT is TC-serializable. □

The following theorem follows immediately from Theorem 3.5.2 by replacing "TC-serializable" with an equivalent condition in Theorem 3.5.1.

Theorem 3.5.3. *If the non-selective assumption holds for a prevention protocol PT, then every rp log L generated by PT over a poset of partitions* $\mathbf{P} = \{P_0, \ldots, P_m, P_f\}$ *have the following two properties :*

(1) *Sublog* L_i *of* L *in each partition* P_i *is serializable;*

(2) *PIO* (L) *has a DITS which orders* P_0 *first and* P_f *last.* □

In the following chapter, we will present an application of Theorem 3.5.3, in which an upper bound on availability will be derived.

CHAPTER 4

LIMITATION ON AVAILABILITY

4.1. Availability in Partitioned Database

Informally, availability is a performance measure of a partition-tolerant protocol. A partition failure degrades the availability of a distributed database. As mentioned in Sections 3.2 and 3.3, many protocols for dealing with partition failures have been proposed. However, little work has been done in investigating the limit on availability. Coan, Oki and Kolodner [COK86] give an upper bound on availability for the 2-partition case, wherein they assume that the transactions submitted are uniformly distributed over all sites. In this section, we will formally define the notion of availability. In Section 4.2, the work by Coan, Oki and Kolodner will be described in more detail. In Section 4.3, we will derive an upper bound on availability using a more general model. In this derivation, the characteristics of the PIO graph of an execution generated by a prevention protocol discovered in Section 3.5 are used.

In order to define availability more formally, let us start with an example.

Example 4.1.1. In a distributed database system, suppose four sites S_1, S_2, S_3 and S_4 are partitioned into two partitions P_1 and P_2 as shown in Figure 4.1.1(a). There are two data objects X and Y, and their copies are distributed as shown in the figure. Transaction $T_1 = R_1[X]W_1[Y]$ is submitted at S_1, transactions $T_2 = R_2[X]W_2[X]$ and $T_3 = R_3[Y]W_3[X]$ are submitted at S_2, and transaction $T_4 = R_4[Y]W_4[Y]$ is submitted at S_3. Let us consider two different executions of this set

of transactions under a quorum protocol. We use $q_r(X)$ and $q_w(X)$ to denote, respectively, the read and write quorums for data object X. The rp log L_1 in Figure 4.1.1(b) represents the first execution, in which $q_r(X)=2$, $q_w(X)=2$, $q_r(Y)=1$ and $q_w(Y)=3$. In this execution, both T_1 and T_4 are rejected because the necessary quorums cannot be obtained in the partitions they belong to. Therefore, only half of the transactions submitted are executed. We say that the **acceptance ratio** of the given set of transactions in L_1, with respect to this set of quorums, $Ac(L_1)$, is 1/2.

If the quorums used are changed to $q_r(X)=2$, $q_w(X)=2$, $q_r(Y)=2$ and $q_w(Y)=2$, then the rp log L_2 in Figure 4.1.1(c) represents another execution of the same set of transactions. In this execution, T_1 and T_3 are rejected, and again $Ac(L_2)=1/2$. Since there are three copies of Y, $q_w(Y) \geq 3/2$ must be satisfied. Therefore, T_1 can never be granted by any quorum protocol, since P_1 has only one copy of Y and no write operation on Y can be performed in P_1. Only one of T_3 and T_4 can be granted, since they have a read-write conflict. Therefore, it can be concluded that 1/2 is the maximum acceptance ratio of this set of transactions over all possible sets of quorums. □

A set of transactions with the sites of submission specified is called a **transaction distribution**. More formally, a transaction distribution is a set of ordered pairs, $\{(T_j, S_i) : T_j$ is a transaction submitted at site $S_i\}$. In Example 4.1.1, $\{(T_1, S_1), (T_2, S_2), (T_3, S_2), (T_4, S_3)\}$ is the transaction distribution. In the following, an execution of the transactions listed in a transaction distribution δ, in which each transaction is submitted to the site specified by an ordered pair in δ, is called an **execution of the transaction distribution**. In the execution of a transaction distribution δ, some transactions may be rejected because of a partition failure. For log L of an execution of a transaction distribution δ over a poset of partitions $\{P_0, \ldots, P_m, P_f\}$, the **acceptance ratio**, $Ac(L)$, is the ratio of the number of transactions in L to the total number of transactions in δ.

From this point on, we focus our attention on the executions generated by a prevention protocol which satisfies the **strongly non-selective assumption**. This assumption is slightly more restrictive

than the non-selective assumption given in Section 3.5. Recall $\hat{RS}_i$ and $\hat{WS}_i$ defined for a rp log L in Section 3.5.

Strongly Non-selective Assumption : *Let* $L = (\Sigma(\mathbf{T}), <)$ *be the rp log of an execution generated by a prevention protocol PT over a poset of partitions* $\mathbf{P}$. *For each partition* $P_i \in \mathbf{P}$, *let* $L_i = (\Sigma_i, <)$ *be the sublog of* L *in* P_i. *If the readset and writeset of a transaction* T *are subsets of* $\hat{RS}_i$ *and* $\hat{WS}_i$, *respectively, then PT must grant all the operations of transaction* T *when it is submitted alone in* P_i. □

For the rest of this chapter, when we refer to a prevention protocol, we assume that it satisfies the strongly non-selective assumption.

Definition 4.1.1. Let **E** be the set of all executions of a transaction distribution δ generated by a partition-tolerant protocol PT over a poset of partitions $\mathbf{P} = \{P_0, \ldots, P_m, P_f\}$. The **availability**, $Av(\delta, PT, \mathbf{P})$, for the transaction distribution δ over **P** with respect to PT is defined to be $max\{Ac(L) : L \in \mathbf{E}\}$. □.

In Definition 4.1.1, we used the maximum acceptance ratio over all possible executions of δ. The following lemma shows that the maximum acceptance ratio does not depend on the order of submission of the transactions in δ. In fact, it depends on the **conflicts** among the transactions submitted in different partitions, which will be defined formally in Section 4.3.

Lemma 4.1.1. *Given a transaction distribution* δ, *a prevention protocol PT and a poset of partitions* $\mathbf{P} = \{P_0, \ldots, P_m, P_f\}$, *there exists a set* $\mathbf{T}'$ *of transactions in* δ *such that, if* L *is any execution of* $\mathbf{T}'$, *in which all the transactions belonging to* P_i *are executed serially in* P_i, *for each* $i = 0, \ldots, m, f$, *then* $Ac(L) = Av(\delta, PT, \mathbf{P})$.

Proof. Let L_1 be an execution of δ such that $Ac(L_1) = Av(\delta, PT, \mathbf{P})$. Then $\mathbf{T}'$ is the set of transactions in L_1. According to the strongly non-selective assumption, if $T_i \in \mathbf{T}'$ and it is submitted alone in P_i, then all the operations of T_i are accepted by PT. Hence all the transactions in $\mathbf{T}'$ that belongs to P_i will be granted by PT if they are submitted and executed serially in P_i. □

In Example 4.1.1, the maximum availability for the transaction distribution given in the example, with respect to any quorum protocol, was 1/2, i.e., the maximum value of $\{Av(\delta, PT, \mathbf{P}) : PT \text{ is a quorum protocol}\}$ is 1/2. However, if other protocols are used, the availability for the same distribution may be different.

Example 4.1.2. Let X_1 and Y_1 be the primary copies (see Section 3.2) in Example 4.1.1, and suppose the primary copy protocol is applied to the transaction distribution. Then all the transactions submitted in P_1 are accepted, and transaction T_4 submitted in P_2 is rejected. Hence, the acceptance ratio of this execution of the transaction distribution is 3/4. It can be seen that no other execution can accept all four transactions, if the primary copy protocol is used. Hence the availability is equal to 3/4, with respect to the primary copy protocol. □

From Examples 4.1.1 and 4.1.2, we can conclude that availability depends not only on the transaction distribution, but also on the protocol used. It is interesting to see if a general upper bound on availability for any prevention protocol exists. In the case of a detection protocol, every transaction submitted to a partition is accepted. Therefore, we would not discuss the limit on the availability for detection protocols. In the following, we consider only the 2-partition case (not counting P_0 and P_f) and hence we will not explicitly mention the poset of partitions, and the availability of a transaction distribution δ with respect to a protocol PT in the 2-partition case will be denoted by $Av(\delta, PT)$. We are interested in finding a general upper bound on the availability for some classes of transaction distributions having a certain property, with respect to any prevention protocol. Finding a general upper bound in the case where there are more than two partitions appears

rather difficult.

4.2. Uniform Transaction Distribution and Availability

In this section, we review the upper bound on availability in the 2-partitions case, given by Coan, Oki and Kolodner [COK86]. The database model they use is a fully replicated database and each transaction has a readset containing its writeset. Also, a transaction always reads a data object before updating it. A transaction which writes some data objects is called an **update transaction.** The class of transaction distributions considered in their model are assumed to satisfy the **uniformity assumption** stated below.

Uniformity Assumption. *For $i = 1, 2, \cdots, n$, let f_i be the fraction of update transactions submitted at a site S_i over all update transactions, and let D be any set of data objects. Among all the update transactions with writeset D, the fraction of update transactions submitted at S_i is also equal to f_i.*

For a given transaction distribution δ, the partition, P_{maj}, which has the majority of transactions is called the **majority partition,** while the partition, P_{min}, which has the minority of transactions is called the **minority partition.** (If the two partitions have the same number of transactions, then either one can be the majority partition and the other the minority partition). The following are the parameters used in specifying the upper bound in [COK86].

t = total number of transactions listed in δ

u_{maj} = fraction of t that are update transactions and are submitted in P_{maj}

u_{min} = fraction of t that are update transactions and are submitted in P_{min}

r_{maj} = fraction of t that are read-only transactions and are submitted in P_{maj}

r_{min} = fraction of t that are read-only transactions and are submitted in P_{min}

Theorem 4.2.1.[COK86] *For any transaction distribution δ that satisfies the uniformity assumption and for any prevention protocol PT, we have*

$$Av(\delta, PT) \leq u_{maj} + r_{maj} + r_{min}. \square$$

Since Theorem 4.2.1 is proved in [COK86], we provide here only an intuitive reasoning for the theorem. Firstly, for any two update transactions T_1 and T_2, which update the same object, say X, if they are submitted in P_{maj} and P_{min}, respectively, then one of them must be rejected. To see this, suppose that both of them are granted. Then both P_{maj} and P_{min} must have a transaction that reads X from T_0 belonging to the initial partition P_0, because the value of X must be inherited from T_0. Also , $T_f \in P_f$ (the final partition) must read X from a transaction belonging to either P_{maj} or P_{min}. If T_f reads X from P_{maj}, then the PIO graph of the global execution contains the subgraph shown in Figure 4.2.1(a). If T_f reads X from P_{min}, then the PIO graph of the global execution contains the subgraph in Figure 4.2.1(b). In either case, the PIO graph does not have a DITS and the corresponding global execution is not serializable. Hence, it is impossible for both T_1 and T_2 to appear in a serializable execution.

Secondly, by the uniformity assumption, for any update transaction accepted in the P_{min}, there are more (or at least as many) update transactions with the same writeset that are rejected in *Pmax*. Therefore, the availability is maximized by accepting the update transactions submitted in *Pmax* and

rejecting those submitted in P_{min}. All the read-only transactions can be accepted. Thus, it follows that $Av(\delta, PT)$ is bounded by $u_{maj} + r_{maj} + r_{min}$.

4.3. A General Upper Bound on Availability

In this section, we shall derive a general upper bound on the availability for a class of transaction distributions in the 2-partition case. As before, we use δ to denote a transaction distribution and P_i, $i = 1$ or 2, to denote the two partitions under consideration. For $i = 1, 2$, let Γ_i denote the set of transactions in the transaction distribution δ, submitted in P_i. Further, we use $WS(A)$ and $RS(A)$ to denote the writeset and readset of a set of transactions A, respectively, i.e., $WS(A)$ ($RS(A)$) is the union of the writesets (readsets) of all the transactions in A. Two transactions $T_1 \in Trans(P_1)$ and $T_2 \in Trans(P_2)$ are said to **conflict**, if $WS(\{T_1\}) \cap RS(\{T_2\})$ or $RS(\{T_1\}) \cap WS(\{T_2\})$ is nonempty. (The reason we do not regard T_1 and T_2 as conflicting, when $WS(\{T_1\}) \cap WS(\{T_2\})$ is nonempty will be explained at the end of this section.) A transaction T **reads from (writes into)** a set D of data objects, if the readset (writeset) of T has a nonempty intersection with D. Note that this condition does not require the readset (writeset) of T to be a subset of D.

In the following discussion of an upper bound on availability, we adopt the **weak uniformity assumption** on transaction distribution δ, defined as follows.

Weak Uniformity Assumption. *For any set D of data objects, if the number of transactions in δ submitted in P_1 (P_2), which read from (write into) D, is larger than the number of transaction in δ submitted in P_2 (P_1), which write into (read from) D, then any subset of D also has this property.* □

Note that the above assumption is not strictly "weaker" than the uniformity assumption, because of the condition on the read set. However, if the readset and writeset of a transaction are always the same, then the uniformity assumption implies the weak uniformity assumption. Because of the conflict between the transactions in Γ_1 and Γ_2, not all the transactions submitted can be accepted. Let C_{12} denote the set $WS(\Gamma_1) \cap RS(\Gamma_2)$ and C_{21} denote the set $WS(\Gamma_2) \cap RS(\Gamma_1)$. In other words, C_{12} and C_{21} are the "source of conflict" between the transactions in Γ_1 and Γ_2. We define the following parameters for specifying an upper bound on the availability for δ.

ρ_1 = the set of all transactions belonging to P_1 that write into C_{12}, i.e., those which conflict with some transactions belonging to P_2 because they have written some data objects which are read by some transitions in $Trans(P_2)$.

θ_2 = the set of all transactions belonging to P_2 that read from C_{12}, i.e., those which conflict with some transactions belonging to P_1 because they read some data objects which were written by some transitions in $Trans(P_1)$.

ρ_2 = the set of all transactions belonging to P_2 that write into C_{21}, i.e., those which conflict with some transactions belonging P_1 because they have written some data objects which are read by some transitions in $Trans(P_1)$.

θ_1 = the set of all transactions belonging to P_1 that read from C_{21}, i.e., those which conflict with some transactions belonging to P_2 because they read some data objects which were written by some transitions in $Trans(P_2)$.

Theorem 4.3.1. *For any transaction distribution* δ *that satisfies the weak uniformity assumption and for any prevention protocol* PT, *we have*

$$Av(\delta, PT) \le 1 - \frac{m}{|\Gamma_1| + |\Gamma_2|},$$

where $m = min(|\rho_1|, |\rho_2|, |\theta_1|, |\theta_2|)$.

Proof. Suppose L is the rp log of an execution of δ over $\{P_1, P_2\}$ with protocol PT. According to Theorem 3.5.3, $PIO(L)$ must have a DITS, which is either $P_0P_1P_2P_f$ or $P_0P_2P_1P_f$.

Let τ_1 be the set of all the transactions appearing in the sublog of L in P_1 and let τ_2 be the set of all the transactions appearing in the sublog of L in P_2. Note that $\tau_1 \subseteq \Gamma_1$ and $\tau_2 \subseteq \Gamma_2$, since some transactions may be rejected by PT. Suppose that $PIO(L)$ has a DITS given by $P_0P_1P_2P_f$. As illustrated in Figure 4.3.1(a), there can be no data object X such that X_1 is written by a transaction in τ_1 and X_2 is read by any transaction in τ_2, where X_1 and X_2 are two copies of X in P_1 and P_2, respectively. Otherwise, $P_0P_1P_2P_f$ would not be a DITS of $PIO(L)$, since $PIO(L)$ would have an arc (P_0, P_2):X, because a transaction in P_2 can read only from T_0 belonging to P_0 (i.e., X_2 is written by T_0,) but not from any transaction belonging to P_1. Hence, $WS(\tau_1)$ and $RS(\tau_2)$ must be disjoint. C_{12} contains two disjoint subsets W_{12} and R_{12}, where $W_{12} = WS(\tau_1) \cap C_{12}$ and $R_{12} = RS(\tau_2) \cap C_{12}$. (The inclusion relationships among the sets $WS(\Gamma_1)$, $RS(\Gamma_2)$, $WS(\tau_1)$, $RS(\tau_2)$, C_{12}, W_{12} and R_{12} are illustrated in Figure 4.3.2).

Let $RJ_1 = \Gamma_1 - \tau_1$, i.e., the set consisting of all the transactions submitted in P_1 that are rejected by PT. In other words, RJ_1 consists of all the transactions submitted in P_1 that write into $C_{12} - W_{12}$. Also let $RJ_2 = \Gamma_2 - \tau_2$, i.e., the set consisting of all the transactions submitted in P_2 that are rejected by PT. RJ_2 thus consists of all the transactions submitted in P_2 that read from $C_{12} - R_{12}$.

Suppose the number of transactions in P_1 which write into C_{12} is larger than the number of transactions in P_2 which read from C_{12}. It follows from the weak uniformity assumption that

$|RJ_1| \geq |AC_2|$, where AC_2 is the set of transactions in P_2 which read from R_{12}. Since θ_2 is defined as the set of transactions in P_2 which read from C_{12}, it is clear that θ_2 is the union of AC_2 and RJ_2. Hence,

$$|\tau_1| + |\tau_2| = |\Gamma_1| - |RJ_1| + |\Gamma_2| - |RJ_2|$$

$$\leq |\Gamma_1| + |\Gamma_2| - (|AC_2| + |RJ_2|)$$

$$\leq |\Gamma_1| + |\Gamma_2| - |\theta_2|. \qquad (4.1)$$

(Note that AC_2 and RJ_2 may not be disjoint, and this is why the last inequality in (4.1) is not an equality). If the number of transactions in P_1 which write into C_{12} is smaller than the number of transactions in P_2 which read from C_{12}, then $|RJ_2| \geq |AC_1|$, where AC_1 is the set of transactions in P_1 which write into W_{12}. Since ρ_1 is defined as the set of transactions in P_1 which write into C_{12}, it is clear that ρ_1 is the union of AC_1 and RJ_1. Hence,

$$|\tau_1| + |\tau_2| = |\Gamma_1| - |RJ_1| + |\Gamma_2| - |RJ_2|$$

$$\leq |\Gamma_1| + |\Gamma_2| - (|AC_1| + |RJ_1|)$$

$$\leq |\Gamma_1| + |\Gamma_2| - |\rho_1|. \qquad (4.2)$$

On the other hand, if the DITS in $PIO(L)$ is $P_0P_2P_1P_f$, then no data object X written by a transaction submitted in P_2 can be read by any transaction submitted in P_1, as illustrated in Figure 4.3.1(b). Hence, $WS(\tau_2)$ and $RS(\tau_1)$ must be disjoint. In this case, the following two inequalities correspond to (4.1) and (4.2), respectively.

$$|\tau_1| + |\tau_2| \leq |\Gamma_1| + |\Gamma_2| - |\theta_1|. \qquad (4.3)$$

$$|\tau_1| + |\tau_2| \le |\Gamma_1| + |\Gamma_2| - |\rho_2|. \qquad (4.4)$$

Therefore, we have

$$Ac(L) = \frac{|\tau_1| + |\tau_2|}{|\Gamma_1| + |\Gamma_2|}$$

$$\le 1 - \frac{m}{|\Gamma_1| + |\Gamma_2|},$$

where $m = min(|\rho_1|, |\rho_2|, |\theta_1|, |\theta_2|)$. Since this is true for the rp log of any execution of δ over $\{P_1, P_2\}$, it follows that

$$Av(\delta, PT) \le 1 - \frac{m}{|\Gamma_1| + |\Gamma_2|}. \Box$$

Note that when we considered the DITS $P_0P_1P_2P_f$ in the above proof, only data objects caused conflicts between the transactions in τ_1 and τ_2. We did not consider the data objects in $WS(\tau_1) \cap WS(\tau_2)$, because, if there exists a data object $X \in WS(\tau_1) \cap WS(\tau_2)$, $P_0P_1P_2P_f$ will still be a DITS for $PIO(L)$ as long as $X \notin RS(\tau_2)$. This is the reason why we did not consider the intersection of their writesets when we defined conflict between transactions.

CHAPTER 5

TECHNIQUES FOR ACHIEVING HIGH AVAILABILITY

5.1. Trade-off between Serializability and Availability

In Chapter 4, it was shown that under a partition failure there is a limitation on the availability of distributed database systems. Serializability and availability are conflicting goals in designing distributed database systems. It was suggested in [GaK87] that the trade-off between these goals can be viewed as a linear spectrum of possible solutions. At one end, there is global serializability, and at the other end, there is 100% availability.

Systems that guarantee serializability, e.g., [AbT86, BeG81, Dav84, DGS85, Gif79, SkW84], suffer from high communication overheads and low availability. When a transaction updates a data object, it must also update all, or at least a majority of, the copies of the object at remote sites. In the worst case, when a communication failure causes partitioning of the network, availability is seriously degraded. However, such a system has the very desirable feature that global serializability is maintained.

As for systems at the other end of the spectrum, e.g., [BGR83, GAB83, SBK85], they emphasize local availability of data objects. All data objects are fully replicated. Read and write operations are always executed locally. Therefore, the execution of a transaction is guaranteed to be fast. Even if a communication failure occurs, there is no degradation in execution speed or availability. The most serious deficiency of these systems is that there is no guarantee of

serializability. In fact, very little can be said about the correctness of these systems.

In Section 5.2, a system which ignores serializability but provides the maximum availability is described. Then, in Section 5.3, we will review an approach to achieving both serializability and high availability by restricting transaction behaviour. For this purpose, we will discuss a model called **fragmented database system.** In Section 5.4, two schemes proposed by Kogan and Garcia-Molina [KoG87] for a fragmented database system will be discussed. Their schemes adopt fixed "access patterns", which are rather restrictive, to achieve both serializability and high availability. In Chapters 6 and 7, we will propose two schemes which are more general than those in [KoG87].

5.2. A Highly Available Distributed Database System

In this section, we describe the system SHARD (System for Highly Available Replicated Data), developed and implemented at CCA (Computer Corporation of America) [SBK85, Sar86].

The database in this system is fully replicated. The execution of a transaction is completed locally at the site of its submission and updates are broadcast afterwards. No site or partition failure can affect the execution of any transaction at an operational site, and updates will eventually arrive at every operational site, after the partition failure has been repaired. Therefore, availability is guaranteed to be 100%. The main issues are how to merge the updates from different sites and how to define correctness for this kind of execution.

Figure 5.2.1 shows the architecture of SHARD at each site. The DB and Update History are two secondary storages, and a copy of the whole database is stored in the DB. In addition, there are three modules, called Interactor, Distributor and Checker. Interactor is the interface between the system and the local users. By reading data from the DB, it generates responses to user requests and "update actions". Here, we use an example of cash withdrawal in a banking system to illustrate the

idea of an update action. After money is output to a user, instead of changing the balance immediately, an update action, which is a transaction in itself that decrements the amount withdrawn from the balance, is generated. If this update action and the output action are executed together as an atomic action, the resulting database is always consistent. However, this would require a transaction to wait for its update actions at all the sites to complete, before the output action can be executed. This prolongs the response time of the transaction and it will be blocked if a partition failure occurs in the middle of its execution. In order to achieve the high availability objective, SHARD takes another approach. Money is output first and an update action is executed afterwards, independently at different sites.

The update actions generated by Interactor are first stored in Update History. Distributor is responsible for broadcasting them to all the other sites by using a reliable broadcast protocol, e.g., [AwE84,GLB85], which guarantees delivery of messages at every site. (The delivery may suffer from a long delay). When an update action is broadcast, it carries a global timestamp, indicating the time when the update action was generated.

Checker executes the update actions in Update History in their timestamp order. When Checker updates the DB, it may miss some update actions from a remote site because of slow communication or a partition failure. This can be remedied only by undoing and redoing some update actions once it is discovered. Eventually, the copies at different sites are guaranteed to be mutually consistent, i.e., the copies of the same logical data object will have the same value.

However, in general, an execution generated in SHARD is not guaranteed to be serializable by the timestamp order of the transactions involved. This can be explained as follows. Suppose that a transaction T_1 is executed at a site S_1 with a timestamp t_1 and it reads only a data object X. When T_1 interacts with the Interactor, it retrieves for T_1 a value of X from the DB. The output action and update action of T_1 are generated based on this value. Suppose also that a transaction T_2 with a

timestamp $t_2 < t_1$ has been executed at another site S_2 and the update action of T_2, which also writes X, arrives at S_1 after the update action and output action of T_1 have been generated. In this case, T_1 has missed the value of X written by T_2. If these two transactions are serialized by their timestamps, T_1 should read the value of X updated by T_2. Hence, an execution in SHARD may not be serializable by the timestamp order of the transactions involved. In other words, the database state in which an update action is eventually executed is, in general, different from the state in which it was generated; therefore, serializability is not guaranteed in SHARD, even though mutual consistency and high availability are achieved [SBK85, Sar86]. In [SBK85, Sar86], "compensation action" is used to remedy inconsistency, once it is discovered.

When Checker tries to merge update actions from different sites, techniques such as **data-patching** and **log transformation** [BGR83, BlK85] may be used.

Another issue in this scheme is the definition of correctness. The correctness of the executions generated using this scheme depends in many cases on the semantics of the operations involved. It is very difficult to give a general correctness criterion to these executions. Lynch, Merrit, Siegel [LBS86] attempt to solve this problem. However, this is an area that requires further research.

5.3. Fragmented Distributed Database System

In Chapter 4, we showed that, if serializability is required, there is a limit on the availability of a general distributed database system. On the other hand, it is demonstrated in SHARD, described in Section 5.2, that it is possible to achieve a full (100 percent) availability by ignoring serializability. In general, the two conflicting goals of serializability and availability must be compromised. However, there is a third approach which provides both serializability and high availability by reducing the generality of transactions allowed in a system. In the rest of this thesis, we focus our

study on this approach.

In the most general case, a transaction submitted at a site can read and write any data object either at the local site or some remote site. However, in some domains of application, this generality can be restricted to some extent without affecting the defined goal. In such a case, it is possible to achieve both serializability and availability by restricting the behaviour of transactions.

For example, in an airline information management system, a flight scheduling system may be centralized at the airline's headquarters. For the sake of fast accessibility and reliability, the database of flight schedules may be replicated at many different sites. In this case, it is reasonable to restrict updating of a flight schedule to be executed only at the headquarters. A site other than the headquarters can read any flight schedule by accessing its local copy. However, no site except for the headquarters is allowed to modify any flight schedule for itself. The only way in which a site can modify a flight schedule is to send a request to the headquarters. On the other hand, it is necessary for a site other than the headquarters to read flight schedules. For example, the accounting department needs to read flight schedules to find the total flying time of every pilot to calculate the payroll. In this example, we are trying to put constraints on data objects that a transaction can read and write. In the following, this kind of restriction is formalized and a new database model, called **fragmented database**, is introduced. A similar model has been independently developed by Garcia-Molina and Kogan [GaK87]. However, their approach and results have different flavors and applications from ours.

In a **fragmented database system**, data objects are fully replicated at all the participating sites[1]. The logical database is partitioned into disjoint pieces called **fragments;** and each site manages exactly one fragment. Therefore, there is a one-one correspondence between the fragments

[1] The assumption of full replication can be relaxed to partial replication, without affecting the results presented in the following. This assumption is made to make the description and discussions simple.

and the sites. The site which manages a fragment is called its **home site**. The fragment managed by a site is called the **home fragment** of the site. Data objects in a fragment **belong** to the fragment's home site. The fragments other than the home fragment of a site are called **remote fragments** with respect to that site. Note that each site has not only its home fragment, but also the copies of the remote fragments.

A transaction is said to **belong** to the site at which it is issued and this site is called its **home site**; we also refer to the home fragment and the remote fragments of a transaction's home site as its **home fragment** and **remote fragments**, respectively. In this model, transactions can update only data objects in their home fragments. Therefore, updating a data object in a remote fragment must be done by sending a request to the home site of the remote fragment.

Two kinds of transactions are allowed to run in a fragmented database system. A **local transaction** can read and write data objects only in its home fragment. A **global transaction** can read from any fragments, but it can write only into its home fragment.

Both local and global transactions execute their read operations by accessing the copies available at their home sites. (Recall that the database is fully replicated). A transaction is committed (i.e., its updates are permanently reflected in the database) after all its read and write operations have completed at its home site. As a result of this, to commit a transaction, the system never has to wait for replies from other sites confirming the arrival of its updates. No waiting is necessary to perform distributive commitment. Instead, the updates by a transaction are packaged and broadcast asynchronously to all other sites after the transaction has been committed locally. Therefore, even if there is a partition failure, a transaction can still commit at its home site and let a broadcast protocol take care of the delivery of its updates afterwards.

At each site, there is a local scheduler which controls local read/write accesses and remote updates from other sites, so that the result of their executions is serializable. We require that at each site the updates of transactions are broadcast to other sites in a serialization order of the transactions. For example, if the local scheduler is timestamp-based [BeG81], then updates granted by it are broadcast in their timestamp order. Furthermore, updates from the same site are processed by every receiver site in the order they are sent. Hence, a transaction always reads all data objects in its readset from a consistent database state.

Example 5.3.1. In Figure 5.3.1, the sites S_1, S_2 and S_3 are the home sites of the fragments F_1, F_2 and F_3, respectively, and $\overline{F}_1$, $\overline{F}_2$ and $\overline{F}_3$ are the replicas of F_1, F_2 and F_3, respectively. The data objects X, Y, Z and W belong to fragments as shown in the figure. The subscripts of the data object copies indicate their sites

A transaction T_1 submitted at S_1, which reads X and writes Y, is a local transaction belonging to S_1. T_1 is completed locally at S_1 before its update is broadcast to S_2 and S_3 to update Y_2 and Y_3, respectively.

A transaction T_2 submitted at S_1, which reads Z, W and writes X, is a global transaction belonging to S_1. T_2 is also completed locally by reading Z_1 and W_1 before its update is broadcast. Note that only transactions submitted at S_1 can update data objects in F_1. □

In this section, we have introduced the notion of a fragmented database system, in which transaction behaviour is restricted. In the next section, we shall discuss results in [GaK87, KoG87], which demonstrate that both high availability and serializability can be achieved in a fragmented database system.

5.4. Transaction Processing with a Static Read Access Graph

In a fragmented database system, in general, a transaction submitted at one site can read data objects belonging to any other site. If the local schedulers located at different sites work independently of each other, non-serializable executions may be generated. We shall illustrate this fact by the following example.

Example 5.4.1. Consider a fragmented database system in Figure 5.4.1(a), which consists of two sites S_1 and S_2 with fragments F_1 and F_2, respectively. The database contains two data objects X and Y with copies at the two sites. Suppose that initially the copies of X have a value λ_0, while the copies of Y have a value v_0.

Two local transactions $T_1=R_1[X]W_1[X]$ and $T_2=R_2[Y]W_2[Y]$ are submitted at S_1 and S_2, respectively. X_1 is updated by T_1 to a new value λ_1, while Y_2 is updated by T_2 to v_1 at roughly the same time. After T_1 and T_2 are committed locally, their updates are sent to the other site. (See Figure 5.4.1(b)). Suppose that before the arrival of these updates, two read-only global transactions $T_3=R_3[X]R_3[Y]$ and $T_4=R_4[X]R_4[Y]$ are executed at sites S_1 and S_2, respectively. Since the updates do not arrive in time, the values read by T_3 are λ_1 for X and v_0 for Y. As for T_4, the values read are λ_0 for X and v_1 for Y. Since T_3 has read the update from T_1 but not from T_2, T_3 must be serialized after T_1 and before T_2. On the other hand, T_4 has read the update from T_2 but not from T_1, T_4 must be serialized after T_2 and before T_1. Therefore, this execution of the four transactions is not serializable. □

From Example 5.4.1, it is clear that the restriction given above on transaction behaviour in a fragmented database is not sufficient to guarantee global serializability. The approach taken in [GaK87, KoG87] to ensure global serializability is to further require each transaction's read and write operations to conform to a static access pattern.

Definition 5.4.1.[GaK87] Given a fragmented database with sites $S_1, \ldots, S_n$ and corresponding fragments $F_1, \ldots, F_n$, the **read-access graph** (**RAG**, for short) is a directed graph $G = (N, A)$, where $N = \{F_1, \ldots, F_n\}$ and $A = \{(F_i, F_j) : i \neq j$ and a transaction T with home fragment F_i can read any data object in $F_j\}$. □

The RAG is an abstraction of the constraints imposed on the activity of transactions. Transactions submitted at S_i are allowed to read data objects belonging to F_j, only if there is an arc from F_i to F_j in the RAG. Of course, reading is not done by sending out a remote request to S_j, but rather by reading the copy of F_j at S_i, which is updated when new values from S_j arrive at S_i. Thus, if there is no outgoing edge at fragment F_i in a RAG, transactions submitted at S_i can read and write only the data objects in F_i. In other words, no global transaction can be submitted at S_i. Since we assume a one-to-one correspondence between the fragments and the sites, we sometimes refer to the nodes of a RAG as sites rather than fragments, whenever it is more convenient. A directed graph is **loopless**, if it has no undirected cycle in it.

Example 5.4.2. Figure 5.4.2(a) shows a RAG, which is a complete graph, for three fragments F_1, F_2 and F_3 of a database. This is the most general RAG for three fragments, in which a transaction submitted at any site is allowed to read data objects from any site.

Figure 5.4.2(b) shows a restricted cyclic RAG for the same set of fragments. Transactions submitted at one site can read data objects at only one more site.

Figure 5.4.2(c) shows an acyclic RAG, in which transactions submitted at S_2 have no restriction on access. They can read from both F_1 and F_3. As for the transactions submitted at S_3, they can read only from S_1. The most severe restriction is imposed on the transactions submitted at S_1. They can read and write only data objects in its own fragment F_1.

The restrictions imposed by the RAG in Figure 5.4.2(d) are even stronger than those in Figure 5.4.2(c). Figure 5.4.2(d) shows a loopless RAG. It is not only acyclic, but its nondirected version is also acyclic. With this RAG, only the transactions submitted at S_2 can read data objects from other fragments. The transactions submitted at S_1 and S_3 can access only their own fragments. □

The following theorem states that a fragmented database with a very restrictive static access pattern can achieve both serializability and high availability.

Theorem 5.4.1.[GaK87] *If the RAG imposed on a fragmented database is loopless, any execution in the database is serializable.* □

The following example illustrates an application of this theorem.

Example 5.4.3. This example is oversimplified; however, it serves the purpose of demonstrating a plausible application of Theorem 5.4.1.

In this application, we keep track of sales and inventory stock for a wholesale company. There are k warehouses at which the merchandise is sold to retailers. At each location, there is a fragment that contains a record of every sale made, a record of every new merchandise shipment received at that location, and the quantity in stock of each product. Name these fragments $F_1, F_2, \cdots, F_k$. Furthermore, there is a fragment C controlled by the company's central office. In this fragment, information is recorded which represents decisions concerning future purchases (from manufacturers). These decisions are arrived at by computations based on the periodic readings of fragments $\{F_i : i = 1, \ldots, k\}$. This database is characterized by the RAG shown in Figure 5.4.3.

Note that there is a high degree of availability in this database system. For instance, each warehouse can still enter the sales and shipment information even if there is a communication failure. On the other hand, global serializability is never violated, even during a partition failure. Of course, if the system is facing communication failure, the central office has to make decisions relying on

stale information. However, this is unavoidable in the case of a partition failure. □

Theorem 5.4.1 implies that it is possible to have both serializability and high availability; however, this is achieved at the expense of a restricted read access pattern. With a slight relaxation on the requirement defined by the RAG, the theorem no longer holds. The following example shows that an acyclic RAG with a loop may generate a non-serializable execution.

Example 5.4.4. Consider the RAG for three fragments F_1, F_2 and F_3 given in Figure 5.4.4(a), which is an acyclic graph with a loop. The copies of three objects X, Y and Z are located as shown in Figure 5.4.4(b). F_1, F_2, and F_3 are the home fragments of S_1, S_2 and S_3, respectively, and $\bar{F}_1$, $\bar{F}_2$, and $\bar{F}_3$ are the replicas of S_1, S_2 and S_3, respectively. Transaction $T_3 = R_3[Z]W_3[Z]$, submitted at S_3, changes the value of Z from λ_0 to a new value λ_1. The value λ_1 is broadcast to S_1 and S_2. Transaction $T_2 = R_2[Z]W_2[Y]$, submitted at S_2, reads the value λ_1 of Z received from T_3. T_2 changes the value of Y from ν_0 to ν_1, and ν_1 is broadcast to S_1 and S_3. Transaction $T_1 = R_1[Y,Z]W_1[X]$, submitted at S_1, is executed before the arrival of λ_1 and after the arrival of ν_1. This is possible because of variable delays on different communication links. Therefore, T_1 reads the obsolete value λ_0 of Z and the updated value ν_1 of Y.

This execution can be described by the rp log L illustrated in Figure 5.4.4(c). It is clear from this execution that T_3 has to be serialized before T_2, because T_2 reads Z from T_3. Also, T_2 has to be serialized before T_1, because T_1 reads Y from T_2. However, T_1 reads Z from T_0, i.e., not from T_3. This makes it impossible for the TIO graph, $TIO(L)$, of log L to have a DITS. (See Figure 5.4.4(d)). Hence, this execution is non-serializable. □

Even though Theorem 5.4.1 is an interesting result, it can be argued that its condition is too restrictive. It is not applicable in the case where a RAG is acyclic but has a loop. Kogan and Garcia-Molina have successfully improved on this result to allow the RAG to contain loops.

With a close examination of Example 5.4.4, it can be seen that the main source of problem, which affects the serializability of the execution given in the example, is the disparity in the arrival times of the update of T_3 at sites S_1 and S_2. Because of the late arrival of this update at S_1, transaction T_1, submitted at S_1, could read both a stale value of Z and an updated value of Y.

On the other hand, suppose that the update of T_3 at S_3 is first sent to S_2 and then relayed to S_1 by S_2. Also let S_2 send the update of T_3 to S_1 before the update of any transaction executed at S_2, which has read the update of T_3. Then the combination of values of Y and Z read by T_1 can be only one of the following three, because the value λ_1 is sent to S_1 before the value ν_1 : $Y = \lambda_0$ and $Z = \nu_0$; $Y = \lambda_1$ and $Z = \nu_0$; $Y = \lambda_1$ and $Z = \nu_1$. The execution of these three transactions is serializable in any one of these three cases. Therefore, if the rule mentioned above for update propagation is adhered to, then any execution in the fragmented database of Example 5.4.4 will be globally serializable. This observation has been abstracted and proved formally in [KoG87], as we explain below.

Given a directed graph $G = (N, E)$, a **topological sort** of G is a total order on N such that if A and B are two nodes in N and there is a directed path from A to B in E, then A is ordered before B. Let G be an acyclic RAG of a fragmented database system with fragments $\{F_1, \ldots, F_n\}$ and sites $\{S_1, \ldots, S_n\}$. For simplicity, assume that $F_1, \ldots, F_n$ is a topological sort of G. We define a propagation order in G by a function *Send*, where

$$Send(i) = i - 1, \text{ for } 1 < i < n.$$

Note that *Send* is not defined for $i = 1$, i.e., the first node in the sorted order. The update, $U(T)$, by a transaction T submitted at S_i, which is carried in a packet containing all the updates done by T, is sent to the site indexed by $Send(i)$, i.e., S_{i-1} in this case. Whenever a site S_k ($k \neq 1$) receives an update $U(T)$ of some transaction T, S_k relays it to the site indexed by $Send(k)$, i.e., S_{k-1}. Furthermore, the update $U(T)$ must be sent out by S_k according to the following rule. Let X be a

data object with an update in $U(T)$. If T_k is a transaction executed at S_k, which has read the value of X before X is updated by T, then $U(T)$ must be sent out after the update of T_k. If T_l is a transaction executed at S_k, which has read the value of X in $U(T)$, then $U(T)$ must be sent out before the update of T_l. Thus, *Send* defines a route for the propagation of updates.

For example, F_1, F_2, F_3 is a topological sort for the acyclic RAG in Figure 5.4.4(a). A route for update propagation can be defined in the reverse order. Therefore, updates done in S_3 are sent to S_2 and then relayed to S_1. Updates done in S_2 are sent directly to S_1. The following was proved by Kogan and Garcia-Molina in [KoG87], which deals with the case in which the RAG is acyclic but not loopless.

Theorem 5.4.2.[KoG87] *Given a fragmented database system with an acyclic RAG, if update propagation is controlled by the propagation function Send defined above, then any execution is globally serializable.* □

In the following, we present an example which shows an application of Theorem 5.4.2.

Example 5.4.5 [KoG87]. This is an example of an airline reservation system. The database contains information on flight schedules, customer reservations, and seat assignments. The database is replicated at different sites, including the airports at which this airline operates.

The RAG for this fragmented database is shown in Figure 5.4.5. Fragment F contains the flight schedules which are managed by the central office. Only the central office needs and has the authority to update flight schedules. There are two disjoint fragments for reservation; one for the west coast (fragment R_w), and the other for the east coast (fragment R_e). The sites which manage R_w and R_e are able to read flight schedules in order to accept clients' reservation orders. However, there is no need for the central office which handles flight scheduling to access reservation information. There are three other sites, which are the airlines' offices at three different airports. Fragments A_a,

A_b and A_c, belonging to these three sites S_a, S_b and S_c, respectively, contain seat allocation information. In order to allocate seats, sites S_a, S_b and S_c have to access flight schedules in F and the reservation lists in R_e and R_w. The RAG in this database is acyclic and contains no loop. Therefore, Theorem 5.4.2 is applicable.

High availability is achieved in this fragmented database system. For example, it is possible to assign passengers to their seats at the airports even if the computers located at the airports are cut-off from the rest of the system. Similarly, there is no need for the operators at the airport and those at the reservation centers to execute schedule changes. Therefore, a temporary cut-off from the rest of the system would not stop the central office from updating the flight schedule. □

There are, however, shortcomings in the application shown in Example 5.4.5. Suppose that the propagation order used is the reverse of a topological order A_a, A_b, A_c, R_w, R_e, F. Any change in flight schedule would have to be sent from the central office to the site managing R_e, then relayed to the site managing R_w, then through S_c and S_b, until it arrives at the last site S_a. If a partition failure puts the sites managing F, A_a, A_b and A_c in one partition and the others in another partition, then the update done on F cannot be sent to S_a, S_b or S_c, even though these three sites are connected to the central office. Instead, seat allocation at S_a, S_b and S_c have to use stale data in F that were received prior to the failure. This problem is unavoidable if a static access pattern is adopted. In the next two chapters, we will study this issue. In particular, we will investigate the case in which the RAG may be any directed graph.

CHAPTER 6

A CONCURRENCY CONTROL SCHEME FOR WIDE-AREA DISTRIBUTED DATABASE SYSTEM

6.1. A Model for Wide-Area Distributed Database Systems

In Chapter 5, we discussed Kogan and Garcia-Molina's scheme for achieving high availability in a fragmented database system, which requires the RAG (read access graph) to be acyclic. In their scheme, any transaction, whether it is local or global, is accepted if its read and write operations satisfy the condition imposed by the RAG. Moreover, no global concurrency control is needed, since serializability is guaranteed by the RAG. However, the acyclicity requirement on the RAG severely restricts the applicability of their scheme.

In this chapter, we relax the restriction on access pattern by allowing the RAG to be any directed graph. It is then no longer possible to achieve serializability without any global concurrency control as in Kogan and Garcia-Molina's scheme. However, it will be shown that to maintain serializability global concurrency control is needed only for global transactions. In other words, local transactions can be managed solely by local schedulers, as if they were running in a single-site system.

An interesting feature of our scheme is that a global transaction which reads a data object belonging to a remote site S_i never makes the data object inaccessible to a local transaction submitted at S_i. In other words, a local transaction is never rejected because it wants to access a data

object which is being read by a global transaction submitted at another site. Namely, local transactions have higher priority than global transactions, if they happen to be competing for the same data object. Also, in our scheme, even if there is a partition failure, any local transaction submitted at an operational site can still be accepted. In this way, local transactions enjoy good response time and high availability. As will become evident in the following, this approach is very useful for a wide-area distributed database system.

In many distributed database systems, participating sites are spread over a geographically wide area. This kind of database system is called a **wide-area distributed database system (WADDS**, for short). In some cases, these sites may even be located on different continents. Communication among the sites in a wide-area distributed database system is conducted through a long haul network and the delay for message transmission across the network might well be on the order of seconds. (For example, it may take several hops of satellite transmission to reach a receiver). In the remainder of this thesis, we consider only fully replicated wide-area distributed database system. The reason for replication is reliability and "accessibility".

When a transaction updates a data object in a conventional replicated database system, besides the local copy, it also must update all affected remote copies before its completion. With communication delays on the order of seconds, these remote updates will undoubtedly cause delays that are unacceptable in many applications. This is why the conventional way of transaction processing is inadequate [NoA83] and new models and schemes for a wide-area distributed database system are investigated.

The model of fragmented database system solves some of the problems raised in a wide-area distributed database system. For example, in a fragmented database system, both local and global transactions complete their executions by accessing local copies available at their home sites; updates are then broadcast asynchronously afterwards. This eliminates the problem of prolonged

waiting for the completion of remote updates. Hence, the updating strategy used in a fragmented database is quite suitable for wide-area distributed database systems.

Besides update propagation, the issue of concurrency control has to be solved in a wide-area distributed database system. One option is to adopt a conventional scheme like the distributed locking scheme. Another option is to adopt the scheme proposed by Kogan and Garcia-Molina which we discussed in Chapter 5. Suppose that we adopt the distributed locking scheme for concurrency control. If a data object is locked by a global transaction, then all local transactions accessing the data object must wait until the global transaction is completed. Because of communication delay, the completion of a global transaction may take a long time. This significantly degrades the performance of local transactions. Therefore, the first option is not practical. The scheme proposed by Kogan and Garcia-Molina will work well with a wide-area distributed database system, except that it requires the underlying RAG to be acyclic, which is too restrictive for general application. In the following, we will propose a new scheme for concurrency control for a wide-area distributed database system, which is based on the fragmented database model.

The difference between our scheme and Kogan and Garcia-Molina's scheme (KG's scheme, for short) are as follows.

(1) The underlying RAG in KG's scheme must be acyclic, while the RAG in our scheme can be any directed graph. In general, in our scheme, we assume that the RAG is a complete graph. Therefore, a transaction submitted at one site can read the data object from any set of remote fragments.

(2) We assume that much more local transactions than global transactions are submitted to the system. In KG's scheme, there is no explicit assumption on the ratio of local transactions to

global transactions.

(3) There is no explicit global concurrency control in KG's scheme except for the constraints imposed by the RAG. In our scheme, no global concurrency control is needed for local transactions; however, it is needed for global transactions in order to maintain serializability among all the transactions. (This will be discussed extensively in Section 6.3).

The difference mentioned in (1) implies that our scheme is more general than KG's scheme. We believe that the assumption mentioned in (2) is usually valid in wide-area distributed database systems. As a matter of fact, it is quite usual that most transactions access only their home fragments. For example, in a banking database system, it is likely that most of the transactions submitted at a particular branch access only the accounts opened at that branch. As for (3), KG's scheme imposes such a strict constraint that global concurrency control is not needed. Since we impose no restriction on the RAG in our scheme, we cannot do away with global concurrency control. However, the merit of our scheme is that global concurrency control needs to be applied only to global transactions. Therefore, the large number of local transactions can be synchronized only by the local schedulers at their home sites as if they were running in a single-site database system. Our scheme also ensures that a local transaction has a higher priority than a global transaction if they are competing for the same data object. This guarantees that all local transactions enjoy good response time and high availability.

The following example illustrates a wide-area fragmented distributed database system for inventory and price control with a complete RAG.

Example 6.1.1. A manufacturing company does business in many districts spread over several continents. In each district, it has an office and a plant. The plant is responsible for three aspects of one line of product; manufacturing, inventory control and pricing. The office manages the local sale

and local inventory of all the products for the district.

In terms of our model, there are n sites $S_1, \ldots, S_n$ and each site S_i owns a fragment F_i ($i = 1, \ldots, n$). In each fragment F_i, there are three sets of data objects. The first set, $INVENT_i$, is the inventory of all the products in the district controlled by S_i. The second set, $TOTAL_i$, is the total inventory over all districts of the line of product manufactured at S_i. The third set, $PRICE_i$, is the price list of the products manufactured at S_i.

Data objects are fully replicated according to the fragmented database system model. Each site S_i must read from $INVENT_j$, for $j = 1, \ldots, n$, in order to update $TOTAL_i$. Then S_i uses $TOTAL_i$ as an indication of the market demand to adjust $PRICE_i$. Also, each site S_i has to read $PRICE_j$, $j = 1, \ldots, n$ to determine the prices of all the products for local sales. Therefore, the RAG for this system is a complete graph. Figure 6.1.1 shows the RAG of this inventory system, assuming that there are only three sites. □

In the following example, we will explain several cases in which an execution consisting of local and global transactions may or may not be serialized.

Example 6.1.2. Suppose that there are two sites S_1 and S_2 in a fragmented database system and the schedulers at S_1 and S_2 are timestamp-based schedulers [BeG81]. Let A_1 and A_2 be the sets of transactions executed at S_1 and S_2, respectively. A transaction in A_1 (A_2) is represented by T_{1i} (T_{2i}), where i is its local timestamp. Timestamps are assigned to the transactions in A_1 and A_2 independently, i.e., a transaction in A_1 may have the same timestamp as another one in A_2. An update sent from a site to another site has the timestamp of the transaction by which this update was written, and this timestamp is stored together with the update in the database at the receiving site.

In case (a), $A_1 = \{T_{11}, T_{12}, T_{13}\}$, $A_2 = \{T_{21}, T_{22}, T_{23}\}$ and all the transactions are local transactions. (See Figure 6.1.2(a).) Since the schedulers are timestamp-based, $(T_{1i})_{i=1}^{3}$ is a

serialization order of the transactions in A_1. Similarly, $(T_{2i})_{i=1}^{3}$ is a serialization order of the transactions in A_2. Since a local transaction reads and writes only data objects in its home fragment, any order of the transactions in $A_1 \cup A_2$, in which the orders $(T_{1i})_{i=1}^{3}$ and $(T_{2i})_{i=1}^{3}$ are preserved, is a serialization order.

In case (b), A_1 is same as in case (a) and A_2 has one more global transaction T_{24} than it has in case (a). (See Figure 6.1.2(b).) In Figure 6.1.2(b), there is an edge from T_{12} to T_{24}, indicating that T_{24} has read a data object from T_{12}. Since T_{21}, T_{22}, T_{23} do not update the data objects written by T_{12}, the order $(T_{11}, T_{12}, T_{21}, T_{22}, T_{23}, T_{24}, T_{13})$ is a serialization order of the transactions in $A_1 \cup A_2$.

In case (c), A_2 has one more local transaction T_{25} than it has in case (b). A_1 has one more global transaction T_{14} than it has in case (b) and T_{14} has read a data object X from T_{25}. (See Figure 6.1.2(c).) It can be seen that the order $(T_{11}, T_{12}, T_{21}, T_{22}, T_{23}, T_{24}, T_{13}, T_{25}, T_{14})$ is a serialization order of the transactions in $A_1 \cup A_2$.

In case (d), A_1 and A_2 have the same contains as in case (c); however, T_{14} reads X from T_{22}, but not from T_{25}. In this case, the transactions in $A_1 \cup A_2$ may not be serializable. When T_{14} was executed at S_1, it read the local copy of X; hence the value of X may not be current, i.e., X may have been updated again at S_2. The only information S_1 knows is that the value of X read by T_{14} is written by a transaction at S_2, whose timestamp is 2. We do not assume S_2 records the writesets of the transactions executed there. Hence after T_{23} has completed, neither S_1 nor S_2 knows that if T_{23}, the transaction serialized immediately after T_{22} at S_2, has written X. Therefore, in order to serialize T_{14} with those at S_2, it would require T_{14} to be serialized after T_{22} and before T_{23}. Similarly T_{24} would be required to be serialized after T_{12} and before T_{13}. However, T_{12} and T_{23} are serialized before T_{14} and T_{24}, respectively. Therefore, there would be a cycle in the serialization order if we try to serialize the transactions in this way. □

In the above example, a way to avoid the problem occurred in case (d) is to send the timestamp of X read by T_{14} to S_2 for **certification.** The idea of certification is to ensure that the transactions in $A_1 \cup A_2$ can be serialized by knowing only the timestamps of the data objects read by global transactions from remote sites. In the above example, we have shown only a few cases in which the read operation of a global transaction has to be certified. It is not yet clear how to perform certification algorithmically. In Section 6.4, we will explain the details of certification.

In Section 6.2, we will present an architecture for wide-area distributed database systems. In Section 6.3, we will model an execution generated in a wide-area distributed database system by a **fragmented execution.** We will also discuss a sufficient condition for a fragmented execution to be serializable. In Section 6.4, we will propose a concurrency control, called **Global Timestamp Order Certification** (GTOC), for a wide-area distributed database with the properties mentioned above. In Section 6.5, the performance of GTOC is compared with other conventional schemes. In Section 6.6, we will discuss partition failures in a fragmented database.

6.2. Architecture for Wide-Area Distributed Database System (WADDS)

A single-site database system, in general, consists of four components : the **transaction manager** (TM), the **scheduler**, the **data manager** (DM) and the **database** (DB). Their relationships are shown in Figure 6.2.1. The TM is an interface between the system and user transactions. The operations of transactions submitted to the TM are rearranged by the scheduler. The scheduler uses concurrency control to ensure that the executions generated are serializable. The operations granted by the scheduler are executed by the DM on the DB. The data values retrieved by the DM are sent back to a workspace in the TM via the scheduler for processing.

The architecture proposed for a WADDS is illustrated in Figure 6.2.2, which is a modification of the architecture in Figure 6.2.1. In the architecture for WADDS, the TM contains two subcomponents : the **local transaction manager** (LTM) and the **global transaction manager** (GTM). The LTM and the GTM are responsible for managing local and global transactions, respectively. The DB is divided into two parts : one part contains the data objects belonging to the home fragment and the other part contains the copies of those belonging to the remote fragments. The scheduler contains three subcomponents : the **home fragment scheduler** (HFS), the **remote fragment scheduler** (RFS) and the **global synchronizer** (GS). The HFS and RFS synchronize operations accessing the home fragment and remote fragments, respectively. The HFS is a timestamp-based scheduler. The RFS can be any scheduler that maintains serializability, e.g., a timestamp-based scheduler. Operations granted by the HFS and the RFS are executed by the DM on the DB. As mentioned in Section 6.1, in our scheme only global transactions have to be synchronized by global concurrency control. More precisely, only read operations of global transactions, which access data objects in remote fragments, have to be synchronized by global concurrency control. The synchronization is done by **certifying** the readings done by global transactions, as explained in detail in Section 6.4. The subcomponent GS in the scheduler is responsible for certification. After a global transaction has finished reading from remote fragments by accessing local copies in the DB, the values read have to be certified by the GS. The GS has to ensure that the global transaction can be serialized with the local and global transactions at other sites. The GS's at all the sites achieve this goal by running a distributed algorithm which will be discussed in Section 6.4. Besides the four components, i.e. the TM, the scheduler, the DM and the DB, there are two additional components. The first is the **update propagation manager** (UPM) which broadcasts the updates by transactions after their completion. The second is the **communication manager** (CM), which is linked to the CM's at all other sites and is responsible for

sending and receiving messages.

Local transactions are managed by the LTM and their operations are synchronized by the HFS. The operations granted by the HFS are executed by the DM and the values retrieved from the home fragment are stored in a workspace in the LTM. When a local transaction commits, its updates are installed permanently in the home fragment. Up to this point, a local transaction is processed in exactly the same way as a transaction in a single site database. After a local transaction has committed its updates at its home site, its updates are then broadcast by the UPM to the UPM's at all other sites via the CM. The UPM at each site installs the updates received from other sites in the copies of the remote fragments by submitting them to the RFS.

The read and write operations of a global transaction, which access data objects in its home fragment, are processed in the same way as an operation of a local transaction. As for the read operations which access data objects in remote fragments, they are synchronized by the RFS and executed by the DM on the local copies of the remote fragments. As mentioned above, the results of these reading operations from remote fragments have to be certified by the GS. If the certification is successful, the global transaction proceeds to complete its remaining part. Then its updates are broadcast by the UPM in the same way as a local transaction. If the certification fails, the global transaction is aborted.

In the following, we describe in detail the processing of a local and a global transaction in the architecture described above.

A local transaction can request four kinds of operations : LOCAL-BEGIN, READ, WRITE and END. Each local transaction starts with a LOCAL-BEGIN and ends with an END. In between these two operations is a sequence of READ and WRITE operations. These operations are interpreted in the usual way [BeG81]. A global transaction, on the other hand, starts with a

GLOBAL-BEGIN, followed by a GLOBAL-READ, then a sequence of READ and WRITE operations, and ends with an END. GLOBAL-READ($X, Y, \cdots$) is a request to read data objects belonging to some remote fragment(s). READ and WRITE operations, on the other hand, access only data objects in the transaction's home fragment.

When a LOCAL-BEGIN operation from a local transaction T arrives at the TM, the LTM initializes a temporary workspace for T. The first thing that the LTM does is to retrieve a timestamp from a local clock and assign it to T. Requests from T are sent via the LTM to the HFS with the timestamp of T. The HFS uses a timestamp-based algorithm to synchronize the requests from the LTM. Every data object in the home fragment has a read-timestamp and a write-timestamp, which indicate the timestamps of the transactions which last read and wrote the data object, respectively. The granted requests are then submitted to the DM. Data retrieved from the DB is sent back to the TM and stored in T's workspace.

When the END operation arrives at the LTM, it initiates two actions. Firstly, it sends a commit message to the DM via the HFS to commit the updates in the DB. Secondly, it compiles a remote update request and sends it to the UPM. The remote update request contains a vector $<x, V_x, y, V_y, \cdots, t>$, where $x, y, \cdots$, are the names of data objects, $V_x, V_y, \cdots$, are their updated values, and t is the timestamp of T. The UPM implements a reliable broadcast protocol which ensures that (1) a remote update request is broadcast to every other site and it will be received eventually even if the network experiences a temporary failure, and that (2) the remote update requests from a site are sent and received by each receiver in their timestamp order [AwE84, GLB85]. The UPM sends out its messages via the CM. Upon receiving a remote update request from another site, the UPM converts it into an internal update transaction which is then submitted to the RFS. The RFS can use any algorithm which guarantees serializability and preserves the timestamp order of the remote updates. When a receiver's RFS installs a remote update request,

it also attaches the timestamp of the request to all the updated data objects in the DB. At each site S_i, every data object has one or two timestamps. If a data object belongs to S_i, it has read and write timestamps. Otherwise, it is a copy of a data object belonging to a remote fragment, and it has the timestamp of the last remote update. The timestamps from different sites are completely independent of each other. This doesn't cause any problem, since data in a fragment can be updated only by transactions submitted at its home site and they get their timestamps from the same clock.

When the GLOBAL-BEGIN operation of a global transaction T arrives at the TM, the GTM initializes a workspace for T. The second operation from T must be a GLOBAL-READ($X, Y, \cdots$), and the GTM interprets this as a read-only transaction which reads the copies named by the parameters. The GTM initiates two consecutive steps to execute the GLOBAL-READ. Firstly, a read request for every parameter is sent to the RFS, and in response to it, both the value of the data object and its timestamp are retrieved by the DM and sent back to the GTM via the scheduler. At this point, the GTM does not know whether the read operations just performed can be serialized with the operations at the other sites. Therefore, the second step is to activate the GS to certify these read operations. The GS first generates a set of certification requests, one for each remote fragment from which some data objects were read by the GLOBAL-READ operation. Then it sends these certification requests to the GS's in the home sites of the remote fragments involved. The local GS and the remote GS's involved use a distributed algorithm to decide whether the values retrieved by the GLOBAL-READ operation can be certified. (One possible certification algorithm is described in Section 6.4). If they cannot be certified, then the GTM is informed by the local GS of this fact and the transaction is aborted. If they are certified, the execution of the remaining part of the global transaction is identical to that of a local transaction. The GTM will now request a timestamp from the local clock and attach it to the remaining READ and WRITE operations of the global transaction and send them to the HFS. (Note that unlike a local transaction, to which a timestamp is assigned

before any operation is submitted to the scheduler, a global transaction is assigned a timestamp after its GLOBAL-READ operation is completed.) When the GTM receives an END, it commits the transaction's updates in the DB. The GTM then generates a remote update request and sends it to the UPM in the same way as the LTM handles remote updates for local transactions.

Note that the GS's have no knowledge about the actions taken by the HFS's. The GS's can use only the information supplied to it in the certification requests to certify GLOBAL-READ's. The separation of the GS from the HFS ensures that the GS at one site never interferes with the HFS's at the other sites. In other words, while a global transaction is executing a GLOBAL-READ on a data object X, any local transaction submitted at the home site of X can still access X. Therefore, local transactions will never be blocked by a global transaction. However, it is mandatory for the GS's at all sites to ensure that the execution generated, which consists of operations from local and global transactions, is serializable. In Section 6.3, we will derive a sufficient condition for the serializability of an execution generated in this architecture. In Section 6.4, we will propose an algorithm for the GS's to use to satisfy the condition.

6.3. Correctness of Fragmented Executions

In this section, we will model the execution of transactions in a fragmented database by a **fragmented execution**, and present a sufficient condition for a fragmented execution to be serializable.

Let ξ be a global execution, i.e., a rp log over a set of global and local transactions, in a fragmented database over a set of sites $S = \{S_i : i = 1, \ldots, n\}$. In the following, we assume that every global transaction involved in ξ accesses some data object(s) in its home fragment. A global transaction that accesses only remote fragment(s) can be modified to satisfy this assumption by

putting a dummy read operation in it to access its home fragment. For each site S_i, the **subexecution** ξ_i of ξ at S_i is a poset which contains all the operations in ξ that are *executed* at S_i. The partial order among the operations in ξ_i is inherited from the partial order in ξ. Note that ξ_i, in general, contains operations from both local and global transactions. Let $\hat{\xi}_i$ be the poset containing all the operations in ξ_i that *access* S_i's home fragment. The partial order among the operations in $\hat{\xi}_i$ is inherited from ξ_i. Note that $\hat{\xi}_i$ contains all the operations executed at S_i except for those that access the copies of remote fragments of S_i. We assume that there is a fictitious **initial** transaction T_{i0} which is executed before any operation at S_i and writes all the data objects in the home fragment of S_i. Execution ξ is **locally serializable** at S_i if $\hat{\xi}_i$ is equivalent to a 1C serial log of all the transactions involved in $\hat{\xi}_i$. (Please refer to Section 2.4 for the meaning of equivalence.)

Definition 6.3.1. A global execution $\xi = (\Sigma(\mathrm{T}), <)$ over a set of sites $\mathrm{S} = \{S_i : i = 1, \ldots, n\}$ is a **fragmented execution** if (1) ξ is locally serializable at S_i, for $i = 1, \ldots, n$, and (2) for each site S_i, there is a serialization order σ_i of the transactions in $\hat{\xi}_i$ such that for any two transactions T_a and T_b in $\hat{\xi}_i$, if T_a precedes T_b in $\hat{\xi}_i$ and $W_a[X]$ and $W_b[Y]$ are, respectively, two write operations of T_a and T_b, then, for each site S_j, $j \neq i$, $W_a[X_j]$ and $W_b[Y_j] \in \Sigma(\mathrm{T})$ and $W_a[X_j] < W_b[Y_j]$, where $W_a[X_j]$ and $W_b[Y_j]$ are the execution of $W_a[X]$ and $W_b[Y]$ on the copies at S_j, respectively. The serialization order σ_i in (2) is called the **broadcast order** at S_i. □

In Definition 6.3.1, (2) has an intuitive meaning. It indicates that the updates of the transactions in σ_i are broadcast in their order in σ_i. If a global execution ξ is locally serializable at a site S_i, there may be more than one serialization order for $\hat{\xi}_i$. If ξ is a fragmented execution, one of these serialization orders is the broadcasting order at S_i. Note that a fragmented execution has n initial transactions, one for each site. This is different from a general rp log.

In defining a fragmented database system in Section 5.3, all transactions submitted at a site are synchronized by a local scheduler and the updates of these transactions are broadcast in a

serialization order of these transactions. Hence every execution generated in a fragmented database system is a fragmented execution. In a WADDS with the architecture given in Section 6.2, the HFS at a site is a timestamp-based scheduler. Hence, all transactions submitted at a site can be serialized by their timestamps and their updates are broadcast in this timestamp order. Therefore, any execution generated in a WADDS is a fragmented execution.

We use an example to illustrate the notion of fragmented execution.

Example 6.3.1. Let S_1 and S_2 be two sites of a WADDS which has the architecture given in Section 6.2. Let $\{X, Y\}$ and $\{A, B\}$ be data objects belonging to the home fragments of S_1 and S_2, respectively. Let ξ be an execution over $\{S_1, S_2\}$. Suppose that subexecution ξ_1 of ξ is

$$W_{10}[X_1, Y_1]R_{11}[X_1]R_{12}[A_1]W_{11}[X_1]R_{12}[X_1]R_{13}[Y_1]W_{12}[X_1]W_{13}[Y_1].$$

According to the architecture of WADDS, every transaction has a timestamp assigned by the LTM or GTM at its home site. In ξ, the second index of an operation is the timestamp of the corresponding transaction. Note that, even though the RFS and HFS are two different subcomponents of the scheduler at a site, the operations granted by them are submitted to the DM sequentially. Therefore, the output from the scheduler is a totally ordered set of operations. Among the transactions involved in ξ_1, T_{11}, T_{13} are local and T_{12} is global. Suppose that subexecution ξ_2 of ξ granted by the scheduler at S_2 is

$$W_{20}[A_2, B_2]R_{21}[A_2]R_{22}[B_2]W_{21}[A_2]W_{22}[B_2]R_{23}[X_2]R_{23}[Y_2]W_{23}[B_2].$$

Among the transactions involved in ξ_2, T_{21}, T_{22} are local, and T_{23} is global. Assume that the read operation $R_{12}[A_1]$ of T_{12} in ξ_1 reads the value of A written by T_{21} in ξ_2. In other words, the update of T_{21} is sent to S_1 and A_1 is updated accordingly before T_{12} reads A_1. Similarly, suppose that the read operations $R_{23}[X_2]$ and $R_{23}[Y_2]$ of T_{23} in ξ_2 read the values of X and Y written by T_{12} and T_{13} in ξ_1, respectively. (For simplicity, when we listed the operations of ξ_1 in the above, update operations

from S_2 such as $W_{21}[A_1]$ were not shown. Similarly, when we listed the operations of ξ_2, update operations from S_1 such as $W_{12}[X_2]$ were not shown.)

Note that

$$\hat{\xi}_1 = W_{10}[X_1, Y_1]R_{11}[X_1]W_{11}[X_1]R_{12}[X_1]R_{13}[Y_1]W_{12}[X_1]W_{13}[Y_1]$$

is equivalent to the serial log

$$W_{10}[X_1, Y_1]R_{11}[X_1]W_{11}[X_1]R_{12}[X_1]W_{12}[X_1]R_{13}[Y_1]W_{13}[Y_1].$$

As a matter of fact, the above equivalency is guaranteed by the timestamp scheduler HFS at S_1 and the transactions are ordered by their timestamps in the above serial log. Hence, ξ is locally serializable at S_1. Let σ_1 represent the timestamp order of the transactions in ξ_1, i.e., $\sigma_1 = (T_{10}, T_{11}, T_{12}, T_{13})$. The updates of the transactions in σ_1 are sent to S_2 by their order in σ_1. Similarly,

$$\hat{\xi}_2 = W_{20}[A_2, B_2]R_{21}[A_2]R_{22}[B_2]W_{21}[A_2]W_{22}[B_2]W_{23}[B_2]$$

is serializable and a serialization order is given by $\sigma_2 = (T_{20}, T_{21}, T_{22}, T_{23})$. S_2 broadcasts the updates of the transactions in σ_2 by their order in σ_2. Hence ξ is locally serializable at both S_1 and S_2, and $\{\sigma_1, \sigma_2\}$ are the broadcast orders at S_1, S_2, respectively. Thus ξ is a fragmented execution over $\{S_1, S_2\}$. □.

We shall now investigate under what condition a fragmented execution is (globally) serializable. Given a set of sequences of transactions $\{\sigma_i = (T_{ij})_{j=0}^{n_i} : i = 1, \ldots, n\}$, let $\text{ID} = \{ij : i = 1, \ldots, n, j = 0, \ldots, n_i\}$ be the set of indices of the transactions in σ_i, $i = 1, \ldots, n$. A **merge** of $\{\sigma_i : i = 1, \ldots, n\}$ is a sequence of transactions $(T_{h(i)})_{i=1}^{m}$, where $m = \sum_{i=1}^{n} n_i + n$ and $h : \{1, \ldots, m\} \rightarrow \text{ID}$ is a permutation on ID such that, if $T_{h(p)}$ is ordered before $T_{h(q)}$ in σ_i, then $p < q$.

Definition 6.3.2 A fragmented execution ξ over a set of sites $S = \{S_i : i = 1, \ldots, n\}$ with a set of broadcast orders $BO = \{\sigma_i = (T_{ij})_{j=0}^{n_i} : i = 1, \ldots, n\}$ is said to be **serializable** if there exists a merge $\sigma = (T_{h(i)})_{i=1}^{m}$ of the sequences $\{\sigma_i : i = 1, \ldots, n\}$, such that ξ is equivalent to the 1C log $[T_{h(1)} \cdots T_{h(m)}]$. □

In Definition 6.3.2, the initial transactions T_{i0}, $i = 1, \ldots, n$, are also included in the sequence σ, and they are not necessarily the first n transactions in σ.

Example 6.3.2. Consider the fragmented execution ξ in Example 6.3.1. Let $BO = \{\sigma_1, \sigma_2\}$, where $\sigma_1 = (T_{10}, T_{11}, T_{12}, T_{13})$ and $\sigma_2 = (T_{20}, T_{21}, T_{22}, T_{23})$. Let σ be the sequence

$$T_{10}, T_{20}, T_{11}, T_{21}, T_{12}, T_{22}, T_{13}, T_{23}.$$

In ξ, T_{12} reads A from T_{21}, T_{23} reads X and Y from T_{12} and T_{13}, respectively. Let L be the 1C serial log $[T_{10} T_{20} T_{11} T_{21} T_{12} T_{22} T_{13} T_{23}]$ corresponding to σ which is a merge of σ_1 and σ_2. In L, no transaction is ordered between T_{21} and T_{12}. Therefore, T_{12} reads A from T_{21} in L. Since T_{22} can update only the data objects belonging to S_2, therefore T_{23} reads X and Y from T_{12} and T_{13} in L, respectively. It follows that ξ is equivalent to the 1C serial log L. Hence ξ is serializable. □

Given a fragmented execution ξ over a set of sites $S = \{S_i : i = 1, \ldots, n\}$, let $BO = \{\sigma_i = (T_{ij})_{j=0}^{n_i} : i = 1, \ldots, n\}$ be the set of its broadcast orders. Let us consider if we can merge $\{\sigma_i : i = 1, \ldots, n\}$ into a sequence $\sigma = (T_{h(i)})_{i=1}^{m}$ such that ξ is equivalent to the 1C log $[T_{h(1)} \cdots T_{h(m)}]$. For a global transaction T_{ij} in σ_i and $k \neq i$, let $RF(T_{ij}, k) = \{T_{kp} : T_{ij}$ reads a data object from $T_{kp}\}$. Thus, $RF(T_{ij}, k)$ is the set of transactions belonging to site S_k from which T_{ij} reads some data object(s). In Example 6.3.1, $RF(T_{12}, 2) = \{T_{21}\}$ and $RF(T_{23}, 1) = \{T_{12}, T_{13}\}$. ($RF(T_{ij}, k)$ is empty if T_{ij} does not read any data object from S_k.) Note that the updates of the transactions in σ_k are broadcast and received in their order in σ_k. Suppose that T_{kl} is the last transaction in σ_k that belongs to $RF(T_{ij}, k)$. If T_{ij} reads X from a transaction $T_{ka} \in RF(T_{ij}, k)$, $a < l$, then no transaction

ordered between T_{ka} and T_{kl} in σ_k writes X. If T_{km}, a transaction ordered between T_{ka} and T_{kl} in σ_m wrote X, then the update on X written by T_{km} would be received by S_i after that of T_{ka} and before that of T_{kl}. Therefore, if T_{ij} has read the update of T_{kl}, it would have read the update of X from T_{km}, not from T_{ka}. Hence, if T_{ij} is serialized with the transactions in σ_k by inserting it after T_{kl} and before the transaction following T_{kl} in σ_k, then all the read-from relations between T_{ij} and the transactions in RF(T_{ij}, k) are preserved. Therefore, to preserve the read-from relationships between T_{ij} and all the transactions in RF(T_{ij}, k) when T_{ij} is merged into σ_k, T_{kl} plays a crucial role. In merging σ_1 and σ_2 in Example 6.3.2, we inserted T_{12} after T_{21}, and T_{23} after T_{13}.

Based on the above observation, we define the **global-read** relation for a fragmented execution containing only those "crucial" read-from relationships. Given a fragmented execution ξ over a set of sites $S = \{S_i : i = 1, \ldots, n\}$ and its broadcast orders $BO = \{\sigma_i = (T_{ij})_{j=0}^{n_i} : i = 1, \ldots, n\}$, the **global-read** relation Gr(ξ) in ξ is a set of all ordered pairs (T_{kl}, T_{ij}), $k \neq i$, such that T_{ij} is a global transaction in σ_i that reads a data object from a transaction T_{kl} in σ_k and $l = max\{p : T_{kp} \in \text{RF}(T_{ij}, k)\}$. In other words, (T_{kl}, T_{ij}) belongs to Gr(ξ), if and only if T_{ij} is a global transaction and T_{kl} is the last transaction in σ_k, from which T_{ij} has read some data objects. In Example 6.3.2, Gr(ξ) = $\{(T_{21}, T_{12}), (T_{13}, T_{23})\}$. Even though T_{23} has read X from T_{12}, since T_{12} is ordered before T_{13} in σ_1, $(T_{12}, T_{23}) \notin$ Gr(ξ).

In the following, we will associate a directed graph with a fragmented execution. Let ξ and BO be defined as above, and consider a transaction T_{ij} in σ_i. The global transaction **preceding** T_{ij} is T_{ij} itself, if it is a global transaction; otherwise it is the last global transaction before T_{ij} in σ_i, if any. The global transaction **following** T_{ij} is the first global transaction after T_{ij} in σ_i, if any. If a global transaction T_{ij} is followed by a global transaction T_{ik}, then T_{ij} and T_{ik} are two **consecutive** global transactions, and there *may be* some information flow from T_{ij} to T_{ik}.

As mentioned in Section 6.1 we will show that it is possible to achieve serializability in a fragmented database by synchronizing only global transactions. To this end, we now introduce a graph to extract the information related to the global transactions. For a fragmented execution ξ over a set of sites $S = \{S_i : i, \ldots, n\}$, let $BO = \{\sigma_i = (T_{ij})_{j=0}^{n_i} : i = 1, \ldots, n\}$ be its broadcast orders, Gr(ξ) be the global-read relation of ξ, GT be the set of global transactions involved in ξ, and IT be the set of initial transactions $\{T_{i0} : i = 1, \ldots, n\}$. The **global-serialization graph** (**GOS graph** , for short) for ξ, denoted by $GOS(\xi)$, is a directed graph (N, E), where N contains a node for each transaction $T_{ij} \in GT \cup IT$, and E contains the following four sets of edges. (T_{ij} is used to represent both a transaction and the node representing it in the GOS graph. For intuitive justification of the edges, see the next paragraph.) (1) There is a **precedence edge** from each transaction $T_{ij} \in GT \cup IT$ to the global transaction T_{ik} following it, if any. (Please see Figure 6.3.1(a).) (2) For each $T_{ij} \in GT$ and $(T_{kl}, T_{ij}) \in Gr(\xi)$, if T_{ka} is the global transaction preceding T_{kl}, then there is a **global-read edge** from T_{ka} to T_{ij}; if no global transaction precedes T_{kl}, then there is a global-read edge from T_{k0} to T_{ij}. (Please see Figure 6.3.1(b).) (3) For each $T_{ij} \in GT$, if $(T_{kl}, T_{ij}) \in Gr(\xi)$ and T_{kb} is the global transaction following T_{kl}, then there is an edge from T_{ij} to T_{kb}. (Please see Figure 6.3.1(c).) (4) For T_{ij}, $T_{mn} \in GT$, if both (T_{kp}, T_{ij}) and (T_{kq}, T_{mn}) belong to Gr(ξ), $p < q$, and the global transactions preceding T_{kp} and T_{kq} is identical, or no global transaction precedes them, then there is an edge from T_{ij} to T_{mn}. (Please see Figure 6.3.1(d).) The edges introduced by (3) and (4) are called **induced edges.**

We now give some intuitive meanings for the four sets of edges introduced above in a GOS graph. The first set are precedence edges. In a merge of $\{\sigma_i : i = 1, \ldots, n\}$, the relative positions of an initial transaction and the global transaction following it should not change. This is also true for two consecutive global transactions; information may flow from a global transaction to the global transaction following it. Precedence edges are used in a GOS graph to represent these orders.

Let us consider the set of edges (2) and (3). Suppose $(T_{kl}, T_{ij}) \in \mathrm{Gr}(\xi)$ and T_{ka}, T_{kb} are the global transactions preceding and following T_{ij}, respectively. Note that T_{kl} is last transaction in σ_k from which T_{ij} has read some data object(s). Since site S_i has no control over the local transactions submitted at S_k, we assume that S_i has no knowledge about the writesets of the local transactions that come after T_{kl} in σ_k. In other words, if T_{ij} has read X from T_{kl}, then it is not known to S_i whether the transaction after T_{kl} in σ_k also writes X. Hence, T_{ij} is serialized after T_{kl} and before the transaction following T_{kl} in σ_k. This implies that T_{ij} should follow T_{ka} and precede T_{kb} in a global serialization order.

As for the set of edges (4), suppose that both (T_{kp}, T_{ij}) and (T_{kq}, T_{mn}), $p < q$, belong to Gr(ξ), and the global transaction preceding T_{kp} and T_{kq} are identical or no global transaction precedes them. Let T_{ka} be the global transaction preceding T_{kp} and T_{kq}, if any, or T_{k0}, otherwise. There are two global-read edges directed from T_{ka} to T_{ij} and T_{mn}. Following the same line of reasoning as given in the previous paragraph, T_{ij} should be serialized after T_{kp} and before the transaction that comes after T_{kp} in σ_k, and T_{mn} is serialized after T_{kq} and before the transaction that comes after T_{kq} in σ_k. Since $p < q$, T_{kp} is ordered before T_{kq} in σ_k and thus T_{kp} must be serialized before T_{kq}. Hence T_{ij} is serialized before T_{kq}. Therefore, T_{ij} should be serialized before T_{mn} in a global serialization order. Thus the fourth set of edges are introduced. We show below an example of a GOS graph.

Example 6.3.3 Consider a fragmented database with four sites S_1, S_2, S_3 and S_4. The set broadcast orders BO = $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ and global-read relation Gr(ξ) of a fragmented execution ξ over $\{S_1, S_2, S_3, S_4\}$ are defined as follows.

$$\sigma_1 = T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{17}, T_{18}, T_{19},$$

$$\sigma_2 = T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{25},$$

$$\sigma_3 = T_{30}, T_{31}, T_{32}, T_{33},$$

$$\sigma_4 = T_{40}, T_{41}, T_{42}, T_{43}, T_{44}, T_{45}.$$

$$Gr(\xi) = \{(T_{25}, T_{17}), (T_{12}, T_{23}), (T_{14}, T_{33}), (T_{43}, T_{33}), (T_{19}, T_{45})\}.$$

$\{T_{17}, T_{23}, T_{33}, T_{45}\}$ and $\{T_{10}, T_{20}, T_{30}, T_{40}\}$ are the set of global transactions and the set of initial transactions in ξ, respectively. Figure 6.3.2 illustrates the GOS graph $GOS(\xi)$. The precedence edges between an initial transaction and the global transaction following it are placed horizontally in the graph. Since there are four sites, there are four such precedence edges in $GOS(\xi)$. There is no other precedence edge in $GOS(\xi)$, because there are no consecutive global transactions in ξ. For the purpose of illustration, for each site S_i, $i = 1, \ldots, 4$, the local transactions in σ_i ordered after T_{i0} and before the global transaction following it are placed on the corresponding precedence edge in Figure 6.3.2. For example, the local transactions $T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}$ are placed on the precedence edge from T_{10} to T_{17}. In the graph, for each element (T_{kl}, T_{ij}) in Gr(ξ), the global-read edge associated with it is labeled by l, the second index of the first element. Since $(T_{12}, T_{23}) \in$ Gr(ξ) and no global transaction is preceding T_{12}, there is a global-read edge from T_{10} to T_{23} labeled by 2. Since T_{17} is the global transaction following T_{12}, there is an induced edge from T_{23} to T_{17}. Similarly, there are global-read edges from T_{23} to T_{17} labeled by 5, from T_{10} to T_{33} labeled by 4, from T_{40} to T_{33} labeled by 3, and from T_{17} to T_{45} labeled by 9. There are induced edges from T_{33} to T_{17} and T_{45}. Also, since both (T_{12}, T_{23}), $(T_{14}, T_{33}) \in$ Gr(ξ) and no global transaction is preceding T_{12} or T_{14}, there is an induced edge from T_{23} to T_{33}. □

Theorem 6.3.1. *A fragmented execution ξ over a set of sites $S = \{S_i : i = 1, \ldots, n\}$ is serializable, if the GOS graph $GOS(\xi)$ is acyclic.*

Example 6.3.4. Before giving a proof for Theorem 6.3.1, as an example, we first construct a 1C serial log equivalent to the fragmented execution ξ in Example 6.3.3. The GOS graph for ξ in Figure 6.3.2 is acyclic, and $T_{10}, T_{20}, T_{30}, T_{40}, T_{23}, T_{33}, T_{17}, T_{45}$ is a topological sort. Now we will show that the local transactions can be inserted into this sequence to create a sequence σ which is a merge of $\{\sigma_i : i = 1, \ldots, 4\}$ such that the 1C serial log corresponding to σ is equivalent to the execution ξ. In a merge of $\{\sigma_i : i = 1, \ldots, 4\}$, the relative positions of all the transactions in each σ_i are preserved. For each site S_i, only transactions belonging to S_i can update data objects in F_i. Therefore, for $i = 1, \ldots, 4$, the read-from relations in ξ among the transactions in σ_i are not altered in a merge of $\{\sigma_i : i = 1, \ldots, 4\}$. The remaining question to be addressed is the read-from relations between global transactions and local transactions at remote sites. For example, $(T_{12}, T_{23}) \in \mathrm{Gr}(\xi)$ implies that T_{23} should be serialized after T_{12} and before T_{13}. Since both (T_{14}, T_{33}) and (T_{43}, T_{33}) belong to $\mathrm{Gr}(\xi)$, T_{33} should be serialized after T_{14} and T_{43} and before T_{15} and T_{44}. Similarly, T_{17} should be serialized after T_{25}. Also, T_{45} should be serialized after T_{19}. It is not difficult to see that the serial order

$$T_{10}\, T_{20}\, T_{30}\, T_{40}\, T_{11}\, T_{12}\, T_{21}\, T_{22}\, T_{23}\, T_{13}\, T_{14}\, T_{41}\, T_{42}\, T_{43}\, T_{31}\, T_{32}\, T_{33}\, T_{24}\, T_{25}\, T_{15}\, T_{16}\, T_{17}\, T_{18}\, T_{19}\, T_{44}\, T_{45},$$

which is a merge of $\{\sigma_i : i = 1, \ldots, 4\}$, preserves all the read-from relations in ξ. Therefore, the fragmented execution ξ is serializable. □

Proof of Theorem 6.3.1.

Let $\mathrm{BO} = \{\sigma_i : \sigma_i = (T_{ij})_{j=0}^{n_i}, i = 1, \ldots, n\}$ be the set of broadcast orders of ξ and $\mathrm{Gr}(\xi)$ be the global-read relation in ξ. Let $GOS(\xi)$ be acyclic and $t(GOS(\xi))$ be a topological sort of $GOS(\xi)$. We assume without loss of generality that the first n transactions in $t(GOS(\xi))$ are the initial transactions $T_{10}, \ldots, T_{n0}$. We want to show that the sequences $\{\sigma_i : i = 1, \ldots, n\}$ can be merged to form a sequence $\sigma = (T_{h(i)})_{i=1}^{m}$, where $m = \sum_{i=1}^{n} n_i + n$, such that the 1C serial log $[T_{h(1)} \cdots T_{h(m)}]$ is equivalent

to ξ. The merging must be done without violating the order given by $t(GOS(\xi))$.

Initially, σ is an empty sequence. In the merging process, let M_i $(i = 1, \ldots, n)$ denote the last transaction in σ_i that has been merged into σ. Initially, M_i is undefined. After the initial transaction T_{i0} of σ_i is merged into σ, M_i is set to T_{i0}.

To construct σ, we first apply the following procedure iteratively to all the nodes in $t(GOS(\xi))$, starting with its first node. Let T_{ia} be the current node under scan in $t(GOS(\xi))$ and apply one of the following two operations to T_{ia}.

(1) If T_{ia} is an initial transaction, then let $\sigma = \sigma T_{ia}$, i.e., append T_{ia} to σ.

(2) If T_{ia} is a global transaction, then it has one incident precedence edge and one or more incident global-read edges. Apply (a) below to every incident global-read edge. Then apply (b) to the incident precedence edge.

(a) If there is a global-read edge from a node T_{kb} to T_{ia} $(k \neq i)$, and $(T_{kl}, T_{ia}) \in \mathrm{Gr}(\xi)$, then let $T_{kp} \cdots T_{kl}$ be the segment of transactions in σ_k between T_{kp} and T_{kl}, inclusive, where T_{kp} is the transaction following M_k in σ_k. Let $\sigma = \sigma T_{kp} \cdots T_{kl}$ and update M_k to T_{kl}. (Note that T_{kl} is unique; please see the definition of global-read relation defined earlier in this section.)

(b) If there is a precedence edge from T_{ib} to T_{ia}, then let $\sigma = \sigma T_{iq} \cdots T_{ia}$, where T_{iq} is the transaction following M_i in σ_i and $T_{iq} \cdots T_{ia}$ is the segment of transactions in σ_i between T_{iq} and T_{ia}, inclusive. Then update M_i to T_{ia}.

After the above procedure has completed, for every $i = 1, \ldots, n$, if M_i is not the last transaction in σ_i, then let $\sigma = \sigma T_{ik} \cdots T_{il}$, where T_{ik} is the transaction following M_i in σ_i, T_{il} is the last transaction in σ_i, and $T_{ik} \cdots T_{il}$ is the segment of transactions in σ_i between T_{ik} and T_{il}, inclusive. The construction of σ completes at this point.

It can be seen that σ is a merge of $\{\sigma_i : i = 1, \ldots, n\}$. Hence σ can be represented by a sequence $(T_{h(i)})_{i=1}^{m}$, where h is a permutation on the indices of the transactions and $m = \sum_{i=1}^{n} n_i + n$. Let L be the 1C log $[T_{h(1)} \cdots T_{h(m)}]$. In the above construction of σ, for each site S_i, the relative positions of all the transactions in σ_i are preserved in σ. Since only transactions belonging to S_i can modify the data objects in F_i, it follows that the read-from relations among transactions in σ_i are the same with respect to ξ and L.

Let us now examine the read-from relations across sites. If a global transaction T_{ia} reads X from a transaction T_{kb} belonging to another site S_k in ξ, then $T_{kb} \in \mathrm{RF}(T_{ia}, k)$ by definition. Let T_{kl} be the last transaction in σ_k that belongs to $\mathrm{RF}(T_{ia}, k)$. Then $(T_{kl}, T_{ia}) \in \mathrm{Gr}(\xi)$ and there is a global-read edge from T_{kp} to T_{ia}, where T_{kp} is either the global transaction preceding T_{kl} or T_{k0}. We will show that T_{ia} is ordered in σ after T_{kl} and before the transaction following T_{kl} in σ_k, if one exists. Besides the global-read edge, there must be a precedence edge from T_{im} to T_{ia} in $GOS(\xi)$, where T_{im} is either the global transaction preceding T_{ia} or T_{i0}. In the construction of σ, T_{kl} is appended to σ when the global-read edge from T_{kp} to T_{ia} is processed. After that, T_{ia} is appended to σ when the precedence edge from T_{im} to T_{ia} is processed. The transaction following T_{kl} in σ_k is appended to σ after T_{ia} has been processed. Therefore, T_{ia} is ordered in σ after T_{kl} and before the transaction following T_{kl} in σ_k. According to an observation made earlier in this section, when the global-read relation was defined, no transaction in σ_k after T_{kb} and before T_{kl} writes X; hence T_{ia} reads X from T_{kb} in L. On the other hand, suppose that T_{ia} reads X from T_{kb} in L. By the above argument, if T_{ia} reads X from T_{kh} in ξ, then T_{ia} reads X from T_{kh} in L. Hence T_{kb} and T_{kh} must be identical and T_{ia} reads X from T_{kb} in ξ. Therefore, the read-from relations between a global transaction and a transaction belonging to a remote site are the same with respect to ξ and L. Hence, ξ is equivalent to L and is serializable. □

6.4. An Algorithm to Control Fragmented Execution

The HFS's in a WADDS with the architecture proposed in Section 6.2 are timestamp-based schedulers. Therefore, an execution generated in a WADDS is always locally serializable at each site; the timestamps of the transactions involved provide a serialization order. The updates of the transactions completed at a site are broadcast in their timestamp order. Hence every execution generated in it is a fragmented execution. In this section, we present the core of a concurrency control algorithm for fragmented execution, called **Global Timestamp Order Certification (GTOC)**, which makes use of Theorem 6.3.1. It runs distributively in the GS's at all sites in a WADDS having the architecture given in Section 6.2, to certify the values read by a GLOBAL-READ. The sole function of GTOC is to ensure that, for any fragmented execution generated, its GOS graph is acyclic, and hence, it is globally serializable.

Let $S = \{S_i : i = 1, \ldots, n\}$ be the set of sites in a WADDS, and $\{F_i : i = 1, \ldots, n\}$ be their corresponding home fragments. In GTOC, we use two sets of timestamps, **local timestamps** and **global timestamps**. Suppose that a transaction T_{ij} is submitted to the TM at S_i. If T_{ij} is a local transaction, before any of its operations are executed, it is assigned a timestamp. This timestamp is called the **local timestamp** of T_{ij}. If T_{ij} is global, on the other hand, after its GLOBAL-READ operation is certified, it is assigned a timestamp, and this timestamp is used for processing the remaining operations in T_{ij}. This timestamp is also called the **local timestamp** of global transaction T_{ij}. We use $Lts(T_{ij})$ to denote the local timestamp of a transaction T_{ij}, whether it is local or global. Local timestamps are retrieved from local clocks. (In Section 6.2, the local timestamp of a transaction T_{ij} was referred to only as a timestamp. We refer to this timestamp in GTOC as the local timestamp, because we will later introduce global timestamps.)

As stated before, the operations submitted to the HFS of a site are scheduled by a timestamp-based scheduler at the HFS. As a matter of fact, the timestamps used by this scheduler are the local

timestamps of these transactions. Also, the updates of these transactions are broadcast and delivered in their local timestamp order. For each i, let σ_i be the sequence of committed transactions at S_i ordered by their local timestamps. Therefore σ_i is also the broadcast order at S_i.

We now introduce the timestamps for data objects, which are used for certification. If a copy X_i in the DB at a site S_i belongs to the home fragment F_i of S_i, then it has a read and a write timestamp. (Please refer to the description of the HFS in Section 6.2.) The write timestamp is the local timestamp of the last transaction that has updated X_i. This write timestamp is called the **timestamp** of X_i, denoted by $ts(X_i)$. If X_i is in a remote fragment F_k, then its value is received from a remote update of a transaction T_{kl} executed at a remote site S_k. In this case, X_i has just one timestamp called the **timestamp** of X_i, which is set to $Lts(T_{kl})$, and this timestamp, denoted by $ts(X_i)$, is attached to X_i in the DB of S_i. (Please refer to the description of the RFS in Section 6.2.)

Suppose that a global transaction T_{ik} is submitted at site S_i and its GLOBAL-READ operation has finished the reading of data object copies in remote fragments in the DB. (Refer to the execution of a global transaction described in Section 6.2.) The major problem we face is how to certify the values read by the GLOBAL-READ of T_{ik}. Recall that certification consists of testing the GOS graph of the execution of all committed transactions and T_{ik} for acyclicity. Unfortunately, testing the GOS distributively for acyclicity incurs too much communication overhead. Therefore we are forced to verify only a sufficient condition for acyclicity. For this purpose, we introduce **global timestamps** for global transactions. *We want to ensure that, if the GLOBAL-READ of T_{ik} is certified, then the edges introduced by T_{ik} in the GOS graph, consisting of T_{ik} and all the committed global transactions, are all directed from a transaction with a smaller global timestamp to another with a larger global timestamp. Hence, the global timestamps of the transactions give a topological sort of the graph.*

The above requirement on global timestamps may or may not be satisfied, depending on the global timestamp assigned to T_{ik}. Unfortunately, we need to assign a global timestamp to T_{ik} *before* the certification. If it is found that the above requirement is not satisfied, we consider the certification as failed even though the GOS graph may be acyclic. Thus we are testing only a sufficient condition for acyclicity. Let GS_i denote the global synchronizer GS at site S_i, $i = 1, \ldots, n$. When GS_i starts to certify the values read by the GLOBAL-READ of a global transaction T_{ik}, it assigns a **global timestamp** $Gts(T_{ik})$ to T_{ik}, *before* a local timestamp is assigned to T_{ik}. Global timestamps are retrieved from a system-wide unique **global clock**. Lamport's logical clock [Lam78] can be used for this purpose, and the site identity can be appended to make the clock values unique. In Lamport's logical clock, whenever a site S_i receives a message with a timestamp t from another site, if t is larger than the clock value at S_i, it is advanced to a value that is one tick larger than t.

Note that we cannot use the global timestamp of a global transaction T_{ik} as T_{ik}'s local timestamp. If this is done, while T_{ik} is waiting for the certification of its GLOBAL-READ, local transactions with timestamps larger than $Gts(T_{ik})$ cannot be executed until T_{ik} has completed, because these local transactions may have to read some values written by T_{ik}. This violates our policy of assigning higher priority to local transactions.

After a global timestamp $Gts(T_{ik})$ is assigned to T_{ik}, GS_i generates a set of **certification requests**, $\{CR_j(T_{ik}) : F_j \in \rho_{ik}\}$, one for each remote fragment F_j in ρ_{ik}, where ρ_{ik} is the "readset" of T_{ik} in terms of fragments, i.e., $\rho_{ik} = \{F_j$: the GLOBAL-READ of T_{ik} reads some data objects from $F_j\}$. For each $F_j \in \rho_{ik}$, GS_i can determine $t_j(T_{ik}) = max\,\{ts(X_i) : X_i$ is a copy of $X \in F_j$ at S_i read by the GLOBAL-READ of $T_{ik}\}$. Clearly, there exists a transaction T_{jl} at site S_j such that $Lts(T_{jl}) = t_j(T_{ik})$. T_{jl} is in fact the last transaction in σ_j whose updates were read by the GLOBAL-READ of T_{ik}. $CR_j(T_{ik})$ contains two timestamps, $Gts(T_{ik})$ and $t_j(T_{ik})$, which are called the **global timestamp** and the **data timestamp** of the certification request $CR_j(T_{ik})$, respectively. For

convenience, these two timestamps are denoted by $CR_j(T_{ik}).Gts$ and $CR_j(T_{ik}).Dts$, respectively. Certification request $CR_j(T_{ik})$ is sent to the corresponding global synchronizer GS_j. As explained below in more detail, GS_j compares the transaction and data timestamps of $CR_j(T_{ik})$ with the timestamps of the global transactions that have already committed at S_j.

As noted above, there exists a transaction T_{jl} in σ_j such that $Lts(T_{jl}) = t_j(T_{ik}) = CR_j(T_{ik}).Dts$. Suppose T_{ja} and T_{jb} are consecutive global transactions in σ_j such that

$$Gts(T_{ja}) < CR_j(T_{ik}).Gts < Gts(T_{jb}). \quad (6.4.1)$$

If

$$Lts(T_{ja}) \le CR_j(T_{ik}).Dts < Lts(T_{jb}), \quad (6.4.2)$$

then the certification request $CR_j(T_{ij})$ can be granted by GS_j *as far as all the committed transactions at S_j are concerned.* The reason for this is as follows. Since $CR_j(T_{ik}).Dts = Lts(T_{jl})$, condition (6.4.2) implies that T_{ja} and T_{jb} are the global transactions preceding and following T_{jl} in σ_j, respectively. Hence, a global-read edge from T_{ja} to T_{ik}, and an induced edge from T_{ik} to T_{jb} are introduced in the GOS graph consisting of T_{ik} and all committed transactions. According to (6.4.1), these three edges are all directed from a node with a smaller global timestamp to a node with a larger global timestamp. If this is true at each site S_j that receives certification request $CR_j(T_{ik})$, then the GOS graph consisting of T_{ik} and all the committed global transactions can be topologically sorted by the global timestamps of the transactions.

According to the above discussion, in certifying $CR_j(T_{ik})$, GS_j has to identify T_{ja} and T_{jb} in condition (6.4.1) and compare their local timestamps with $CR_j(T_{ik}).Dts$. If condition (6.4.2) is satisfied, $CR_j(T_{ik})$ should be certified by GS_j as far as the committed transactions are concerned. Besides considering committed transactions at S_j, GS_j also has to compare the timestamps of $CR_j(T_{ik})$ with that of some waiting global transactions; this will be discussed later in this section.

Before doing so, we give an example to illustrate what we have just discussed and an additional requirement of certification.

Example 6.4.1. Consider three sites S_1, S_2, and S_3 in a WADDS, and suppose that a global transaction T_{22} is submitted at S_2. Let σ_1 be the sequence of all committed transactions at S_1, ordered by their local timestamps. Suppose that T_{12} and T_{15} are two global transactions committed at S_1 such that $Lts(T_{12}) = 2$, $Gts(T_{12}) = 1$, $Lts(T_{15}) = 5$, and $Gts(T_{15}) = 4$, and that there is no global transaction between T_{12} and T_{15} in σ_1. If the GLOBAL-READ of T_{22} has read some data objects belonging to S_1 such that $CR_1(T_{22}).Dts = t_1$, then there exists a transaction T_{1k} $(1 < k < 5)$ belonging to S_1 such that $Lts(T_{1k}) = t_1$ and T_{22} has read a data object from T_{1k}. When $CR_1(T_{22})$ arrives at GS_1, it is found that (6.4.1) is satisfied, i.e., $Gts(T_{12}) < CR_1(T_{22}).Gts = Gts(T_{22}) < Gts(T_{15})$. If (6.4.2) holds, i.e., $Lts(T_{12}) \leq t_1 < Lts(T_{15})$, then T_{12} and T_{15} are the global transactions preceding and following T_{1k}, respectively, and the GOS graph involving T_{12}, T_{15} and T_{22} has a precedence edge from T_{12} to T_{15}, a global-read edge from T_{12} to T_{22} and an induced edge from T_{22} to T_{15}. This GOS graph is shown in Figure 6.4.1. Since $Gts(T_{12}) < Gts(T_{22}) < Gts(T_{15})$, these three edges are all directed from a node with a smaller global timestamp to a node with a larger global timestamp. Therefore the certification request $CR_1(T_{22})$ should be granted if $Lts(T_{12}) \leq CR_1(T_{22}).Dts < Lts(T_{15})$.

Now suppose that T_{22} has committed, and a new global transaction T_{34} is submitted at S_3 whose GLOBAL-READ operation reads some data objects belonging to S_1 such that $CR_1(T_{34}).Dts = t_2$. When $CR_1(T_{34})$ arrives at S_1, it is found that $Gts(T_{22}) < CR_1(T_{34}).Gts = Gts(T_{34}) < Gts(T_{15})$ at GS_1. If $CR_1(T_{22}).Dts = t_1 < t_2 < Lts(T_{15})$ holds, then T_{22} and T_{34} have the same global transaction T_{12} preceding them. The GOS graph involving T_{12}, T_{15}, T_{22}, and T_{34} is shown in Figure 6.4.2. There is a global-read edge from T_{12} to T_{34} and an induced edge from T_{34} to T_{15}. Besides these two edges, there is an additional induced edge from T_{22} to T_{34}. (Recall the definition of edge set (4) of a GOS graph in Section 6.3.) Since $Gts(T_{12}) < Gts(T_{22}) < Gts(T_{34}) < Gts(T_{15})$, the four

edges mentioned above are all directed from a node with a smaller global timestamp to a node with a larger global timestamp. Therefore, $CR_1(T_{34})$ should be certified if $CR_1(T_{22}).Dts < t_2 < Lts(T_{15})$. □

In the above example, GS_1 could certify $CR_1(T_{22})$ using the conditions (6.4.1) and (6.4.2). However, to certify $CR_1(T_{34})$, GS_1 had to compare the timestamps of $CR_1(T_{34})$ not only with those of the global transactions already committed at S_1 but also with that of T_{22} which has sent a certification request to GS_1 and has committed at another site S_2. We wish to process both $CR_1(T_{22})$ and $CR_1(T_{34})$ uniformly.

It follows from the above observation that there are two sets of committed global transactions to consider when GS_j is certifying $CR_j(T_{ik})$. The first set consists of the committed global transactions belonging to S_j, such as T_{12} and T_{15} at GS_1 in the above example. The second set consists of the committed global transactions belonging to sites other than S_j, which have sent certification requests to GS_j, such as T_{22} when T_{34} arrives at GS_1 in the above example. *In order to treat all certification requests (e.g., $CR_1(T_{22})$ and $CR_1(T_{34})$) uniformly, we order the two sets of global transactions together by their global timestamps at S_j, and replace T_{ja} and T_{jb} in (6.4.1) and (6.4.2) by T_{ca} and T_{db}, respectively, where T_{ca} (T_{db}) is defined as the global transaction with the largest (smallest) global timestamp less (larger) than $CR_j(T_{ik}).Gts$. T_{ca} and T_{db} may or may not belong to S_j.* Hence (6.4.1) is modified to

$$Gts(T_{ca}) < CR_j(T_{ik}).Gts < Gts(T_{db}), \tag{6.4.3}$$

and (6.4.2) to

$$LDts_j(T_{ca}) \leq CR_j(T_{ik}).Dts \leq LDts_j(T_{db}), \tag{6.4.4}$$

where $LDts_j(T_{xy}) = Lts(T_{xy})$, if T_{xy} belongs to S_j; otherwise, $LDts_j(T_{xy}) = CRj(T_{xy}).Dts$, for $xy = ca, db$. The second inequality in (6.4.2), 'strictly less than', has been replaced by 'less than or equal to' in (6.4.4), because $CR_j(T_{ik}).Dts$ and $LDts_j(T_{db})$ may be equal if T_{db} doesn't belong to S_j. If T_{db} belongs

to S_j, i.e., if $d = j$, then $CR_j(T_{ik}).Dts \neq LDts_j(T_{db})$; if $CR_j(T_{ik}).Dts = LDts_j(T_{db})$, T_{ik} would have read from T_{db} and we would have $Gts(T_{ik}) > Gts(T_{db})$ and (6.4.3) would not hold. Hence testing '≤' in the second inequality of (6.4.4) is equivalent to testing '<' in this case.

To facilitate the comparisons in (6.4.3) and (6.4.4), at each site S_j, GS_j makes use of a data structure $COMMIT_j$, which is a list of records. Each record in $COMMIT_j$ represents a transaction T in one of the above two sets of committed global transactions and has two fields Gts and lts. Gts is the global timestamp of transaction T. If T is a global transaction in the first set, $lts = Lts(T)$. If T is a global transaction in the second set, the home site of T must have sent a certification request $CR_j(T)$ to GS_j, and $lts = CR_j(T).Dts$. The records in $COMMIT_j$ are sorted in the increasing order of their values in the Gts field.

If there are p fragments in the system, let $t_1, \ldots, t_p$ be the p smallest global timestamps. For any site S_i, $COMMIT_i$ is initialized to a list containing only one record corresponding to the initial transaction T_{i0}. This record has $Gts = t_i$ and $lts = 0$.

Suppose the certification request $CR_j(T_{ik})$ of a global transaction T_{ik} is received by GS_j from S_i. GS_j checks (6.4.3) and (6.4.4) as follows. Since $COMMIT_j$ is sorted by the Gts field, it is easy to determine T_{ca} and T_{db}. An interval $[t_1, t_2]$, where t_1 (t_2) is the lts value of the record representing T_{ca} (T_{db}), is called the **safe interval** for $CR_j(T_{ik})$ with respect to $COMMIT_j$. If T_{db} doesn't exist, then $t_2 = \infty$. If $CR_j(T_{ik}).Dts \in [t_1, t_2]$, the certification request $CR_j(T_{ik})$ is said to have passed the **acceptance test against** $COMMIT_j$, and $CR_j(T_{ik})$ can be granted as far as all the transactions in $COMMIT_j$ are concerned. If T_{ik} is eventually committed at S_i, then a record representing it will be inserted into $COMMIT_j$, with fields $Gts = Gts(T_{ik})$ and $lts = CR_j(T_{ik}).Dts$. It is clear from the construction of $COMMIT_j$ and the acceptance test that the Gts values are monotonically increasing, while the lts values are monotonically non-decreasing.

The acceptance test against $COMMIT_j$ given above is justified by the following lemma.

Lemma 6.4.1. *Let G be the (acyclic) GOS graph representing all committed global transactions such that the global timestamps of the transactions correspond to a topological sort of G. Let T_{ik} be a new global transaction whose certification requests are to be tested and $\rho_{ik} = \{F_j$: the GLOBAL-READ of T_{ik} reads some data objects from $F_j\}$. Consider the GOS graph G' which is obtained from G by adding T_{ik} and the edges associated with it to G. The global timestamps of the transactions correspond to a topological sort of G' if both (6.4.3) and (6.4.4) hold for T_{ik} with respect to all site S_j, $F_j \in \rho_{ik}$.*

Proof. Consider any of the new edge $e = (u, v)$ introduced into G to form G', e can be either a global-read edge or an induced edge. We want to show that, if (6.4.3) and (6.4.4) hold for T_{ik}, then the global timestamp of u is larger than that of v. There are three cases to consider. Let T_{ca} and T_{db} be as defined in (6.4.3) and (6.4.4) when $CR_j(T_{ik})$ is tested against $COMMIT_j$, $F_j \in \rho_{ik}$. Note that it is always true that $LDts_j(u) \le LDts_j(v)$ for any $(u, v) \in G'$.

(1) e is a global-read edge (T_{je}, T_{ik}).

T_{ik} has read a copy of a data object belonging to F_j. Let X_j be such a copy with the largest timestamp. When the value of X_j was broadcast from S_j, S_i should have received a global timestamp at least as large as $Gts(T_{je})$. Hence $Gts(T_{je}) < Gts(T_{ik})$.

(2) e is an induced edge (T_{ik}, T_{jh}) belonging to edge set (3) in the definition of GOS graph.

Since e is an induced edge, $LDts_j(T_{ik}) \ne Lts(T_{jh})$. As commented earlier, $LDts_j(T_{ik}) \le Lts(T_{jh})$; therefore $LDts_j(T_{ik}) < Lts(T_{jh})$. By definition of T_{ca} and T_{db}, it is not possible that $Gts(T_{ca}) < Gts(T_{jh}) < Gts(T_{db})$. If $Gts(T_{jh}) \le Gts(T_{ca})$, it follows from the monotonicity of the records in $COMMIT_j$ that $LDts_j(T_{ik}) < Lts(T_{jh}) \le LDts_j(T_{ca})$, which contradicts (6.4.4). Hence $Gts(T_{jh}) \ge Gts(T_{db}) > Gts(T_{ik})$.

(3) e is an induced edge (T_{ik}, T_{hl}) belonging to edge set (4), where $h \neq i, j$.

It is not possible that $Gts(T_{ca}) < Gts(T_{hl}) < Gts(T_{db})$. According to the definition of edge set (4), $LDts_j(T_{ik}) < LDts_j(T_{hl})$. If $Gts(T_{hl}) \leq Gts(T_{ca})$, then $LDts_j(T_{ik}) < LDts_j(T_{hl}) \leq LDts_j(T_{ca})$, which contradicts (6.4.4). Hence $Gts(T_{hl}) \geq Gts(T_{db}) > Gts(T_{ik})$. □

After the certification request $CR_j(T_{ik})$ passes the acceptance test against $COMMIT_j$, GS_j has to test it against some waiting global transactions. There are two sets of waiting transactions to be considered at S_j. The first set consists of those belonging to S_j that are waiting for the GS's at remote sites to grant their certification requests. If T_{jl} is such a transaction, its local timestamp $Lts(T_{jl})$ will not be assigned until its GLOBAL-READ is certified. Hence $Lts(T_{jl})$ must be larger than $CR_j(T_{ik}).Dts$. Therefore, $Gts(T_{jl})$ must also be larger than $Gts(T_{ik})$ in order that $CR_j(T_{ik})$ can be granted by GS_j. The second set consists of the global transactions waiting at sites other than S_j whose certification requests sent to GS_j have been granted. Suppose T_{hl} is such a transaction and its request $CR_j(T_{hl})$ has been granted by GS_j. If $CR_j(T_{ik})$ arrives at GS_j after $CR_j(T_{hl})$ has been granted, it cannot ignore the fact that T_{hl} may be committed and join $COMMIT_j$ later. Therefore, T_{ik} must be tested against these two sets of waiting transactions in the same way as it was tested against the committed transactions in $COMMIT_j$. If T_{ik} passes this second test, then even if all the waiting transactions are committed later and are added to the list $COMMIT_j$, T_{ik} will still pass the acceptance test against this extended $COMMIT_j$. If T_{ik} fails this test, the waiting transaction T that caused the failure is identified and $CR_j(T_{ik})$ will be made waiting at GS_j until either T is committed or aborted. If T is committed, $CR_j(T_{ik})$ is doomed to be rejected; otherwise, it is submitted again to GS_j.

In order to keep track of the waiting transactions, GS_j maintains a list $WAIT_j$, which contains the records representing the waiting transactions mentioned above. Each record in $WAIT_j$ has three fields Gts, lts and $retry$. Gts is the global timestamp of a waiting transaction T that the record represents. If T belongs to the first set of waiting transactions, then the lts field is undefined. If T

belongs to the second set, then $CR_j(T)$ has been granted by GS_j and $lts = CR_j(T).Dts$. The field *retry* is a pointer to a **retry list** of records representing the global transactions which have been rejected because of T. As with $COMMIT_j$, the records in $WAIT_j$ are sorted in the increasing order of their values in the *Gts* field. Initially, $WAIT_j$ contains only one record W_{j0} such that $Gts = t_j$, (t_j is the timestamp used to initialize $COMMIT_j$,) $lts = 0$ and *retry* = the null pointer. If we ignore the retry field, each record in $WAIT_j$ is just like a record in $COMMIT_j$. In fact, we could add all records in $WAIT_j$ to $COMMIT_j$ and apply the test based on (6.4.3) and (6.4.4). However, since there is a possibility that some transactions represented in $WAIT_j$ may not be committed, we maintain a separate list $WAIT_j$.

$CR_j(T_{ik})$ is tested against $WAIT_j$ by GS_j in the following way. Firstly, we determine a **safe interval** $[w_1, w_2]$ **against** $WAIT_j$. We compute $[w_1, w_2]$ as we computed the safe interval of $CR_j(T_{ik})$ against $COMMIT_j$, (list $COMMIT_j$ should be replaced by $WAIT_j$). If $CR_j(T_{ik}).Dts \in [w_1, w_2]$, $CR_j(T_{ik})$ passes the **acceptance test against** $WAIT_j$. Note that $w_1 = LDts_j(T_{ca})$ and $w_2 = LDts_j(T_{db})$, where T_{ca} and T_{db} are defined in (6.4.3) when $CR_j(T_{ik})$ is tested against $WAIT_j$. For the convenience of future discussion, we say that T_{ca} (T_{db}) is the transaction associated with w_1 (w_2) in the acceptance test against $WAIT_j$. If $CR_j(T_{ik})$ passes the acceptance test against $WAIT_j$, then it is granted by GS_j and a reply message is sent back to GS_i. If $CR_j(T_{ik})$ does not pass the acceptance test against $WAIT_j$, then either $t_1 \le CR_j(T_{ik}).Dts < w_1$ or $w_2 < CR_j(T_{ik}).Dts \le t_2$, since we assume $CR_j(T_{ik})$ has already passed the acceptance test against $COMMIT_j$. In the first case, the acceptance test fails because of the transaction T_{ca}, and $CR_j(T_{ik})$ is put to wait in list *retry* of the record representing T_{ca} in $WAIT_j$. In the second case, the acceptance test fails because of the transaction T_{db} and $CR_j(T_{ik})$ is not made to wait but simply rejected. The reason for rejecting $CR_j(T_{ik})$ in the latter case is to avoid a possible deadlock among waiting transactions. The policy used is to allow only a transaction with a larger global timestamp to wait for another one with a smaller global timestamp. (An example of deadlock

will be shown in Example 6.4.2 below, if T_{ik} is allowed to wait for the transaction associated with T_{db}.) If $CR_j(T_{ik})$ is granted by GS_j, a record is inserted into $WAIT_j$ to represent T_{ik}, of which $Gts = CR_j(T_{ik}).Gts$, $lts = CR_j(T_{ik}).Dts$, and *retry* is the null pointer.

The GLOBAL-READ of transaction T_{ik} is granted by GS_i if all certification requests $\{CR_j(T_{ik}) : F_j \in \rho_{ik}\}$ are granted; otherwise, T_{ik} is aborted. If T_{ik} is committed at S_i eventually, then a record associated with it is inserted in $COMMIT_i$ and $COMMIT_j$, for each site S_j such that $F_j \in \rho_{ik}$. Also, the records associated with T_{ik} in $WAIT_i$ and $WAIT_j$, for each site S_j mentioned above, are deleted.

Example 6.4.2. Consider a WADDS consisting of four sites S_1, S_2, S_3 and S_4, in which two global transactions T_{3a} and T_{4b} are waiting for GTOC to certify their certification requests. Suppose that T_{3a} submitted at S_3 has sent two certification requests $CR_1(T_{3a})$ and $CR_2(T_{3a})$ to S_1 and S_2, respectively, and that T_{4b} submitted at S_4 has sent two certification requests $CR_1(T_{4b})$ and $CR_2(T_{4b})$ to S_1 and S_2, respectively. $CR_1(T_{3a})$ now arrives at S_1 before $CR_1(T_{4b})$ and has been granted by GS_1. $WAIT_1$ thus contains only two records associated with T_{3a} and the initial transaction T_{10}, when $CR_1(T_{4b})$ arrives at GS_1. Let $CR_1(T_{3a}).Dts = w_1$ and $CR_1(T_{4b}).Dts = t_4$. When $CR_1(T_{4b})$ arrives at GS_1, suppose it passes the acceptance test against $COMMIT_1$ and the safe interval is $[t_1, t_2]$ such that $t_1 < t_4 < w_1 < t_2$. If $Gts(T_{3a}) < Gts(T_{4b})$, then the safe interval of T_{4b} against $WAIT_1$ is $[w_1, \infty]$. This is illustrated in Figure 6.4.3(a). Since $t_4 \notin [w_1, \infty]$, the acceptance test of T_{4b} against $WAIT_1$ is negative. Since $t_4 \in [t_1, w_1]$, the failure of T_{4b} in the acceptance test is caused by T_{3a}; hence T_{4b} is put to wait in the retry list associated with T_{3a}.

Next, suppose that $CR_2(T_{4b})$ arrives at S_2 before $CR_2(T_{3a})$ and has been granted by GS_2. $WAIT_2$ thus contains only two records associated with T_{4b} and the initial transaction T_{20}, when $CR_2(T_{3a})$ arrives at GS_2. Let $CR_2(T_{4b}).Dts = w'_2$ and $CR_2(T_{3a}).Dts = t_3$. When $CR_2(T_{3a})$ arrives at GS_2, suppose it passes the acceptance test against $COMMIT_2$ and the safe interval is $[t'_1, t'_2]$ such that

$t'_1 < w'_2 < t_3 < t'_2$. If $Gts(T_{3a}) < Gts(T_{4b})$, the safe interval of T_{3a} against $WAIT_2$ is $[0, w'_2]$. This is illustrated in Figure 6.4.3(b). Since $t_3 \notin [0, w'_2]$, the acceptance test of T_{3a} against $WAIT_2$ is negative. Since $t_3 \in [w'_2, t'_2]$, the failure of T_{3a} in the acceptance test is caused by T_{4b}. If we allow T_{3a} to wait for T_{4b}, then a deadlock occurs. This is the reason that this waiting is not allowed in the algorithm GTOC. □

In the following, the Global Timestamp Order Certification algorithm is formally presented in four phases.

Algorithm GTOC

Input : A certification request by global transaction T_{ik}.

Phase One [Initialization] :

(a) GS_i assigns a global timestamp $Gts(T_{ik})$ to T_{ik}.

(b) $\{CR_j(T_{ik}) : F_j \in \rho_{ik}\}$ is generated, where $\rho_{ik} = \{F_j :$ the GLOBAL-READ of T_{ik} reads some data objects from $F_j\}$. Each $CR_j(T_{ik})$ is sent by S_i to the corresponding GS_j.

(c) A record W is appended to $WAIT_i$, where $W.Gts = Gts(T_{ik})$, $W.lts$ is undefined, and $W.retry$ is the null pointer.

Phase Two [Acceptance Test against committed transactions] :

(a) When $CR_j(T_{ik})$ arrives at S_j, an acceptance test of $CR_j(T_{ik})$ against $COMMIT_j$ is performed by GS_j.

(b) If the acceptance test is negative, $CR_j(T_{ik})$ is rejected and a reply message $reject(CR_j(T_{ik}))$ is sent back to GS_i. Otherwise, the algorithm proceeds to its third phase

at GS_j.

Phase Three [Acceptance Test against waiting transactions] :

(a) An acceptance test on $CR_j(T_{ik})$ against $WAIT_j$ is performed by GS_j.

(b) If the acceptance test is positive, a reply message $accept(CR_j(T_{ik}))$ is sent back to GS_i and a record W is inserted between the records associated with w_1 and w_2 in $WAIT_j$, where w_1, w_2 are as defined in the acceptance test against $WAIT_j$ and $W.Gts = Gts(T_{ik})$, $W.lts = CR_j(T_{ik}).Dts$ and $W.retry$ is set to the null pointer.

(c) If the acceptance test is negative, let R be a record with two fields such that $R.Gts = Gts(T_{ik})$ and $R.lts = CR_j(T_{ik}).Dts$. If $t_1 \le CR_j(T_{ik}).Dts < w_1$, appends R to the *retry* list of the record associated with w_1, where t_1 is as defined in the acceptance test against $WAIT_j$. Otherwise, send a reply message $reject(CR_j(T_{ik}))$ to GS_i.

Phase Four [Termination] :

(a) If GS_i receives a $reject(CR_a(T_{ik}))$ from a site S_a, then abort T_{ik} and delete the records representing T_{ik} in $WAIT_i$ and $WAIT_j$, for all $F_j \in \rho_{ik}$. The transaction in the retry list $W.retry$ is resubmitted to GS_i, where W is the record representing T_{ik} in $WAIT_i$. For each $F_j \in \rho_{ik}$, GS_j retests the transactions in the retry lists $W'.retry$, where W' is the record representing T_{ik} in $WAIT_j$.

(b) If GS_i receives an $accept(CR_j(T_{ik}))$ from every site S_j, $F_j \in \rho_{ik}$, it delays T_{ik} until no other transaction with a smaller global timestamp belonging to S_i is in $WAIT_i$, namely, until all these waiting transactions have completed their GLOBAL-READ operations. (This waiting ensures that all the precedence edges between global transactions executed at S_i are directed from one with a smaller (older) global timestamp to another one with a

larger (newer) global timestamp.)

(c) Once the GLOBAL-READ operation of T_{ik} is completed, a message is sent to every site S_j ($F_j \in \rho_{ik}$) to delete the record representing T_{ik} in $WAIT_j$ and to insert into $COMMIT_j$ a record representing T_{ik}. At the home site S_i of T_{ik}, delete the record representing T_{ik} from $WAIT_i$ and insert a new record representing T_{ik} into $COMMIT_i$. □

Let us examine if the possibility of a deadlock among waiting transactions. In Phase Three (c), we allow T_{ik} to wait only for the transaction associated with w_1, but not the one associated with w_2. This ensures that T_{ik} may wait only for a transaction with a smaller global timestamp. Also in Phase Four (b), a transaction waiting to complete a GLOBAL-READ operation may wait only for transactions with smaller global timestamps. Therefore, deadlock is not possible in GTOC.

Theorem 6.4.1. *A fragmented execution generated by GTOC in a WADDS is globally serializable.*

Proof. Suppose ξ is a fragmented execution generated by GTOC over a set of sites $\{S_i : i = 1, \ldots, n\}$. Let **T** be the set of transactions involved in ξ. Since a transaction submitted is eventually either committed or aborted, **T** contains only committed global transactions. Let σ_i, $i = 1, \ldots, n$, be the sequence of transactions belonging to S_i ordered by their local timestamps. We want to show that the edges in $GOS(\xi)$ are all directed from a global transaction with a smaller global timestamp to another one with a larger global timestamp, and thus $GOS(\xi)$ is acyclic.

Let $T_{ik} \in \mathbf{T}$. All the certification requests of T_{ik} must have been granted before it commits. It follows from Lemma 6.4.1 that the new edges introduced in $GOS(\xi)$ by T_{ik} are all directed from a global transaction with a smaller global timestamp to another with a larger global timestamp. Hence all the global-read edges and induced edges in $GOS(\xi)$ satisfy the required property.

Step (b) of Phase Four of GTOC ensures that the global transaction preceding T_{ik} in σ_i has a global timestamp smaller than that of T_{ik}. Hence the precedence edge introduced by T_{ik} also satisfies the required property. Hence all edges in $GOS(\xi)$ are directed from a transaction with a smaller global timestamp to another with a larger global timestamp and $GOS(\xi)$ is acyclic. Therefore ξ is globally serializable. □

In the following, we will discuss some problems related to GTOC. The first problem with GTOC is the amount of storage occupied by $COMMIT_i$ and $WAIT_i$ at each site S_i. Records in $WAIT_i$ are deleted after the corresponding transactions are committed. Therefore, the space occupied by $WAIT_i$ is not a serious problem. What we need is a garbage collection mechanism for the records in the $COMMIT_i$. This can be done by finding out the smallest global timestamp t_s among all the waiting transactions. Periodically, every site S_i finds the smallest global timestamp among all the transactions in $WAIT_i$. These global timestamps are then broadcast to all the other sites. The smallest global timestamp t_s is then computed. At each site S_i, let C be the record in $COMMIT_i$ with the largest $C.Gts < t_s$. After C has been identified, the global timestamps of all certification requests arriving at S_i must be larger than $C.Gts$. Therefore, all the records C_k in $COMMIT_i$ such that $C_k.Gts < C.Gts$ are no longer needed for acceptance test. All these records can be deleted from $COMMIT_i$.

The second problem concerns aborted global transactions. Suppose a certification request $CR_j(T_{ik})$ of T_{ik} is rejected by GS_j at site S_j because $Gts(T_{ik})$ is too large. Suppose T_{ik} is resubmitted after being assigned a larger global timestamp. Let $RS(T_{ik})$ be the readset of T_{ik}. If no data object in $RS(T_{ik}) \cap F_j$ is updated since T_{ik} last read it, then $CR_j(T_{ik}).Dts$ remains unchanged and $CR_j(T_{ik})$ is rejected again by GS_j. In order to avoid this, GS_i can send the name of a data object $X \in RS(T_{ik}) \cap F_j$ to S_j when it sends $CR_j(T_{ik})$. If S_j finds that $CR_j(T_{ik})$ is rejected by GS_j because $Gts(T_{ik})$ is too large, then it can broadcast X with a timestamp larger than the current time at S_j. In

this way, $CR_j(T_{ik}).Dts$ will be increased every time T_{ik} is submitted again.

If global transactions are rare, then the probability that a global transaction is aborted because its GLOBAL-READ is rejected by a GS is small. However, it cannot be guaranteed that a global transaction won't be aborted. In the worst case, the GLOBAL-READ of a global transaction may be rejected every time the global transaction is submitted or resubmitted. In order to remedy this "starvation", a site S_i can change the status of a global transaction to **urgent**. We want to ensure that no certification request of an urgent global transaction will be rejected by the GS at a remote site. To this end, site S_i sends out a **timestamp query message** $q(T_{ik})$ which contains the name of a data object $X \in RS(T_{ik}) \cap F_j$ to each site S_j with $F_j \in \rho_{ik}$ on behalf of an urgent global transaction. When a site S_j receives $q(T_{ik})$, GS_j stops processing certification requests. After all waiting transactions in $WAIT_j$ have committed or aborted, GS_j sends to S_i the largest global timestamp c_j among all the transactions in $COMMIT_j$ and broadcasts X with a new timestamp larger than the current time at S_j. After receiving all timestamps $\{c_j : F_j \in \rho_{ik}\}$, T_{ik} executes it GLOBAL-READ and sends out certification requests with a global timestamp larger than any timestamp in $\{c_j : F_j \in \rho_{ik}\}$. For each $F_j \in \rho_{ik}$, let C_j be the record in $COMMIT_j$ such that $C_j.Gts = c_j$. Since $CR_j(T_{ik}).Gts = Gts(T_{ik}) > c_j = C_j.Gts$, the safe interval for T_{ik} against $COMMIT_j$ is $[C_j.lts, \infty]$. As mentioned above, S_j broadcasts a data object in $RS(T_{ik}) \cap F_j$ with a timestamp larger than $C_j.lts$ after it has received $q(T_{ik})$. Therefore, $CR_j(T_{ik}).Dts > C_j.lts$. Hence $CR_j(T_{ik})$ passes the acceptance test against $COMMIT_j$. $CR_j(T_{ik})$ also passes the acceptance test against $WAIT_j$, because $WAIT_j$ contains only one record associated with the initial transaction. Since this is true for all S_j with $F_j \in \rho_{ik}$, the GLOBAL-READ of T_{ik} won't be rejected. This is achieved by sacrificing the response time of other global transactions. Since we don't anticipate a frequent occurrence of urgent global transactions, this strategy is acceptable. A site can issue only one urgent global transaction at a time. Also, a site can reply positively to only one timestamp query message at a time; timestamp query

messages arriving later have to wait until the urgent global transaction associated with the first timestamp query message has completed. In order to avoid deadlock, we allow only the timestamp query message of a urgent global transaction with a larger timestamp to wait for another with a smaller global timestamp.

6.5. Performance Analysis

Compared with other conventional schemes for concurrency control, GTOC strongly favors local transactions. In the following, we compare its performance with the primary copy two-phase locking scheme (PC, for short). (See Section 3.2 for an explanation of the primary copy locking scheme.) In a fragmented database, it is natural to regard the copy of a data object at its home site as the primary copy of the data object.

Let t_G denote the execution time for a local transaction under GTOC. Note that t_G equals local processing time L_G at its home site. If PC is used, the execution time t_P of a local transaction consists of two parts, L_P and t_c, where L_P is local processing time and t_c is the time spent for communication. The communication time t_c in turn consists of the time to deliver the updates and to run a commit protocol. Here, we assume that the most primitive 2-phase commit protocol is used. Therefore, two round trips of communication are needed. In the first round, the home site of the local transaction sends out update messages to all other sites and acknowledgements of receipt are returned to the home site. In the second round, commit messages are sent to all these sites to instruct them to commit the updates, and confirmations are returned to the home site so that the locks on the updated data objects can be released. We thus have $t_c = 4\alpha_a$, where α_a is the average time for sending a message across the network.

Subtracting t_G from t_P, we get

$$t_P - t_G = (L_P - L_G) + 4\alpha_a. \tag{6.5.1}$$

In a wide-area distributed database (WADDS), α_a may be as large as a few seconds. In general, it is much larger than the difference $(L_P - L_G)$. This is particularly true for short transactions which can be processed in microseconds. Hence, in this kind of environment, GTOC performs much better than PC with regard to the execution time of a local transaction.

Let us consider the average execution time over all transactions, assuming that a fraction r of all transactions are global transactions. Let a_G and a_P be, respectively, the average execution times of a transaction under GTOC and PC.

When a global transaction T_{ik} is executed under GTOC, its execution time consists of two parts. The first part is G_G which includes the processing time of T_{ik} at its home site and the processing time of its certification requests at remote sites. G_G also includes the waiting time if a certification request of T_{ik} has to wait in a retry list at a remote site. The second part is $2\alpha_a$ which is the communication time for sending out certification requests and receiving replys from remote sites. Hence

$$a_G = (1-r)L_G + r(G_G + 2\alpha_a),$$

where L_G is the processing time of a local transaction under GTOC as defined above.

Under PC, a global transaction T_{ik} spends $2\alpha_a$ units of time to remotely lock the primary copies of the data objects in its readset which belong to other sites. It spends $2\alpha_a$ units of time to send out updates and receive replies. Lastly, it has to send out commit messages and wait for reply messages to release locks. Therefore, the execution time of T_{ik} under PC is $G_P + 6\alpha_a$, where G_P is the local processing time of T_{ik} at its home site. As analysed above, the execution time of a local transaction under PC is $(L_P + 4\alpha_a)$. Hence

$$a_P = (1-r)(L_P + 4\alpha_a) + r(G_P + 6\alpha_a), \text{ and}$$

$$a_P - a_G = (1-r)(L_P - L_G) + r(G_P - G_G) + 4\alpha_a.$$

As discussed above, the difference $(L_P - L_G)$ can be ignored when compared with the term $4\alpha_a$. Therefore,

$$a_P - a_G = r(G_P - G_G) + 4\alpha_a. \tag{6.5.2}$$

In general, G_G might be larger than G_P. However, if r is sufficiently small, the difference $(a_P - a_G)$ is dominated by the term $4\alpha_a$. Therefore, in case r is small, GTOC performs better than PC in terms of average execution time of a transaction. In fact, the term $2\alpha_a$ for sending out updates of a local transaction cannot be avoided in any conventional concurrency control. Therefore, GTOC is definitely better than such schemes if fast response to local transaction is crucial. As for the average execution time, it depends very much on the fraction r of global transactions among all transactions. Still, if r is small, GTOC is preferable to other conventional schemes.

6.6. Partition Failures in a Fragmented Database System

The technique used in Chapter 4 for deriving an upper bound on availability can be applied to fragmented databases. We will show that the upper bound is achievable given some additional information about the partitions.

Consider a fragmented database system over n sites $\{S_i : i = 1, \ldots, n\}$ with corresponding fragments $\{F_i : i = 1, \ldots, n\}$. Suppose the system is divided into two partitions P_1 and P_2. As in Section 4.3, let δ be a transaction distribution submitted in P_1 and P_2, and L be any serializable execution of δ generated by a prevention protocol. Then $PIO(L)$ has a DITS. There are only two possible DITS's, i.e., $P_0P_1P_2P_f$ and $P_0P_2P_1P_f$.

Consider a DITS given by $P_0 P_1 P_2 P_f$. For every site S_i, we assume that many more local transactions are submitted at S_i than global transactions. We further assume that the number of local update transactions submitted at S_i is larger than the total number of global transactions submitted at all other S_j, $(j \neq i)$, that read some data objects from F_i.

Let us first introduce a set of notations which will be used in the following.

Γ_1 (Γ_2) : the set of transactions submitted at the sites in P_1 (P_2).

LC_1 (LC_2) : the set of local transactions submitted at the sites in P_1 (P_2).

G_1 (G_2) : the set of global transactions submitted at the sites in P_1 (P_2).

$\overline{G}_1$ $(\overline{G}_2)$: the set of global transactions submitted at the sites in P_1 (P_2) that read only data objects belonging to sites in P_1 (P_2).

LR_1 (LR_2) : the set of local read-only transactions submitted at the sites in P_1 (P_2).

With respect to the execution L, we define the following subsets :

τ_1 (τ_2) consists of all the transactions in Γ_1 (Γ_2) that are executed in L,

lc_1 (lc_2) consists of all the transactions in LC_1 (LC_2) that are executed in L,

g_1 (g_2) consists of all the transactions in G_1 (G_2) that are executed in L,

lr_1 (lr_2) consists of all the transactions in LR_1 (LR_2) that are executed in L, and

$\overline{g}_1$ $(\overline{g}_2)$ consists of all the transactions in $\overline{G}_1$ $(\overline{G}_2)$ that are executed in L.

For $i = 1, \ldots, m$, let A_{1i} be the set of local update transactions belonging to S_i.

B_{2i} : the set of global transactions submitted in P_2 that have read data objects belonging to S_i.

α_{1i} : consists of all the transactions in A_{1i} that are executed in L.

β_{2i} : consists of all the transactions in B_{2i} that are executed in L.

Note that $|A_{1i}|$ is much larger than $|B_{2i}|$ by assumption. For $i = 1, \ldots, m$, we also assume that the weak uniformity assumption (see Section 4.3) holds for the transaction distribution consisting of all the transactions in $A_{1i} \cup B_{2i}$. In this transaction distribution, all the transactions are submitted at their

home sites. In this particular case, this assumption means that, for any subset X of F_i $(i = 1, \ldots, m)$, the number of local transactions in A_{1i} that write into X is larger than the number of global transactions in B_{2i} that read some data objects from F_i. It follows from the analysis done in the proof of Theorem 4.3.1 that

$$|\alpha_{1i}| + |\beta_{2i}| \le |A_{1i}|, \text{ for } i = 1, \ldots, m. \tag{6.1}$$

A transaction in τ_1 is either a local update transaction, a local read-only transaction, or a global transaction. Hence

$$\tau_1 = \alpha_{11} \cup \alpha_{12} \cup \cdots \cup \alpha_{1m} \cup lr_1 \cup g_1, \text{ and}$$

$$\tau_2 = lc_2 \cup \overline{g}_2 \cup \beta_{21} \cup \beta_{22} \cup \cdots \cup \beta_{2m}.$$

Therefore,

$$|\tau_1| = \sum_{i=1}^{m} |\alpha_{1i}| + |lr_1| + |g_1|, \tag{6.2}$$

$$|\tau_2| \le |lc_2| + |\overline{g}_2| + \sum_{i=1}^{m} |\beta_{2i}|. \tag{6.3}$$

(6.3) is an inequality, because some of the β_{2i}'s may have nonempty intersection.

From (6.2) and (6.3), we have

$$|\tau_1| + |\tau_2| \le |lr_1| + \sum_{i=1}^{m} |\alpha_{1i}| + \sum_{i=1}^{m} |\beta_{2i}| + |g_1| + |lc_2| + |\overline{g}_2|. \tag{6.4}$$

It follows from (6.1) and (6.4) that

$$|\tau_1| + |\tau_2| \le |lr_1| + \sum_{i=1}^{m} |A_{1i}| + |g_1| + |lc_2| + |\overline{g}_2|. \tag{6.5}$$

Note that $(\bigcup A_{1i})_{i=1}^{m} \cup lr_1$ is a subset of LC_1. Therefore, it follows from (6.5) that

$$|\tau_1| + |\tau_2| \le |LC_1| + |G_1| + |LC_2| + |\overline{G}_2|. \quad \textbf{(6.6)}$$

On the other hand, if the DITS order in $PIO(L)$ is $P_0P_2P_1P_f$, then

$$|\tau_1| + |\tau_2| \le |LC_2| + |G_2| + |LC_1| + |\overline{G}_1|. \quad (6.7)$$

Let $h_1 = |G_1| - |\overline{G}_1|$ and $h_2 = |G_2| - |\overline{G}_2|$. Note that h_1 (h_2) is the number of global transactions submitted in P_1 that have read some data objects belonging to sites in P_2 (P_1). It follows from (6.6) and (6.7) that

$$|\tau_1| + |\tau_2| \le |LC_1| + |G_1| + |LC_2| + |G_2| - min(h_1, h_2)$$

$$= |\Gamma_1| + |\Gamma_2| - min(h_1, h_2).$$

Hence

$$\frac{|\tau_1| + |\tau_2|}{|\Gamma_1| + |\Gamma_2|} \le 1 - \frac{min(h_1, h_2)}{|\Gamma_1| + |\Gamma_2|},$$

and $1 - \frac{min(h_1, h_2)}{|\Gamma_1| + |\Gamma_2|}$ is an upper bound on the availability of δ.

Note that even if the sites in both P_1 and P_2 have no knowledge about the sizes of h_1 and h_2, they can execute the transactions in $LC_1 \cup \overline{G}_1$ and $LC_2 \cup \overline{G}_2$ without violating serializability. This gives availability $1 - \frac{(h_1 + h_2)}{|\Gamma_1| + |\Gamma_2|}$, which is near-optimal. That is, the sites in each partition can still execute all local transactions and global transactions that access only fragments whose home sites are in the partition. This shows that a fragmented database achieves a high availability.

Furthermore, if there is enough information available so that h_1 and h_2 can be computed by the sites in the two partitions, then the optimal availability is achievable. For example, if n_i, the number of global transactions submitted at S_i, is known to every site, and the likelihood that a fragment F_k is read by a transaction issued at S_i is the same for every $k \ne i$, then

$$h_1 = \frac{n-m}{n-1} \sum_{i=1}^{m} n_i \text{ , and}$$

$$h_2 = \frac{m}{n-1} \sum_{i=m+1}^{n} n_i \text{ .}$$

Hence $min(h_1, h_2)$ is computable in both of the two partitions. If $h_2 = min(h_1, h_2)$, then P_1 can execute all the transactions submitted in it, and P_2 has to give up those global transactions in $G_2 - \overline{G}_2$. If $h_1 = min(h_1, h_2)$, on the other hand, then P_2, instead of P_1, can execute all the transaction submitted in it, and P_1 has to give up those global transactions in $G_1 - \overline{G}_1$. In any case, the optimal availability is achievable.

CHAPTER 7

ANOTHER CONCURRENCY CONTROL SCHEME FOR FRAGMENTED EXECUTION

7.1. Another Scheme to Control Fragmented Execution

As described in Section 6.4, GTOC sends certification requests to remote sites to certify the values read by a GLOBAL-READ. After the GLOBAL-READ of a global transaction T_a has finished its reading from copies in the DB of T_a's home site S_i, certification requests are generated and sent to the global synchronizers (GS's) at the remote sites whose fragments were read. The certification request of T_a received by a remote site S_j is tested against $COMMIT_j$ and $WAIT_j$. If the request passes both tests, it is granted by the GS at S_j. Transaction T_a can execute its remaining part if all the recipient remote GS's reply positively to its certification requests. This scheme is an **active** scheme in the sense that the home site takes the initiative to send certification requests.

The second scheme, to be discussed in this section, is termed **passive**. Unlike GTOC, this scheme makes use of only global timestamps, which are generated by a system-wide global clock. (An implementation of the global clock was discussed in Section 6.4.) Every transaction, be it local or global, is assigned a timestamp by the global clock. In this scheme, transactions are scheduled using their timestamps. The updates of transactions, which contain the timestamps of the transactions, are broadcast to the other sites in their timestamp order. The copy of a data object in the DB has a timestamp which is the timestamp of the transaction which performed the last update

on it. Global concurrency control is needed only for global transactions, i.e., no global concurrency control is needed for local transactions. As in GTOC, local transactions executed under GTOS are given higher priority than global transactions. Thus, a local transaction trying to access a data object X will never be blocked by a global transaction which is accessing X.

The scheduler lets a global transaction T_a first read all the data objects in its readset and then assign a timestamp to T_a, denoted by $ts(T_a)$. (The method to assign a timestamp to a global transaction will be explained later in Section 7.2.) We refer to a data object belonging to a remote fragment as a **remote data object**. Note that the actual reading of a remote data object takes place locally from the copy of remote fragment. T_a waits until its home site finds out if all the values of remote data object read by T_a are **correct** with respect to T_a's timestamp. The correctness of a value of a remote data object read by a global transaction is defined as follows. Let X_i be the copy of a remote data object X belonging to F_j. The value of a copy X_i read by T_a at its home site S_i is **correct** for T_a if (1) the value of X_i was written at S_j by a transaction T_b with a timestamp $ts(T_b) < ts(T_a)$, and (2) no other transaction with a timestamp smaller than $ts(T_a)$ and larger than $ts(T_b)$ has written X at S_j. The timestamp $ts(T_b)$ is sent to S_i by S_j together with an update of T_b on X. Therefore, when $ts(T_a)$ is assigned to T_a, $ts(T_a)$ can always be set larger than t_b. Hence we can always make condition (1) hold. If the values of all remote data objects that T_a has read are correct, then T_a can be committed on completion. (Further details on committing a global transaction are described in Section 7.2.) In this way, all the execution generated will be serialized; the timestamps of the transaction provide a serialization order. The only problem is how to find out whether the value of a remote data object read by T_a is correct. We assume that the messages from the same site are broadcast and received by other sites in their timestamp order.

Suppose T_a has read a copy X_i of a remote data object belonging to F_j with a timestamp $t_1 < ts(T_a)$. If no update of X with a timestamp smaller than $ts(T_a)$ and larger than t_1 is received by

S_i, and an update of some data object belonging to F_j with a timestamp larger than $ts(T_a)$ is received by S_i, then S_i knows that the value of X_i is correct for T_a. If a new copy of X with a timestamp smaller than $ts(T_a)$ and larger than t_1 is received by S_i, then S_i knows that the value of X_i read earlier by T_a is not correct; in this case, T_a replaces the value of X by the new value and goes back to waiting until S_i can determine whether this new value is the correct value for T_a. However, if nothing is received from S_j, then S_i cannot know if the value of X_i is correct and hence the transaction may have to wait forever. Therefore, **timeout messages** which indicate the time at each site must be broadcast when needed. If S_i receives nothing but a timeout message with a timestamp larger than $ts(T_a)$ from S_j, then it knows that there is no new update of X generated between t_1 and $ts(T_a)$ at S_j. Hence the value of X_i is correct for T_a. The method used in this scheme is called **Global Timestamp Order Synchronization** (GTOS).

In Section 7.2, an architecture for a WADDS to implement GTOS is described. In particular, the execution of both local and global transactions under GTOS will be described. The management of timestamps and **virtual clocks** are crucial in GTOS. A virtual clock for a site S_j at another site S_i has the latest time of S_j, which is known to S_i by receiving messages from S_j. In Section 7.3, we will discuss the management of virtual clocks in detail. In Section 7.4, an implementation of GTOS will be described. The correctness of GTOS will be discussed in Section 7.5. In Section 7.6, we will discuss some performance issue of GTOS.

7.2. An Architecture for GTOS

In the WADDS architecture in which GTOS is implemented, there are five functional components at each site : the **transaction manager** (TM), the **scheduler**, the **database manager** (DM), the **update propagation manager** (UPM), the **timestamp manager** (TSM) and the

communication manager (CM). The scheduler has two subcomponents, the **home fragment scheduler** (HFS) and the **remote fragment scheduler** (RFS). The interconnection among these components is illustrated in Figure 7.2.1. To describe the functions of these components, we will now explain how GTOS processes local and global transactions.

A local transaction starts with a LOCAL-BEGIN operation, followed by a sequence of READ and WRITE operations, and ends with an END operation just as in GTOC. For simplicity, we assume that each global transaction consists of two steps. In Section 7.5, we will show that GTOS, with a slight modification, is also applicable to more general (non 2-step) transactions. A **2-step transaction** first executes all its read operations and then all its write operations. For a 2-step transaction T_a, we use D_a and $\hat{D_a}$ to denote its readset and writeset, respectively. T_a can thus be represented as $T_a = R[D_a]W[\hat{D_a}]$.

Let us use a 2-phase locking scheduler for the HFS. (We can use some other scheduler, but we choose a 2-phase locking scheduler for simplicity). In order to give higher priority to local transactions, GTOS allows a local transaction to preempt a global transaction in case the latter holds a lock on a data object required by the former transaction. Two types of locks are provided by the HFS: **high priority lock (h-lock)** and **low priority lock (l-lock)**. H-locks are used by local transactions, while l-locks are used by global transactions. High priority read lock, low priority read lock, high priority write lock, and low priority write lock are denoted, respectively, by hr-lock, lr-lock, hw-lock and lw-lock. As usual, read locks are compatible among themselves and incompatible with write locks, while write locks are incompatible with each other as well as with read locks. A local transaction requesting a hw-lock can preempt a lw-lock or lr-lock on a data object held by a global transaction. This aborts the latter transaction. The compatibility among h-locks and l-locks is shown in Figure 7.2.2. Any hr-lock or hw-lock request submitted when there is a queue of lock requests waiting to lock a data object is always inserted before all the lr-locks and lw-locks in the

queue. The RFS is a FIFO queue. Since updates from a remote site are received in their timestamp order and the RFS is a FIFO queue, these updates are submitted to the DM in their timestamp order. There is no need to control the read operations of global transactions accessing the remote fragments in the DB, because a global transaction will be given the correct value if it is not the value it read.

As shown in Figure 7.2.1, at each site S_i, there is a timestamp manager, named TSM. The TSM maintains a local clock C_i and a set of virtual clocks C_{ij}, one for each site S_j ($j \neq i$). Time $t(C_i)$ retrieved from C_i is the local clock value with the site identity of S_i appended to it at the least significant end to make it globally unique. Whenever the communication manager CM receives a message from another site, C_i is adjusted to a value that is one tick larger than the timestamp in the received message, if the received timestamp is larger than $t(C_i)$. With a clock defined in this way, the timestamps are unique and all timestamped events can be totally ordered [Lam78]. This global clock is implemented in the same way as the global clock used in GTOC described in Section 6.4.

The time of a virtual clock C_{ij} at a site S_i reflects the time of C_j known to S_i. More precisely, site S_i knows that the time $t(C_j)$ is at least as large as that of C_{ij}. In fact, two time values are associated with C_{ij}. In Section 7.3, we will explain how a TSM manages its virtual clocks and how the virtual clocks are used to determine whether the value of a remote data object read by a global transaction is correct.

At site S_i, when LOCAL-BEGIN of a local transaction T_a arrives, the TM allocates a workspace for T_a and sends hr-lock (hw-lock) requests to the HSF on behalf of T_a's READ (WRITE) operations. (Note that local transactions are not necessarily 2-step transactions). If a hr-lock is granted, then the READ operation is executed by the DM. If a hw-lock is granted, updating is done in the workspace. When the END operation of T_a arrives, the TM performs two steps. In the first step, it obtains $t(C_i)$ and assigns it to T_a. (Note that this is the first time that a timestamp is assigned to a local transaction.) Then it tells the DM to commit the updates and releases all the locks held by

T_a. In the second step, the TM submits a remote update request to the UPM on behalf of T_a. The request contains the committed values and $t(C_i)$. The UPM broadcasts the request to every other site through the CM, which is connected to the CM's at all other sites. The UPM uses a reliable broadcast protocol (e.g., [AwE84, GLB85]) to send out remote update requests. (The properties of a reliable broadcast protocol were mentioned in Section 6.2). The UPM broadcasts remote update requests in their timestamp order.

When the CM at a site S_j receives a remote update request from S_i, the request's timestamp is sent to S_j's TSM to update the virtual clock C_{ji}. To process the updates in the request, the UPM issues a write-only transaction to install them in the replica of F_i. The write operations of this transaction are submitted to the FIFO queue at the RFS. This completes the discussion of a local transaction.

A global transaction starts with a GLOBAL-BEGIN operation, followed by a set of READ operations, then a set of WRITE operations, and terminates with an END operation. The execution of a global transaction $T_a = R[D_a]W[\hat{D}_a]$ submitted at a site S_i consists of the following two phases.

Phase One [Reading and Locking].

When T_a's first operation, GLOBAL-BEGIN, arrives at S_i, the TM allocates a workspace for it. Then the TM submits a lr-lock request to the HFS for all the data objects in $D_a \cap F_i$ on behalf on T_a. For each remote fragment F_j, TM submits a read request to the RFS for all the remote data objects in $D_a \cap F_j$, if it is not empty. If a lr-lock submitted to the HFS or a read request submitted to the RFS on a data object is granted, the READ operation on the data object is executed and the value is retrieved into the workspace. After all data objects in D_a are read, the TM issues lw-locks for all data objects in $\hat{D}_a$. The first phase completes at this point. Even if a lw-lock is granted, T_a does not immediately execute the corresponding WRITE operation. Instead, this is done in the second phase.

Phase Two [Waiting and Commit/Abort].

The clock value $t(C_i)$ is now assigned to T_a as its timestamp $ts(T_a)$. Then T_a waits until the TM can determine that the values of all the remote data objects it has read are correct. If these values are correct, then all the l-locks of T_a are converted to h-locks and all its WRITE operations are executed in T_a's workspace. We convert the l-locks to h-locks to prevent a global transaction from being preempted by a local transaction after it has been determined that all the remote data objects it has read are correct. When T_a issues an END, T_a commits and releases all its locks. Then its updates are broadcast to all the other sites in the same way as the updates of a local transaction are broadcast. If S_i receives a new copy of a data object X and finds that the old value of X read earlier by T_a is not correct for T_a, then the TM replaces the value of X stored in the workspace by the new value and T_a waits again. □

Note that if we ignore the read operations of global transactions that access remote fragments, the HFS is a 2-phase locking scheduler. A timestamp is assigned to a transaction, be it local or global, after it has locked all the data objects in its readset and writeset that belong to its home fragment and before it releases these locks. Hence, the timestamp order is a serialization order of all transactions. In addition, if all values of remote data objects read by every global transaction are correct, then it is clear that all transactions can be serialized in their timestamp order.

The above description of the execution of a global transaction is not complete. In Section 7.3, we will discuss the management of virtual clocks. The algorithm used to determine whether the value of a remote data object read by a global transaction is correct will be described in Section 7.4.

7.3. Timestamp and Virtual Clock Management

In the following, a message broadcast by a site which contains the update of a transaction and its timestamp is called an **update message**. (In fact, update messages are the remote update requests discussed in Section 7.2.) If A is the set of updates of a transaction with timestamp t, the update message broadcast on behalf of the transaction is represented by $u(A, t)$. If A contains only a data object X, $u(X, t)$ is used to represent $u(\{X\}, t)$.

How does a site know that a global transaction has read the correct value of a remote data object? There are three possible cases to be discussed, which are illustrated in Figure 7.3.1. The time axes in all these figures refer to the global time, i.e., the time given by the clock C_i at site S_i. As before, the timestamp of a transaction T_a is denoted by $ts(T_a)$.

In Figure 7.3.1(a), an update message $u(X, t_1)$ is broadcast by site S_1 at time t_1 and is received by site S_2 at time t'_1. More precisely, t_1 is the timestamp of the transaction which issues the update message $u(X, t_1)$. Suppose that a global transaction T_2 submitted at S_2, with a timestamp $t'_2 > t'_1$, has read the value of X in $u(X, t_1)$ and is waiting to determine if the value read is correct. Later at time $t_2 > t'_2$, another update message $u(Y, t_2)$ on a data object Y is broadcast by S_1 and is received by S_2 at t'_3. If no other update of X is received between t'_1 and t'_3, S_2 at t'_3 can confirm that the value of X in $u(X, t_1)$ is the latest value of X written by a transaction with a timestamp smaller than $t'_2 = ts(T_2)$. Hence, S_2 confirms that the value of X read by T_2 is correct.

In the second case shown in Figure 7.3.1(b), a new update message $u(X, t_2)$ of X is broadcast by S_1 at $t_2 < t'_2$ and received by S_2 at $t'_3 > t'_2$. That is, when T_2 is waiting, a newer value of X is received which has a timestamp less than $t'_2 = ts(T_2)$. At t'_3, S_2 knows that the value of X in $u(X, t_1)$ is not the latest copy of X before t'_2. Hence, the value of X that T_2 has read is not correct. In this case, T_2 replaces the value of X by the new value and goes back to waiting until S_2 can determine if

the new value is correct for T_2.

In the third case (see Figure 7.3.1(c)), after the update message $u(X, t_1)$ is broadcast, no further update is broadcast from S_1. If this the situation remains, it will cause T_2 to wait for a very long time. To prevent this situation from arising, the timestamp manager TSM at every site S_i broadcasts **timeout messages** periodically with a fixed period Δ, called the **time tolerance.** If the latest update broadcast by S_i has timestamp t and since then, no update has been broadcast by S_i for a period of Δ, then S_i's TSM broadcasts a timeout message $tm(t+\Delta)$ at time $t+\Delta$, which carries the timestamp $t+\Delta$. When this timeout message arrives at any other site S_j, S_j can be sure that no new update has been broadcast by S_i since t.

In Figure 7.3.1(c), a timeout message $tm(t_2)$, $t_2 = t_1 + \Delta$, is broadcast after a period of Δ with no update. When the timeout message arrives at site S_2 at time t'_3, S_2 knows that it was correct for T_2 to read the value of X in $u(X, t_1)$.

Timeout messages and update messages are broadcast by a site in their timestamp order. At any site S_i, a timeout message with a timestamp t_2 cannot be broadcast before an update message with a timestamp $t_1 < t_2$, i.e., before a transaction with timestamp t_1 is completed. Otherwise, when the timeout message with timestamp t_2 arrives at another site S_j before the update message with timestamp t_1, S_j may mistake that no update has happened before t_2 at S_i.

The above scheme works correctly in deciding whether a global transaction has read correct values. However, it can cause a deadlock, which must be remedied. An example of deadlock is illustrated in Figure 7.3.2(a). An update message $u(X, t_1)$ broadcast at t_1 by site S_1 arrives at site S_2 at time t'_2, and at roughly the same time t_2, an update message $u(Y, t'_1)$ broadcast by S_2 at t'_1 arrives at S_1. A global transaction T_1 with $ts(T_1) = t_3$, which has read only the value of Y in $u(Y, t'_1)$, is waiting at S_1. Similarly, a global transaction T_2 with $ts(T_2) = t'_3$, which has read only the value of X

in $u(X, t_1)$, is waiting at S_2. No update has been broadcast since t_1 and t'_1. Suppose $t_4 = t_1 + \Delta > t_3$ and $t'_4 = t'_1 + \Delta > t'_3$. In this case, only the arrival of the timeout messages $tm(t'_4)$ and $tm(t_4)$ can unblock T_1 and T_2, respectively. However, these timeout messages cannot be sent out, because, as mentioned above, $tm(t_4)$ can be sent out only after T_1 has completed and its update message has been broadcast. Otherwise, $tm(t_4)$ would be sent out before the update message of T_1, which has timestamp $t_3 < t_4$. Similarly $tm(t'_4)$ can only be sent out after T_2 has completed. Hence, T_1 and T_2 wait for events blocked by each other. This is a deadlock and neither of the two transactions can proceed.

In order to solve the deadlock problem mentioned above, a global transaction T_a waiting at a site S_i with $ts(T_a) = t$ should let the other sites know that the time at S_i is already t. However, this cannot be achieved by sending out an update message for T_a, because T_a has not completed yet. In order to solve this dilemma, a **global-start message** $gs(t)$ which carries the timestamp t is broadcast when t is assigned to T_a. The timestamp t in the global-start message is called a **global-start timestamp.** In the example illustrated in Figure 7.3.2(b), global-start messages $gs(t_3)$ and $gs(t'_3)$ arrive at S_2 and S_1 at times t'_5 and t_5, respectively. By looking at the times in the global-start messages, S_1 (S_2) at t_5 (t'_5) knows that T_1 (T_2) has read the correct values, and T_1 (T_2) can proceed without waiting. Therefore, the deadlock between T_1 and T_2 is resolved.

We have mentioned above that update messages and timeout messages are sent out in their timestamp order. After the introduction of global-start message, this must be modified. The purpose of introducing global-start messages is to allow a global transaction T_a waiting at site S_i to inform the other sites of the time $ts(T_a)$. However, the broadcast of the update message of T_a is delayed until T_a is completed, even though the timestamp of the update message is also $ts(T_a)$. Hence, there is a time lag between the broadcasts of the global-start message and the update message of T_a. Some local transactions may be committed with timestamp larger than $ts(T_a)$ before T_a is completed. In

order not to delay the broadcast of the updates of these local transactions and the advance of the virtual clocks for S_i at other sites, some update messages of local transactions, timeout messages and global-start messages with timestamps larger than $ts(T_a)$ may be broadcast before the update message of T_a. Therefore, global-start messages, timeout messages and update messages of local transactions are broadcast in their timestamp order, *but the update message of a global transaction is broadcast at the time when the transaction is completed.* Therefore, some messages with larger timestamps may be broadcast before it, which violates the rule that messages are sent in their timestamp order.

The introduction of global-start messages raises another problem. By a global-start message, time information is sent out prematurely. For a local transaction, its timestamp is broadcast with its updates. In this case, there is no time lag between the arrival of an update and its timestamp. However, this does not hold any more once global-start messages are introduced for global transactions. As mentioned above, there is a time lag between the broadcasts of the global-start message and the update message of a global transaction. This time lag could cause a problem for a waiting global transaction when it is trying to determine whether a value read by it is correct. This problem will be illustrated by the following example.

Example 7.3.1. In Figure 7.3.3(a), an update message $u(X, t_1)$ of a data object X is broadcast at t_1 by S_1 and is received by S_2 at t'_1. At S_1, a global transaction T_2 is assigned a timestamp t_2 and a global-start time message $gs(t_2)$ is broadcast at t_2 and received by S_2 at t'_2. Assume that T_2 has updated only X and its update message $u(X, t_2)$ is broadcast after a waiting period at t_4, which arrives at S_2 at t'_4. Before T_2 completes at t_4, another global transaction T_3 at S_1 is assigned timestamp $t_3 < t_4$. Hence, a global-start message $gs(t_3)$ is broadcast by S_1 on behalf of T_3 at t_3 and received by S_2 at t'_3. Assume that no update of X is issued by S_1 after t_1 and before t_3.

Suppose that there is a global transaction T_4 submitted at S_2 with a timestamp t such that $t'_2 < t < t'_3$, and T_4 has read the value of X in $u(X, t_1)$. Both global-start messages $gs(t_2)$ and $gs(t_3)$ carry no information except their timestamps. When the global-start message $gs(t_3)$ is received at t'_3, it would indicate to S_2 that the time at S_1 is at least $t_3 > t$. Since no update of X has been broadcast after t_1 at S_1, S_2 would mistakenly regard the value of X in $u(X, t_1)$ as the correct value for T_4. However, the correct value of X for T_4 should be the value of X in $u(X, t_2)$. □

Example 7.3.1 shows that the introduction of global-start message may cause problem in identifying the correct values for a waiting global transaction. Note that the same problem occurs if the second global-start message $gs(t_3)$ in Figure 7.3.3(a) is replaced by a timeout message or an update message of a local transaction with the same timestamp t_3.

One way to remedy the above problem would be to stop broadcasting any message after a global-start message has been broadcast until the update message of the corresponding global transaction has been broadcast. However, this would delay the broadcast of the updates of all other local and global transactions. Another remedy is to let a local transaction with a larger timestamp to preempt a waiting global transaction, even if there is no need to do so because the union of the readset and writeset of the local transaction has no intersection with the writeset of the global transaction. However, this would increase significantly the chance of preemption of global transactions.

We now propose a more efficient way to remedy this problem. Firstly, we modify the global-start message $gs(t_a)$ of a global transaction $T_a = R[D_a]W[\hat{D}_a]$ to carry not only the timestamp $t_a = ts(T_a)$, but also the writeset $\hat{D}_a$ of T_a. We use $WS(gs(t_a))$ to represent the writeset in $gs(t_a)$. In this way, when $gs(t_a)$ is received by a site S_j, even though the update of T_a has not yet arrived at S_j, a global transaction T_k with a timestamp larger than t_a waiting at S_j can decide whether it should wait to read the update of T_a by checking if $WS(gs(t_a)) \cap RS(T_k) = \emptyset$, where $RS(T_k)$ is the readset of

T_k. Secondly, when a global transaction T_a broadcasts its update, a **global-completion message** $gc(A, t_a, t)$ is broadcast instead of an update message, which carries the set A of updates by T_a and two timestamps $t_a = ts(T_a)$ and t, where t is the time at which T_a committed. If A contains only one data object Y, $gc(Y, t_a, t)$ is used to represent $gc(\{Y\}, t_a, t)$. The timestamp t is called the **global-completion timestamp** of T_a. Note that the timestamp of the updates in a global-completion message $gc(A, t_a, t)$ is t_a, not t. When these updates are stored in the DB of a site, the timestamp t_a are attached to them. Global-completion messages from the same site are broadcast in the order of their global-completion timestamps.

We now examine how the above affects the execution of global and local transactions. Suppose $T_a = R[D_a]W[\hat{D}_a]$ and $T_b = R[D_b]W[\hat{D}_b]$ are two global transactions waiting at the same site S_i, where $ts(T_a) = t_a < ts(T_b) = t_b$. Therefore $gs(t_a)$ was broadcast before $gs(t_b)$. Suppose T_b commits earlier than T_a. This is possible if $\hat{D}_a \cap (D_b \cup \hat{D}_b) = \emptyset$. In this case, the global-completion message of T_b will be broadcast before that of T_a, even though $ts(T_a) < ts(T_b)$. Note that the earlier arrival of T_b's updates would not cause any problem for those global transactions waiting to read the updates of T_a, because $\hat{D}_a \cap \hat{D}_b = \emptyset$. Therefore, the updates of global transactions may sometimes be broadcast out of their timestamp order.

Consider another scenario in which a local transaction T_k is submitted at a site S_i while a global transaction $T_a = R[D_a]W[\hat{D}_a]$ is waiting at S_i. If $\hat{D}_a$ overlaps with either the writeset or readset of T_k, then T_a will be preempted by T_k; otherwise, T_k may be committed earlier than T_a. Hence the update message of T_k may be broadcast earlier than that of T_a, even though $ts(T_k) > ts(T_a)$. Based on the above examples, we make the following observation : *the updates of a transaction with a larger timestamp may be broadcast earlier than that of a global transaction with a smaller timestamp, if their writesets do not overlap. However, the updates on a data object X are still broadcast by its home site in their timestamp order.*

Since the update message of a global transaction is replaced by a global-completion message, from now on, update messages are used only for local transactions. For clearness, we rename update messages as **local-update messages**. The arrival of a global-completion message indicates the completion of a global transaction whose timestamp has been announced by a global-start message broadcast earlier.

Let us see how the inclusion of a writeset in a global-start message and the introduction of global-completion messages solve the problem raised in Example 7.3.1. The update message $u(X, t_2)$ in Figure 7.3.3(a) is replaced by a global-completion message $gc(X, t_2, t_4)$ in Figure 7.3.3(b). After $gs(t_3)$ is received by S_2 at t'_3, S_2 knows that the time of clock C_1 at S_1 is at least $t_3 > t$. Also, it knows that the smallest global-start timestamp of the global transactions waiting at S_1 is $t_2 < t$. At this point, S_2 can determine whether T_4 has to wait for the global-completion message of T_2. If $WS(gs(t_2)) \cap RS(T_4) = \emptyset$, (this doesn't hold for the example in Figure 7.3.3(b)), then T_4 does not have to wait for the global-completion message of T_2 and the value of X in $u(X, t_1)$ is correct for T_4. Otherwise, T_4 has to wait for $gc(X, t_2, t_4)$. In the latter case, after $gc(X, t_2, t_4)$ has arrived, S_2 knows that T_2 is no longer a waiting transaction and the smallest global-start timestamp of the global transactions waiting at S_1 becomes $t_3 > t$. Hence, S_2 knows that T_4 no longer has to wait for the update from other global transactions and the value of X in $gc(X, t_2, t_4)$ is correct for T_4.

According to the above discussion, S_2 must keep track of two types of information sent from S_1. The first is the largest timestamp, t_l, received so far. S_2 can infer that the time at S_1 is not smaller than t_l. The second is the list of global-start messages of the global transactions waiting at S_1. With the writesets in the global-start messages received by S_2, S_2 can determine whether a waiting global transaction at S_2 has to wait for some global-completion messages from S_1. This observation forms the basis of the virtual clock management which will be discussed below.

In the above discussion, four kinds of messages have been discussed : timeout messages, local-update messages, global-start messages and global-completion messages. The first three kinds of messages carry only one timestamp. A global-completion message carries two timestamps, and we refer to its global-completion timestamp as the timestamp of the message. After the introduction of completion timestamp, there is no more violation in the sending order of messages, i.e., *all messages are broadcast and received in their timestamp order.*

Corresponding to the four kinds of messages, there are four kinds of timestamps. The timestamp in a local-update message of a local transaction is called a **local-update timestamp**. The timestamp in a timeout message is called a **timeout timestamp**. We have already defined a global-start timestamp and a global-completion timestamp.

We are now in a position to discuss the management of virtual clocks. As mentioned in Section 7.2, at each site S_i, a set of virtual clocks C_{ij}, one for each remote site S_j $(j \neq i)$, are maintained. The virtual clock C_{ij} has two **virtual times**. The **upper virtual time** $ut(C_{ij})$ is the largest timestamp that S_i has received from S_j. The **lower virtual time** $lt(C_{ij})$ is the smallest global-start timestamp received by S_i from S_j such that the corresponding global transaction is still waiting at S_j. If no such global-start timestamp has been received or no global transaction associated with the received global-start timestamps is waiting at S_j, then $lt(C_{ij}) = ut(C_{ij})$. These two virtual times are used by S_i to determine if the values of remote data objects read by a global transaction at S_i are correct. Thus C_{ij} is actually a list of timestamps, which initially contains the smallest timestamp which was received in a timeout message. The timestamps in C_{ij} are ordered by their values. The upper virtual time $ut(C_{ij})$ is given by the largest timestamp in C_{ij}. The lower virtual timestamp $lt(C_{ij})$ is given by the smallest timestamp in C_{ij}. Whenever a message with timestamp t arrives at S_i from a site S_j, the TSM of S_i uses the following procedure to update the list C_{ij}.

Before presenting a formal procedure, we briefly describe the idea behind it. Since local-update timestamps and timeout timestamps will be handled in the same way, we call them **independent timestamp**. Suppose a timestamp t from S_j arrives at S_i.

(1) Suppose t is an independent timestamp or a global-start timestamp. Since t is larger than any timestamp in C_{ij}, $ut(C_{ij})$ should be advanced to t. If C_{ij} contains a global-start timestamp, it must be smaller than t and $lt(C_{ij})$ should remain unchanged. If there is no global-stamp in C_{ij}, $lt(C_{ij})$ should be advanced to t. This is done by first deleting all independent timestamps in C_{ij}, if any, and then appending t to C_{ij}. From the above discussion, it can be seen that if t is an independent timestamp or a global-start timestamp, then we have only one of the following two cases after t has been processed. (a) C_{ij} contains nothing but an independent timestamp. (b) C_{ij} is a list of some global-start timestamps and at most one independent timestamp. In case (b), the independent timestamp, if any, is always the largest timestamp in C_{ij}. Note that when a global-start timestamp t is appended to C_{ij}, the writeset $WS(gs(t))$ of the associated global-start message $gs(t)$ is stored in TM.

(2) If t is a global-completion timestamp which arrives in a global-completion message $gc(A, t', t)$, then there must be a global-start timestamp t' in C_{ij}. It follows from the discussion in (1) that C_{ij} contains a list of global-start timestamps and at most one independent timestamp. The arrival of the global-completion timestamp t indicates that the global transaction with timestamp t' is no longer waiting. Therefore, t' should be removed from C_{ij}. After the removal of t', $lt(C_{ij})$ is equal to the smallest remaining global-start timestamp in C_{ij}, if any; if no global-start timestamp remains, then $lt(C_{ij})$ will advance to t. Also, $ut(C_{ij})$ should be advanced to t, because it is the largest timestamp received from S_j. This is done by first deleting t' and the only independent timestamp from C_{ij}, if any, then converting t to an independent timestamp and appending it to C_{ij}. Since the global transaction with timestamp t' is no longer

waiting, the value of t simply indicates the latest time at S_j. This is why t is converted to an independent timestamp. When a global-start timestamp is deleted from C_{ij}, the writeset associated with it is no longer needed and hence is removed from the TM at site S_i.

Procedure to maintain a virtual clock C_{ij} at site S_i

Initialization : C_{ij} is initialized to a list containing only the smallest independent timestamp = 0.

Input : a message with timestamp t from site S_j

(1) If t is a simple or global-start timestamp, then delete all independent timestamps in C_{ij}, if any, and append t to C_{ij}.

(2) If t is a global-completion timestamp which arrives in $gc(A, t', t)$, then delete t' and all independent timestamps from C_{ij}, if any, convert t to an independent timestamp and append it to C_{ij}. □

7.4. Global Timestamp Ordering Synchronization Algorithm

In this section, we describe the method used by GTOS for scheduling global and local transactions. The execution of a local transaction has been described in Section 7.2. The main concern of GTOS is to ensure that all the values read by a global transaction are correct. The procedure for executing a global transaction consists of two phases and was briefly described in Section 7.2. In the following, its second phase, which provides a mechanism to determine if the values read are correct, will be described in more detail.

Procedure for the execution of a global transaction

Input : a global transaction $T_a = R[D_a]W[\hat{D}_a]$ submitted at S_i

Phase One [Reading and Locking].

As described in Section 7.2.

Phase Two [Waiting and Commit/Abort].

Assign clock value $t(C_i)$ to T_a as its timestamp. Broadcast a global-start message containing T_a's timestamp and writeset. Then execute the following steps.

(1) Make T_a wait until either condition C1, C2, or C3 is true, where

C1 : there exists a remote fragment F_j and a data object $X \in D_a \cap F_j$ such that a new update of X with a timestamp t_1 is received by S_i at t_2, where $t_1 < ts(T_a) < t_2$,

C2 : for all remote sites S_j (with fragment F_j) such that $F_j \cap D_a \neq \emptyset$, (i) $ut(C_{ij}) > ts(T_a)$, and (ii) $lt(C_{ij}) > ts(T_a)$ or $WS(gs(t_k)) \cap D_a = \emptyset$, for all global-start timestamps $t_k < ts(T_a)$ in C_{ij}, where $gs(t_k)$ is the global-start message carrying t_k,

C3 : T_a is preempted by a local transaction.

(2) If C1 is true, then replace the old value of X by the new value in T_a's workspace. Then goto (1).

(3) If C2 is true, then convert all the l-locks of T_a to h-locks and execute all its WRITE operations on the data objects in $\hat{D}_a$. Then commit T_a, retrieve a timestamp t, release all the locks held by T_a, and broadcast a global-completion message for T_a with t as its global-completion timestamp. The global-completion message also contains T_a's updates and timestamp.

(4) If C3 is true, then abort T_a and broadcast to all other sites a global-completion message, which contains the timestamp of T_a and a global-completion timestamp which is the time of abortion of T_a, but no update. □

Note that if C1 is true, the value of X that T_a has read is not correct for T_a. GTOS replaces the old value of X in F_j by the new one and makes T_a wait again.

If C2 is true, condition (i) implies that the times of C_j at all these remote sites S_j have passed $ts(T_a)$. For each of these remote site S_j, if $lt(C_{ij}) > ts(T_a)$, then no waiting global transaction at S_j has timestamp smaller than $ts(T_a)$. On the other hand, if $WS(gs(t_k)) \cap D_a = \emptyset$ for all global-start timestamps $t_k < ts(T_a)$ in C_{ij}, then T_a does not have to wait for the update of any waiting global transaction at S_j. In either case, it can be concluded that T_a has read the correct values from F_j. Hence, T_a can execute its write operations and commit. The reason that we convert the l-locks to h-locks is to protect T_a from being preempted by a local transaction at this stage. Since T_a is no longer waiting for any remote message and is about to commit, there is no need to abort it because of a competing local transaction.

If C3 is true, T_a is preempted because it is holding a l-lock on a data object on which a local transaction is requesting an incompatible h-lock. In this case, T_a is aborted. When T_a is aborted, a global-completion message with no update has to be broadcast to delete T_a's global-start timestamp from the virtual clocks at all remote sites. Otherwise, there will be some dangling global-start timestamps.

Example 7.4.1. Figure 7.4.1 is a possible time chart of message broadcasts from a site S_i. The upper horizontal axis records the timestamps of the messages. The lower horizontal axis records the times when the corresponding messages are broadcast. In the beginning, two timeout messages are sent out separated by a time interval of Δ. Then two local transactions T_{i1} and T_{i2} are executed. Their local-update messages are broadcast at the times when timestamps are assigned to the transactions. When a timestamp is assigned to a global transaction T_{i3}, a global-start message is sent out. After it has finished a waiting period, it executes all its write operations and a global-completion message is broadcast at the end. In the case of global transaction T_{i4}, after it has broadcast a global-

start message, it is preempted twice by local transactions T_{i5} and T_{i6}. Eventually, T_{i4} is restarted with a new timestamp and completed after T_{i6}. □

7.5. Correctness of GTOS

We first show that GTOS is deadlock-free. A set of sites Ω deadlocks, only if there are some waiting global transactions at $S_k \in \Omega$, and each site S_k is waiting for some message(s) from some other sites in Ω. Without loss of generality assume that all global transactions at the sites in Ω are waiting forever. To see that this is impossible, suppose that $T_{im} = R[D_{im}]W[\hat{D}_{im}]$ belonging to site S_i has the smallest global-start timestamp t_{im} among all the waiting global transactions. Note that a global transaction waits only if it is in phase two and none of the three conditions C1, C2, and C3 holds. (See the procedure for executing a global transaction in Section 7.4.) Suppose all global-start messages have been received. For every remote site S_j (with fragment F_j), such that $D_{im} \cap F_j \neq \emptyset$, let t_{jk} be the smallest timestamp of the waiting transactions at S_j. The set consisting of all these smallest timestamps is represented by ST. C_{ij} does not contain any global-start timestamp smaller than t_{jk}. Otherwise, t_{jk} would not be the smallest. Therefore, C_{ij} contains only an independent timestamp. (Refer to the discussion of virtual clock in Section 7.3.) In this case, both $ut(C_{ij})$ and $lt(C_{ij})$ are advanced to $t_{jk} > t_{im}$. (See (1) in the procedure for maintaining virtual clocks, in Section 7.3). After all the global-start timestamps in ST have been received by S_i, condition C2 in GTOS becomes true for T_{im} and T_{im} is unblocked. This shows that GTOS is deadlock-free.

It is relatively easy to see that any execution generated is serializable in timestamp order. No matter whether a transaction T_a is local or global, the timestamp of T_a is assigned to it after it has locked all the data objects in its readset and writeset that belong to its home fragment, and before any of these locks is released. Since the HFS at each site uses 2-phase locking, the timestamp order of all

the transactions belonging to a site is compatible with the order of their lock-points [BSW79]. Hence, a serial execution in their timestamp order of all the transactions belonging to a site preserves all read-from relations among them. As for a global transaction T_k submitted at S_i, if it has read a copy $X_i \in F_j$ written by a transaction T_l (belonging to site S_j), then GTOS ensures that the value read is correct for T_k. In other words, T_l is the transaction with the largest timestamp smaller than $ts(T_k)$ that has written X. Hence, the read-from relation between T_k and T_l is maintained in a serial execution in which all the transactions are ordered by their timestamps. This proves that any execution generated by GTOS is serializable.

In GTOS, we cannot bound the waiting time of a global transaction because it may be preempted many times by local transactions. In order to avoid infinite waiting, the system or the user can assign different modes to a global transaction. We propose two different modes, **high priority mode (hp-mode)** or **low priority mode (lp-mode)**, indicating two levels of priority. A global transaction $T_a = R[D_a]W[\hat{D_a}]$ in the hp-mode requests h-locks for all the data objects in $(D_a \cup \hat{D_a}) \cap F_i$, where F_i is its home fragment; if it is in the lp-mode, it requests l-locks for all these data objects. Since a hp-mode global transaction holds h-locks on data objects in its readset and writeset, no local transaction can preempt it. A user can explicitly specify the hp-mode for a global transaction so that its execution is guaranteed. Alternatively, a TM can convert a global transaction, whose waiting time has exceeded a predefined limit, from the lp-mode to hp-mode. Of course, we have to trade this off with the blocking of some local transactions that are competing with a hp-mode global transaction for accessing the same data object(s).

As the last remark, we mention that global transactions can be generalized from 2-steps to multi-steps. A timestamp can be assigned to a global transaction when it commits, i.e., after all the read and write operations have been executed, instead of before any write operation is executed. (Updates are recorded only in a workspace and changes are reflected in the database only if the

transaction commits). If this is done, read and write operations can interleave and the 2-step requirement is not necessary. However, step (2) in GTOS has to be modified in this case. If condition C1 in the procedure for executing a global transaction (see Section 7.4) is true, i.e., a new copy of a data object read by a waiting global transaction T_a has arrived at T_a's home site, T_a cannot just read the new copy. T_a may have to redo its computation, if the computation depends on the value of this new copy.

7.6. Performance of GTOS

In the following, we will analyse the performance of transaction execution under GTOS. Under GTOS, a local transaction is never blocked by a waiting global transaction in the lp-mode. Since we don't assume a frequent occurrence of hp-mode global transactions, the blocking caused by them will be insignificant. The execution time of a local transaction consists only of local processing time and no communication time. Hence, there should be no significant difference between the response time for a local transaction under GTOS and GTOC.

Now we want to estimate the time that a global transaction has to wait under GTOS after it is assigned a timestamp and before it proceeds to execute its write operations or before it is preempted by a local transaction. Let ε be the maximum difference between the clocks at any pair of sites. We assume that the fastest local clock remains to be the fastest for some time until the clocks are reset. (This assumption may not be very practical, but it provides a basis to do the estimation.) We want to show that ε is bounded by $\alpha_m + \Delta$, where α_m is the maximum time required to send a message from one site to another in the underlying network, measured by the fastest local clock. This is derived as follows. Let the site with the fastest local clock be S_i. Suppose S_i broadcasts a message at time t. When this message is received by another site S_j, the local clock at S_j is advanced to $t+1$. At the

same time the local clock at S_i by our assumption must be larger than $t+1$ and is at most $t+\alpha_m$. Therefore, at any time when S_j receives a message from S_i, the difference of the clock values at S_i and S_j is bounded by α_m. If there is no message to be broadcast, S_i sends out timeout messages in every Δ units of time. Therefore, during the time when S_j receives no message from S_i, the difference between their local clocks is at most $\alpha_m+\Delta$, i.e., $\varepsilon \le \alpha_m+\Delta$.

Let W_g be the time that a global transaction has to wait until it can proceed to execute its write operations or until it is preempted by a local transaction. There are two cases that a global transaction T_a with timestamp t_a has to wait. In the first case, it waits for some messages but not for the global-completion message from a waiting global transaction. In the second case, it is waits for the global-completion messages of some waiting global transactions as well as other messages. In the first case, after a waiting period of at most ε, the local clocks at all other sites will pass t_a. After waiting for additional $\Delta+\alpha_m$ time units, T_a will definitely receive all messages it was waiting for. In other words, within a period of $\varepsilon+\Delta+\alpha_m$, T_a is either aborted because condition C3 in GTOS becomes true, or T_a can start to execute its write operations because condition C2 in GTOS becomes true. Condition C1 may become true many times during the waiting period and T_a goes back to wait every time it has read a new value. However, this does not affect the bound $\varepsilon+\Delta+\alpha_m$ on the waiting time. Hence W_g is bounded by $\varepsilon+\Delta+\alpha_m$ and the average value of W_g is $\varepsilon+\Delta/2+\alpha_a$, where α_a is average time required to send a message from one site to another site.

In the second case, at any real time t, let WT be the set of all global transactions that are waiting at t. The size of WT depends on t. Let $G=(WT, E)$ be a directed graph in which the nodes are the transactions in WT and for any two transactions T_i and T_j in WT, $(T_i, T_j) \in E$ if and only if T_j is waiting for the global-completion message of T_i. The graph G is acyclic; otherwise, there would be a deadlock among the transactions in WT. Let $(T_1, T_2, \ldots, T_h)$ be a longest path in G. We will show that the waiting time W_g of any global transaction in WT is bounded by $\varepsilon+\Delta+h\,\alpha_m$. Note that,

for $i = 2, \ldots, h$, the timestamp of T_i is larger than that of T_{i-1}, and T_i is waiting for the global-completion message(s) of T_{i-1} and possibly some of the transactions in $\{T_1, \ldots, T_{i-1}\}$. Since T_1 is not waiting for the global-completion message of any global transaction in WT, it follows from the above discussion that the waiting time of T_1 is bounded by $\varepsilon + \Delta + \alpha_m$. In the worst case, the global-completion message of T_1 is sent after T_1 has waited for $\varepsilon + \Delta + \alpha_m$ units of time. This global-completion message is received by T_2 after T_2 has waited for $\varepsilon + \Delta + 2\alpha_m$. By induction, the waiting time for T_h is bounded by $\varepsilon + \Delta + h\alpha_m$. (Note that in the above discussion, we have ignored the time for processing all its write operations after a global transaction has finished its waiting period, because this time is insignificant compared with its waiting time.)

In the ideal case, in which global transactions are very rare, a global transaction seldom has to wait for the global-completion messages of other global transactions. Thus, the waiting time W_g is $\varepsilon + \Delta/2 + \alpha_a$ on average. In general, Δ is set much smaller than α_a, because the smaller Δ is, the shorter is the waiting time of a global transaction. Since $\varepsilon \leq \alpha_m + \Delta$, the average value of W_g is approximately equal to $2\alpha_a$. Recall that, under GTOC, if all certification requests of a global transaction are granted by remote GS's (global synchronizers) without much waiting, the time spent for communication is also $2\alpha_a$. Therefore, if global transactions are very rare, there is no significant difference between the waiting times for a global transaction under GTOC and GTOS. However, if the speeds of the clocks at different sites are very close to each other so that ε is much smaller than α_a, then the waiting time W_g under GTOS would be much smaller than $2\alpha_a$ and hence GTOS would perform better than GTOC in this case.

If we compare GTOS with the primary copy locking scheme (PC), the result would be similar to those in Section 6.5. In particular, the equations (6.5.1) and (6.5.2) still hold when GTOC is replaced by GTOS. Hence, in a WADDS in which global transactions are rare, GTOS, as well as GTOC, is definitely better than PC and other conventional concurrency control schemes, in which

the updates of a local transaction must be sent out before it can be completed.

When comparing GTOC with GTOS, it can be seen that they have different flavors. However, it is not clear which is better. Under GTOC, a global transaction may have to wait in a retry list for other global transactions, when it is waiting for remote sites to grant its certification requests. On the other hand, under GTOS, a global transaction has to wait for messages from other sites to determine whether the values it has read are correct. Also, it may be preempted many times before it can be completed. The control mechanism of GTOS is simpler than that of GTOC. A site under GTOS can decide by itself if a global transaction should be committed or aborted by simply waiting for messages. In contrast, a site under GTOC has to run a distributed algorithm to get remote sites to certify the GLOBAL-READ of a global transaction. Also, at every site S_i, GTOC has to consume a considerable amount of space to store two lists of records, $COMMIT_i$ and $WAIT_i$. As for GTOS, a drawback is the cost of broadcasting timeout messages. In any case, they both serve the purpose in that local transactions enjoy good response time and high availability. Moreover, only global transactions have to be controlled by a global concurrency control.

CONCLUSION

If reliability and availability are adopted as design goals, then replication is indispensable in a distributed database system, even though replication substantially increases the complexity of the control algorithm. Unfortunately, in a partitioned database system, degradation of availability is inevitable.

In this thesis, we have shown that an execution generated by a prevention protocol in a partitioned database can be characterized by a partition IO graph; an execution is serializable if and only if the partition IO graph has a DITS. We have also shown that every execution generated by a prevention protocol is TC-serializable.

We have derived an upper bound on the availability of any database system with two partitions. The weak uniformity assumption on the transaction distribution submitted is more general than the uniformity assumption used elsewhere [COK86]. We have shown that our upper bound is achievable in a fragmented database system.

The inherent conflict between serializability and availability in a general database system has forced us to introduce a model with less generality. We have demonstrated that both serializability and high availability are achievable in a fragmented database. The activity of transactions in a fragmented database system is rather restricted. However, this model is applicable in many situations, particularly in a wide-area distributed database system, in which communication delay is substantial. We have chosen a policy which favors local transactions over global transactions. Under this policy, no local transaction will be blocked due to a global transaction.

We have introduced a fragmented execution to model an execution in a fragmented database system. It was proved that a fragmented execution is serializable if its GOS graph is acyclic. This

result shows that global concurrency control is necessary only for global transactions in a fragmented database.

Two algorithms, Global Timestamp Order Certification (GTOC) and Global Timestamp Order Synchronization (GTOS), are proposed in this thesis to synchronize a partition execution. The former is an active scheme which sends out certification requests to remote sites. If all certification requests are granted by the remote sites, a global transaction can execute and broadcast its update. The latter is a passive scheme in which a global transaction waits until it is determined whether the copies from remote sites are correct. We have also shown that these two algorithms perform better than other conventional schemes such as primary copy locking, if they are applied to a fragmented database system.

REFERENCES

References

[ASC85] A. E. Abbadi, D. Skeen and F. Cristian, An Efficient, Fault-Tolerant Protocol for Replicated Data Management, *Proc. 4th ACM Symposium on Principles of Database Systems*, Mar 1985, 215-229.

[AbT86] A. E. Abbadi and S. Toueg, Availability in Partitioned Replicated Databases, *Proc. 5th ACM Symosium on Principles of Database Systems*, Mar 1986, 240-251.

[AlD76] P. A. Alsberg and J. D. Day, A Principle for Resilient Sharing of Distributed Resources, *Proc. of the 2nd International Conference on Software Engineering*, Oct 1976, 562-570.

[AwE84] B. Awerbuch and S. Even, Efficient and Reliable Broadcast is Achievable in an Eventually Connected Network, *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984, 278-281.

[BGS86] D. Barbara, H. Garcia-Molina and A. Spauster, Protocols for Dynamic Vote Assignment, *Proc. 5th ACM Symposium on Principles of Distributed Computing*, Aug. 86, 195-205.

[BSW79] P. A. Bernstein, D. W. Shipman and S. W. Wong, Formal Aspects of Serializability in Database Concurrency Control, *IEEE Trans. on Software Eng. SE-5*, 3 (May 1979), 203-216.

[BeG81] P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, *ACM Computing Surveys 13*, 2 (June 1981), 185-221.

[BeG83] P. A. Bernstein and N. Goodman, The Failure and Recovery Problem for Replicated Databases, *Proc. 2nd ACM Symposium on Principles of Distributed Computing*, Aug. 1983, 114-122.

[BGR83] B. T. Blaustein, H. Garcia-Molina, D. R. Ries, R. M. Chilenskas and C. W. Kaufman, Maintaining Replicated Database Even in the Presence of Partitions, *Proc. IEEE EASCON Conference*, 1983, 1-8.

[BlK85] B. T. Blaustein and C. W. Kaufman, Updating Replicated Data During Communications Failures, *Proc. of the 11th Intl. Conf. on Very Large Databases*, Aug 1985, 49-58.

[Cas81] M. A. Casanova, The Concurrency Control problem for Database Systems, in *Lecture Notes in Computer Science 116*, Springer Verlag, Berlin, 1981.

[COK86] B. A. Coan, B. M. Oki and E. K. Kolodner, Limitations on Database Availability When Networks Partition, *Proc. 5th ACM Symposium on Principles of Distributed Computing*, Aug 1986, 187-194.

[Dav84] S. B. Davidson, Optimism and Consistency in Partitioned Distributed Database Systems, *ACM Transactions on Database Systems 9* , 3 (Sept 1984), 456-481.

[DGS85] S. B. Davidson, H. Garcia-Molina and D. Skeen, Consistency in partitioned networks, *ACM Computing Surveys 17*, 3 (September 1985), 341-370.

[Dol82] D. Dolev, The Byzanitine Generals Strike Again, *J. of Algorithms 3*, (1982), 14-30.

[EaS83] D. Eager and K. C. Sevcik, Achieving Robustness in Distributed Database Systems, *ACM Trans. Database Systems 8*, 3 (Sept, 1983), 354-381.

[FLP85] M. Fischer, N. Lynch and M. Paterson, Impossibility of Distributed Consensus with One Faulty Process, *J. ACM 32*, 2 (1985), 374-382.

[GAB83] H. Garcia-Molina, T. Allen, B. Blaustein, R. M. Chilenskas and D. R. Ries, Data-Patch: Integrating Inconsistent Copies of a Database after a Partition, *Proc. 3rd IEEE Symposium on Reliability of Distributed Software and Database Systems,* Oct 1983, 38-46.

[GLB85] H. Garcia-Molina, N. Lynch, B. Blaustein, C. Kaufman, S. Sarin and O. Shmueli, Notes on a Reliable Broadcast Protocol, Tech. Rep. CCA-85-08, Computer Corporation of America, Dec 1985.

[GaK87] H. Garcia-Molina and B. Kogan, Achieving High Availability in Distributed Databases, *Proc. 3rd International Conf. on Data Engineering,* Feb, 1987.

[Gif79] D. K. Gifford, Weighted Voting for Replicated Data, *Proc. 7th ACM Symposium on Operating System Principles,* Dec 1979, 150-162.

[Gra78] J. Gray, *Notes on Database Operating Systems. Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60,* Springer-Verlag, New York, 1978.

[IKM87] T. Ibaraki, T. Kameda and T. Minoura, Serializability with constraints, *ACM Trans. Database Systems 12,* 3 (Sept 1987), 429-452.

[Jaj87] S. Jajodia, Managing Replicated Files in Partitioned Distributed Database Systems, *Proc. 3rd Int. Conf. on Data Eng.,* Feb. 87, 412-418.

[KoG87] B. Kogan and H. Garcia-Molina, Update Propagation in Bakunin Data Networks, *Proc. 6th ACM Symposium on Principles of Distributed Computing,* Aug 1987, 13-26.

[Lam78] L. Lamport, Time, Clocks and the Ordering of Events in a Distributed Multiprocess Systems, *Comm. ACM 21,* (July 1978), 558-564.

[LBS86] N. Lynch, B. Blaustein and M. Siegel, Correctness Conditions for Highly Available Replicated Databases, *Proc. 5th ACM Symposium on Principles of Distributed*

Computing, Aug 1986, 11-28.

[MiW82] T. Minoura and G. Wiederhold, Resilient Extended True-Copy Token Scheme for a Distributed Database System, *IEEE Transactions on Software Engineering SE8*, May 1982, 173-189.

[NoA83] A. D. Norman and M. Anderton, Empact, a distributed database application, *Proc. AFIPS Nat. Computer. Conf. 52*, (1983), 203-217.

[Pap79] C. H. Papadimitriou, The Serializability of Concurrent Database Updates, *J. ACM 26*, 4 (Oct. 1979), 631-653.

[PSL80] M. Pease, R. Shostak and L. Lamport, Reaching Agreement in the Presence of Faults, *J. ACM 27*, (1980), 228-234.

[SBK85] S. K. Sarin, B. T. Blaustein and C. W. Kaufman, System Architecture for Partition-Tolerant Distributed Databases, *IEEE Transactions on Computers C-34*, 12 (Dec 1985), 1158-1163.

[Sar86] S. K. Sarin, Robust Application Design in Highly Available Distributed Databases, *Proc. Fifth Symposium on Reliability in Distributed Software and Database Systems*, Jan 1986, 87-94.

[SkW84] D. Skeen and D. D. Wright, Increasing Availability in Partitioned Networks, *Proc. 3rd ACM Symposium on Principles of Database Systems*, April 1984, 290-299.

[Sto79] M. Stonebraker, Concurrency Control and Consistency of Multiple Copies in Distributed INGRES, *IEEE Transaction on Software Engineering SE-3*, 3 (May, 1979), 188-194.

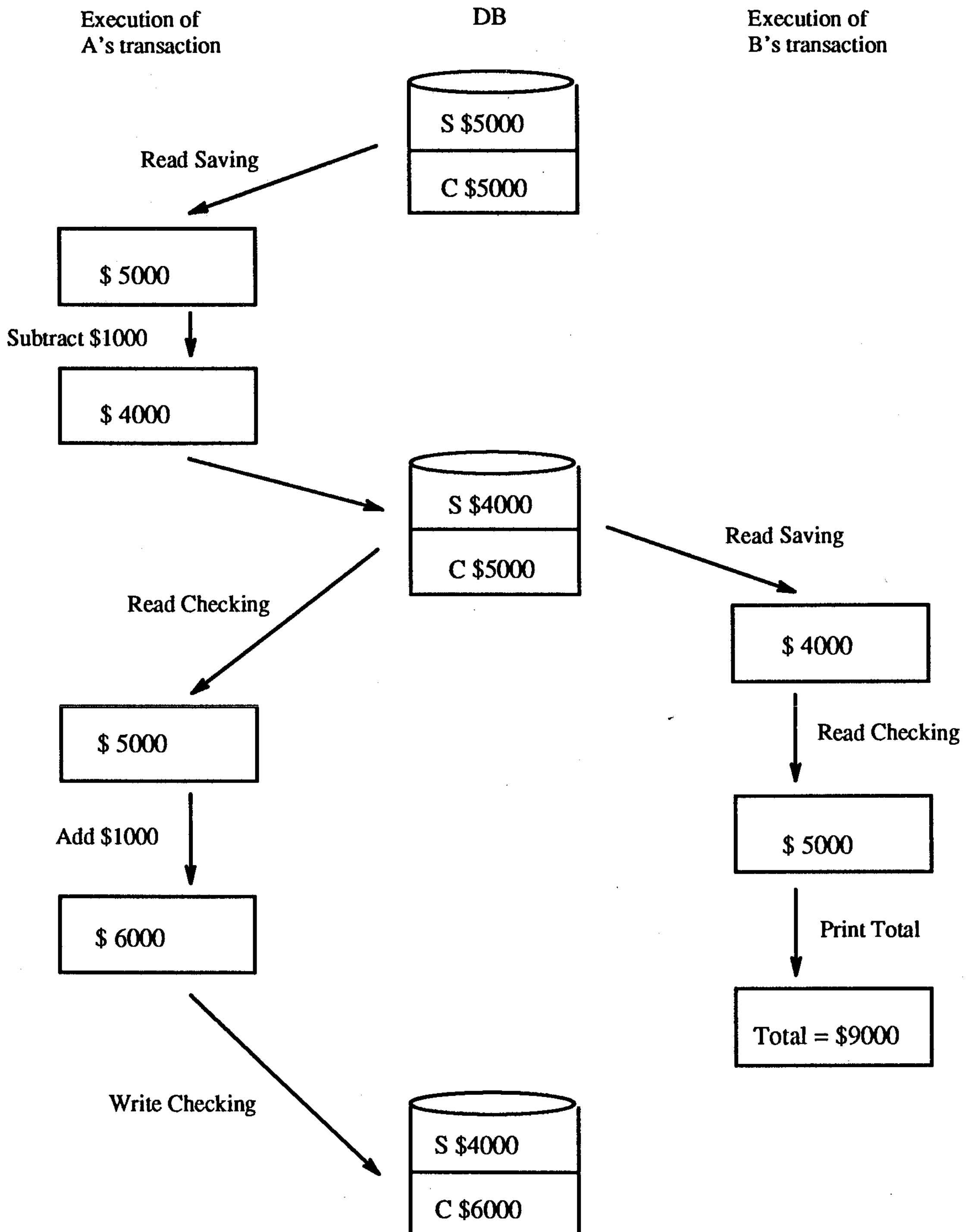
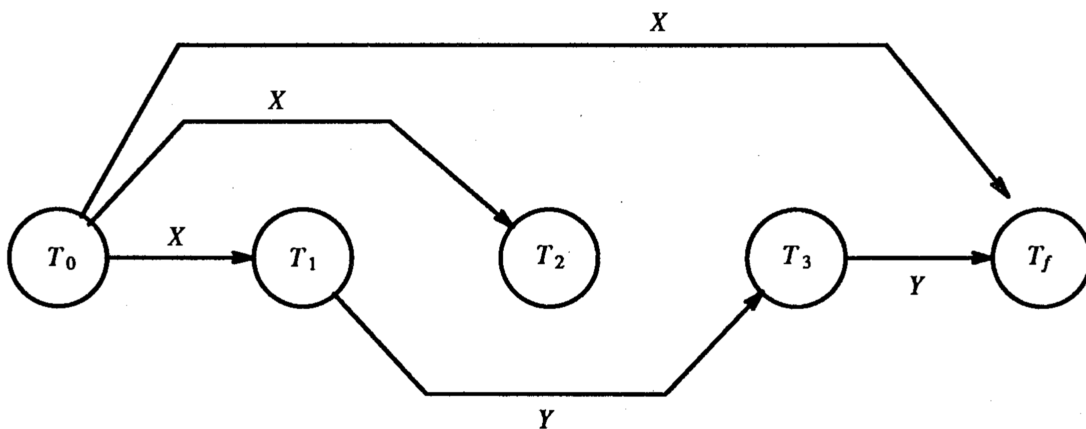
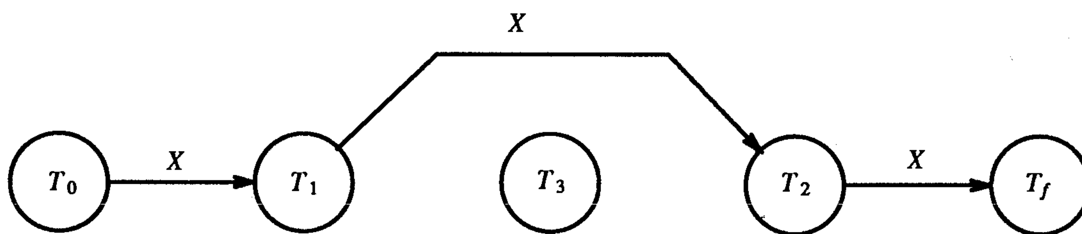


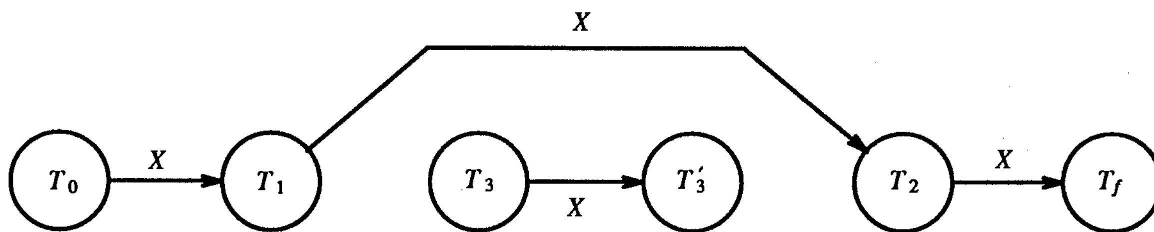
Figure 2.1.1 Illustration for Example 2.1.1.



(a) $TRF(L)$.

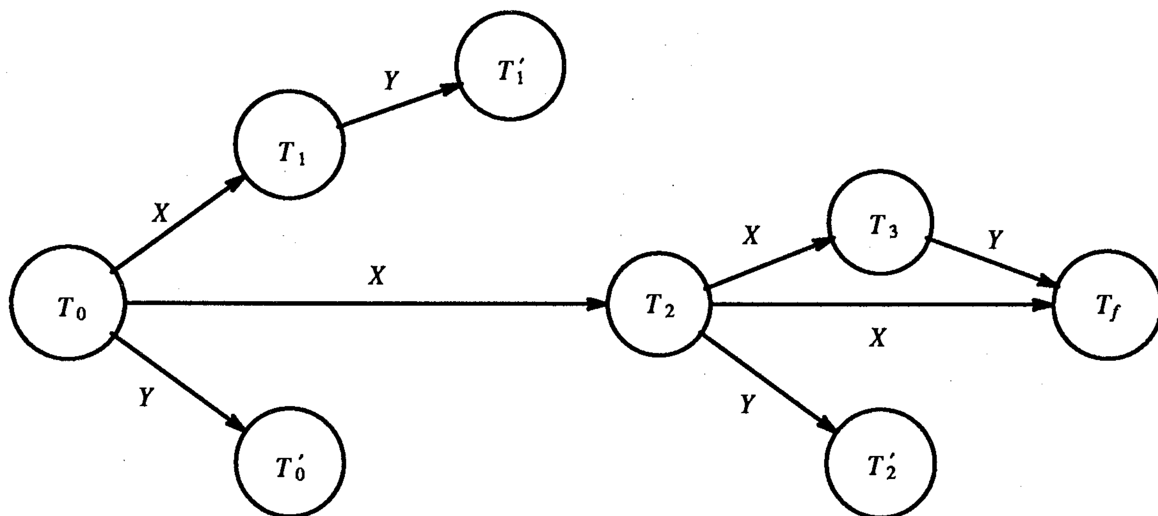


(b) $TRF(L_3)$.

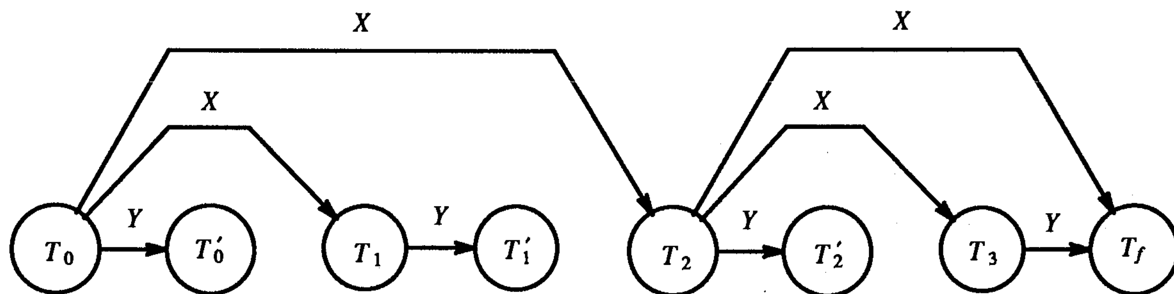


(c) Modified $TRF(L_3)$.

Figure 2.2.1 Transaction Read-from graphs.



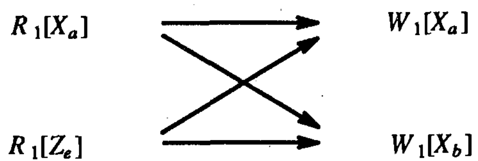
(a) *TIO* (L_2).



(b) DITS for *TIO* (L_2).

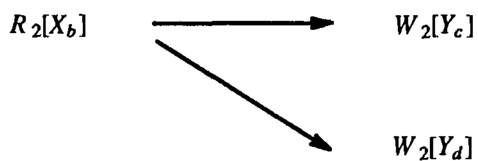
Figure 2.3.1 TIO graph and DITS.

$$T_1 = R_1[X]\, R_1[Z]\, W_1[X]$$



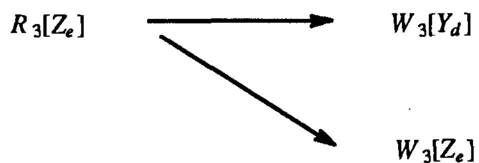
(a)

$$T_2 = R_2[X]\, W_2[Y]$$



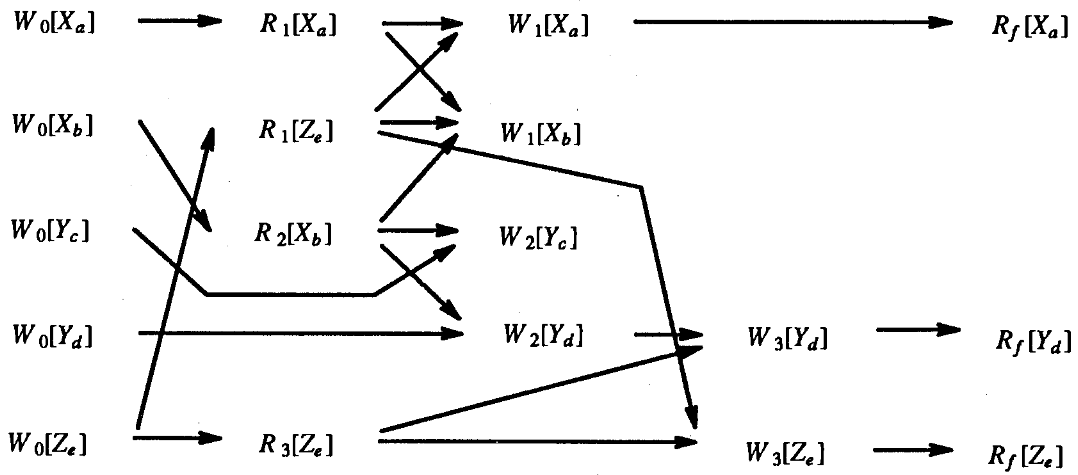
(b)

$$T_3 = R_3[Z]\, W_3[Y]\, W_3[Z]$$



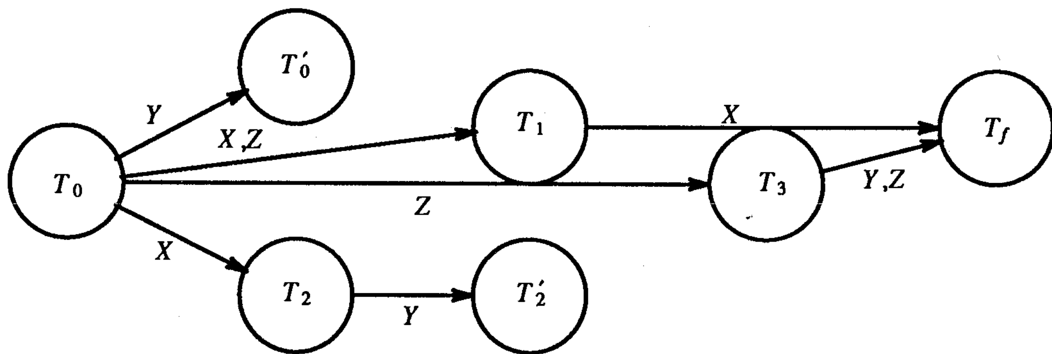
(c)

Figure 2.4.1 Translation of transactions into posets of physical operations.

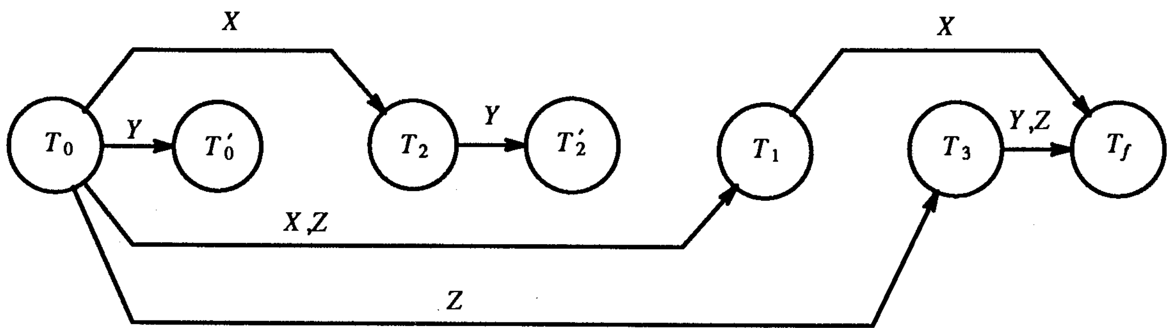


Rp log L

Figure 2.4.2 An rp log.

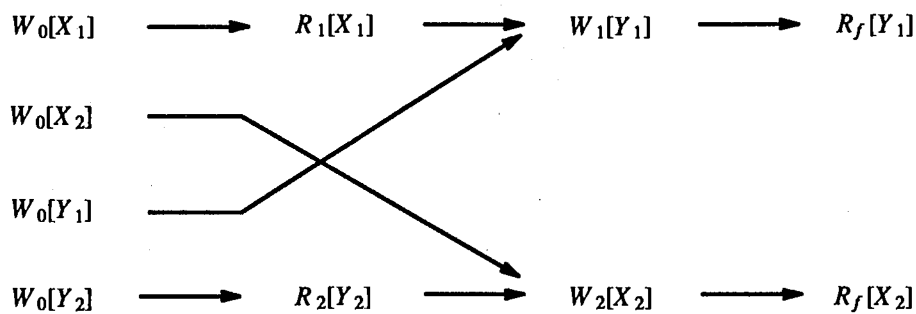


(a) *TIO* (L).

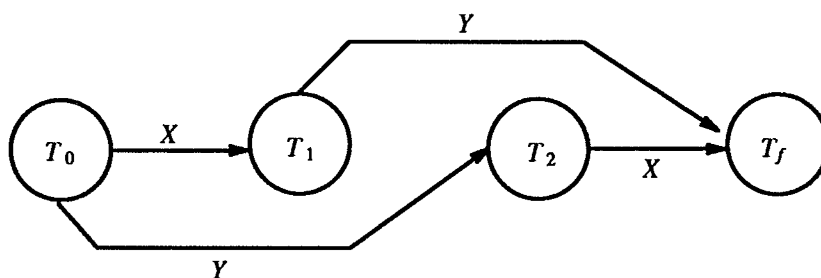


(b) DITS of *TIO* (L).

Figure 2.4.3 TIO graph of a rp log L and DITS.



(a) A rp log L



(b) $TIO(L)$

Figure 3.1.1 Rp log L and $TIO(L)$.

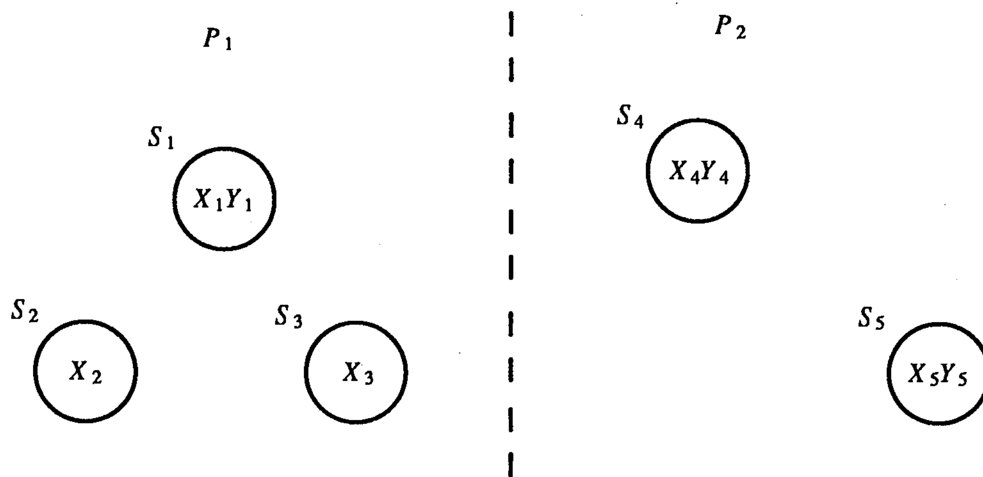
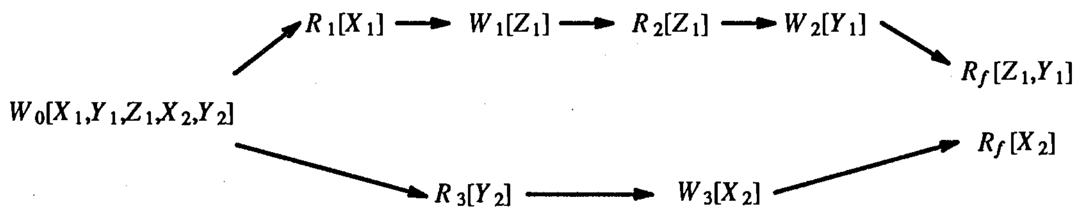
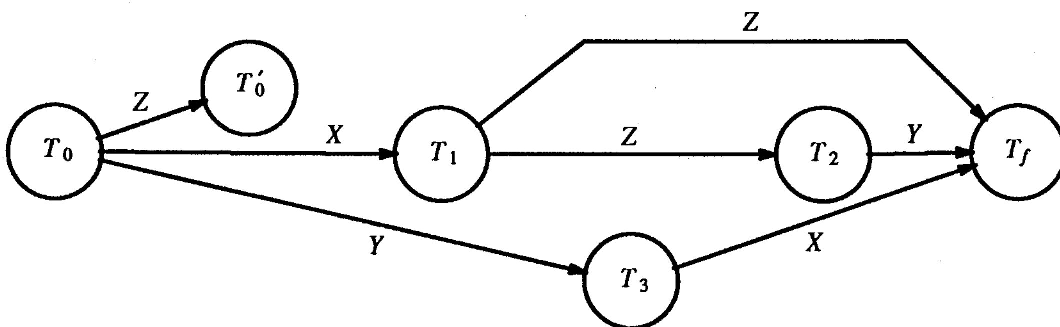


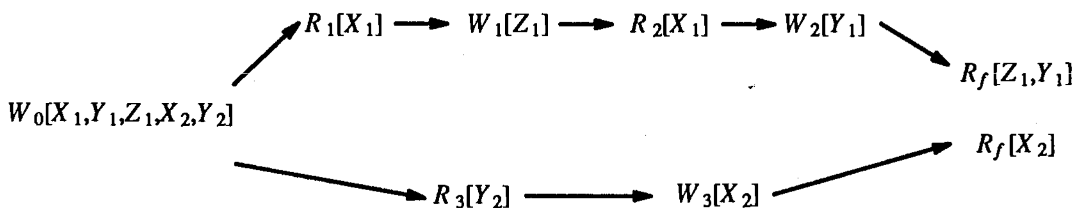
Figure 3.2.1 Illustration for Example 3.2.1.



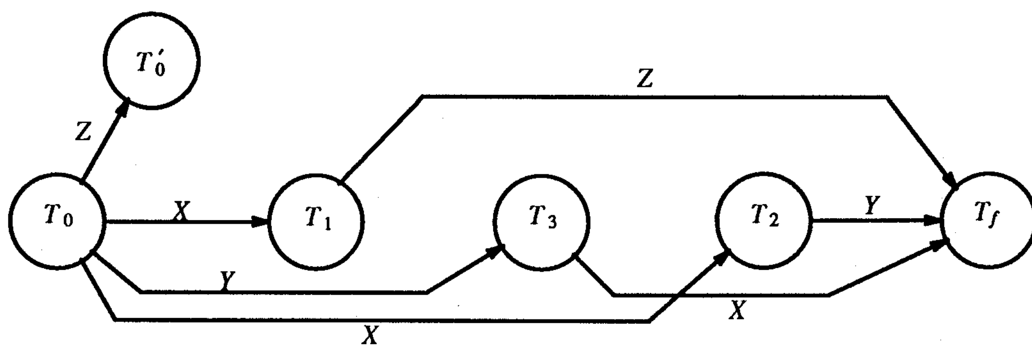
(a) Rp log L



(b) $TIO(L)$

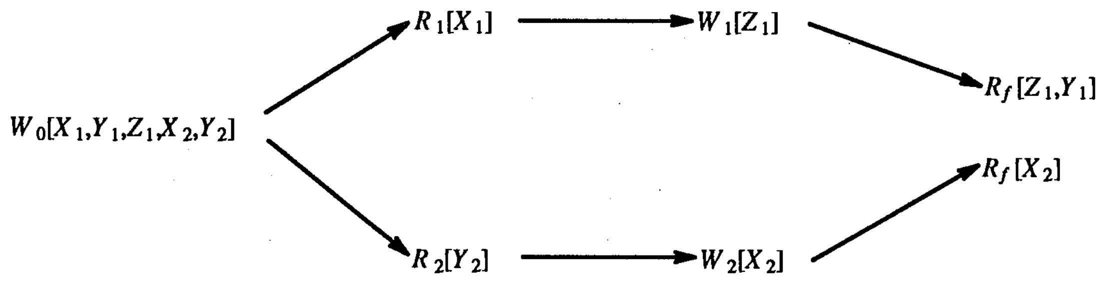


(c) Rp log L_1

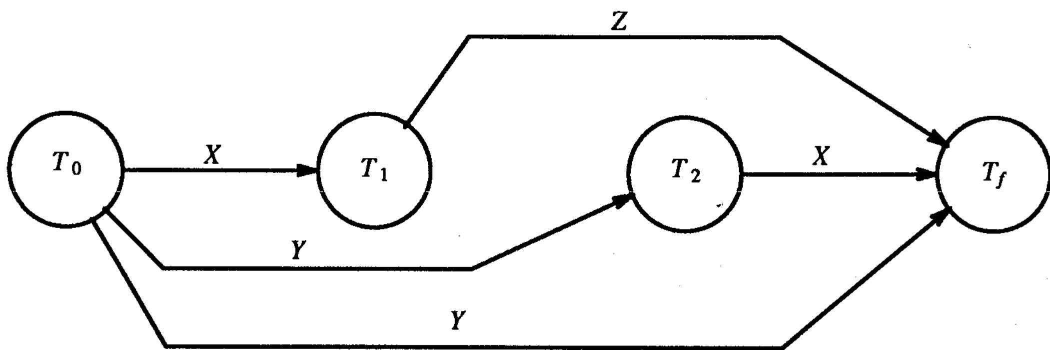


(d) $TIO(L_1)$

Figure 3.3.1 Illustration for Example 3.3.1.

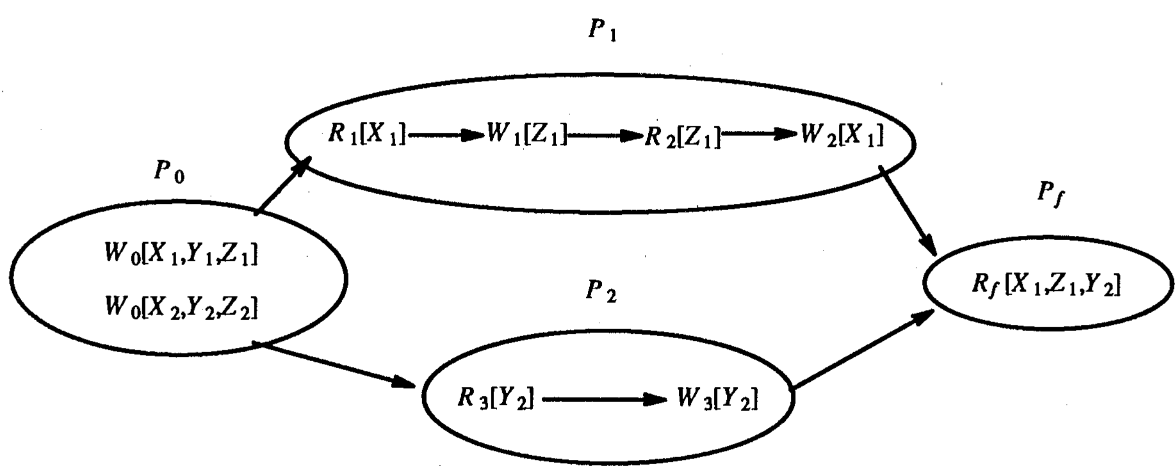


(e) Rp log L_3

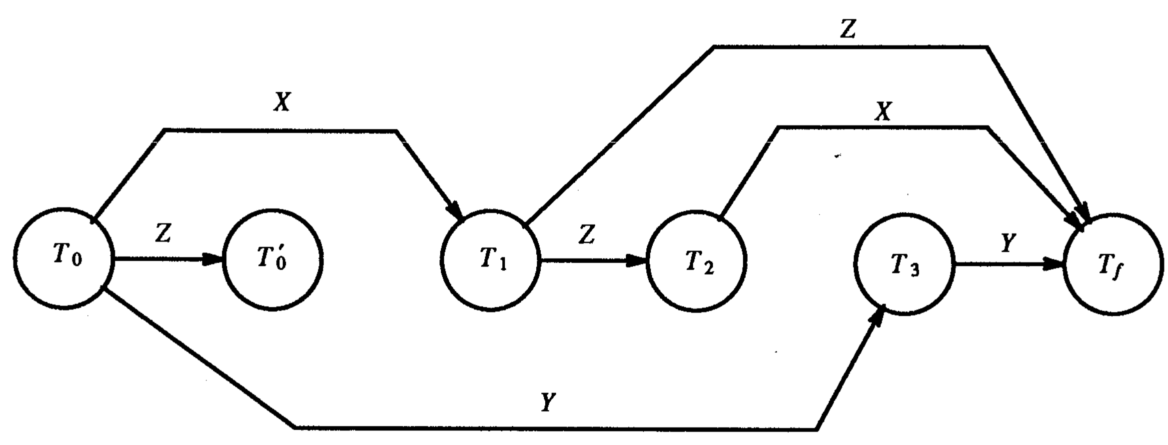


(f) $TIO(L_3)$

Figure 3.3.1 Illustration for Example 3.3.1.

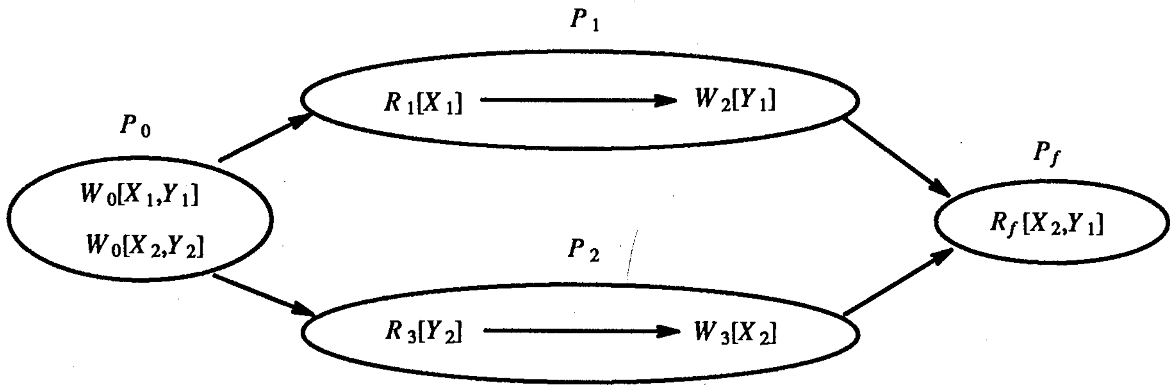


(a) Rp log L

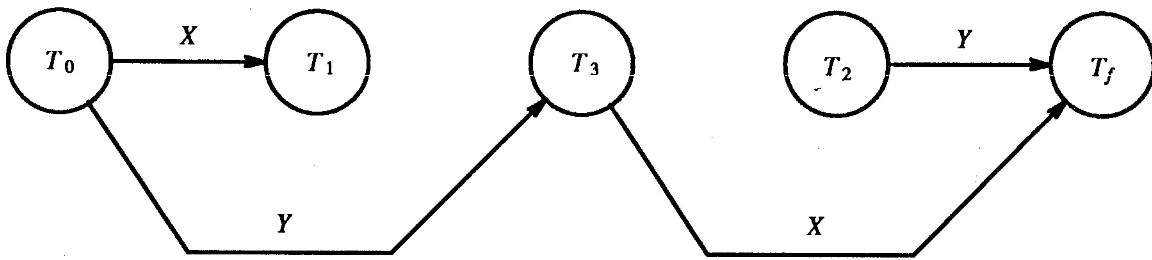


(b) $TIO(L)$

Figure 3.4.1 Illustration for Example 3.4.1.

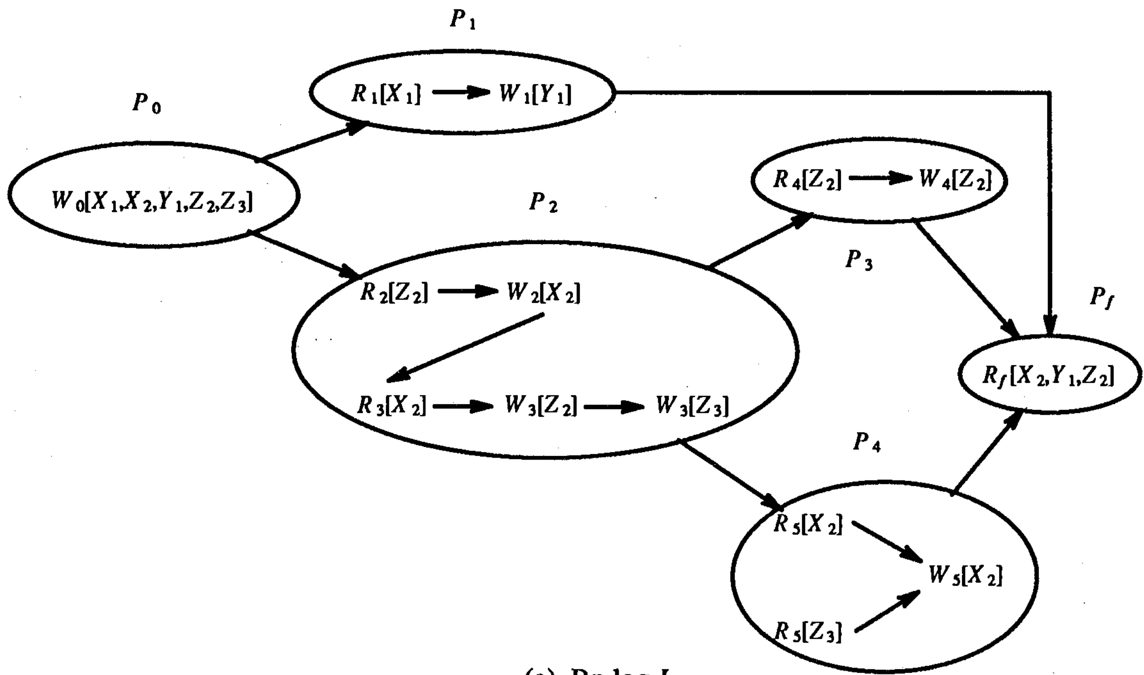


(a) Rp log L'

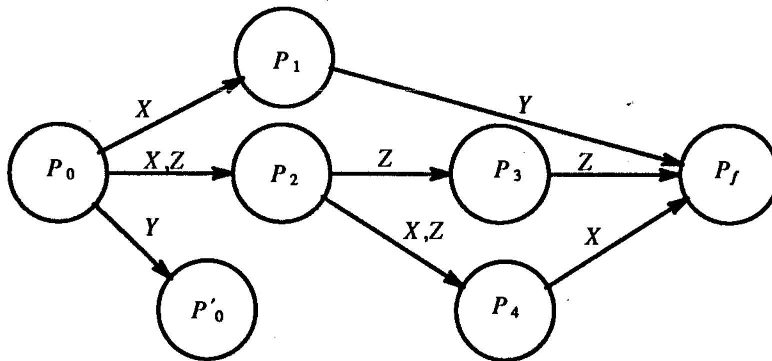


(b) *TIO* (L')

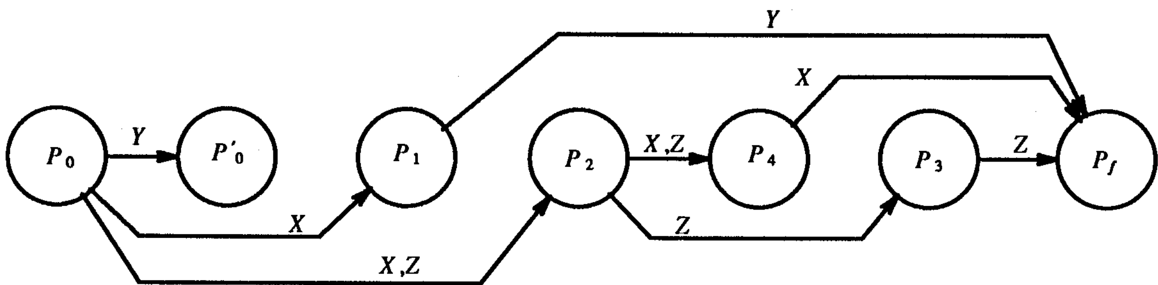
Figure 3.4.2 Illustration for Example 3.4.2.



(a) Rp log L

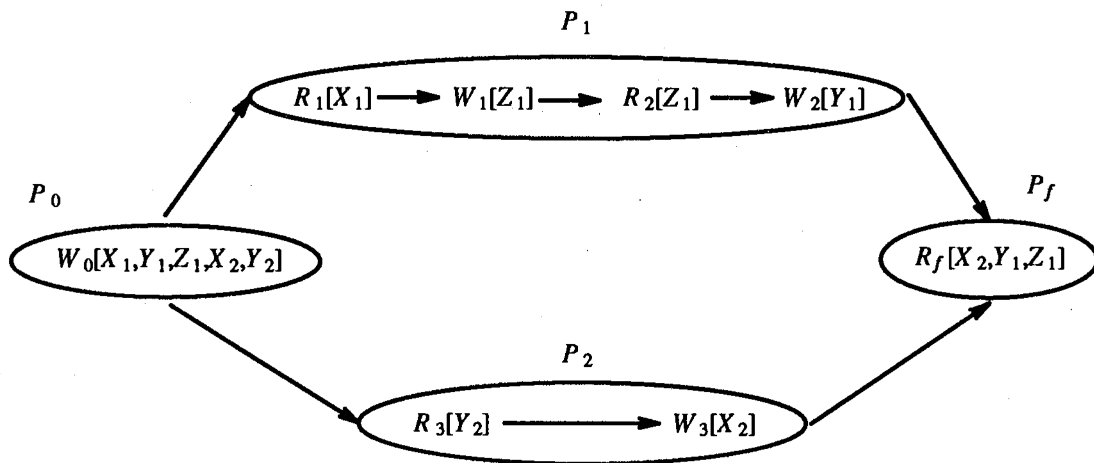


(b) *PIO* (L)

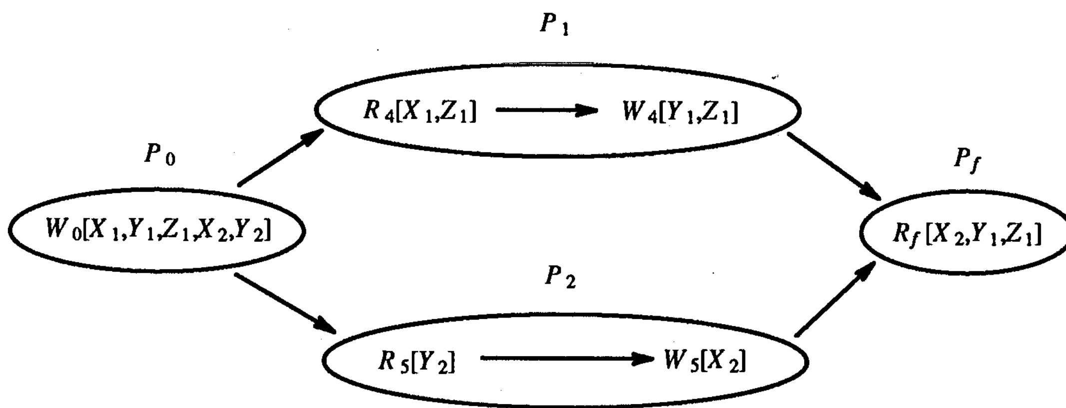


(c) DITS of *PIO* (L)

Figure 3.5.1 Illustration for Examples 3.5.1 and 3.5.2.

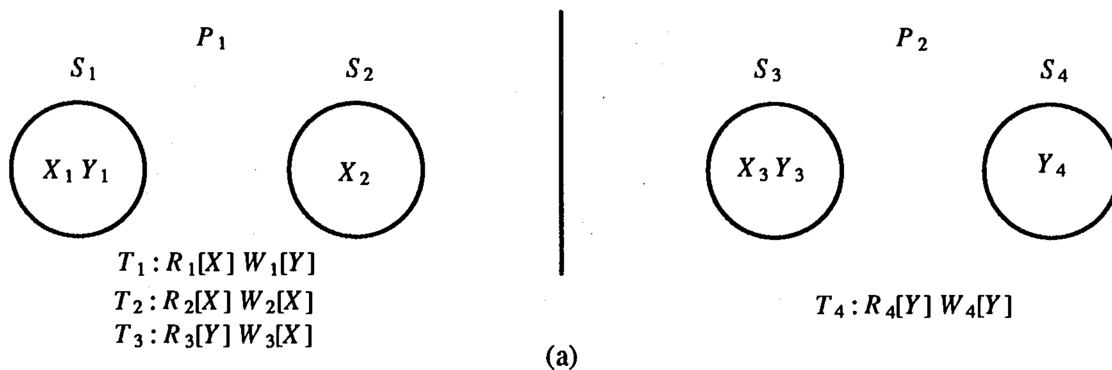


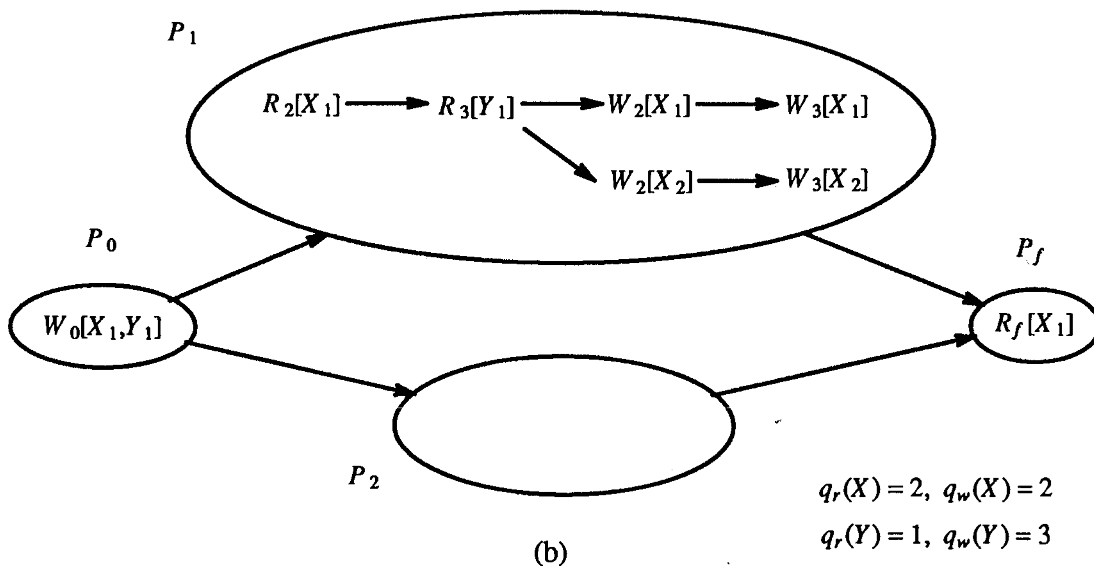
(a) Rp log L



(b) Rp log L_1

Figure 3.5.2 An illustration for the Non-Selective Assumption.





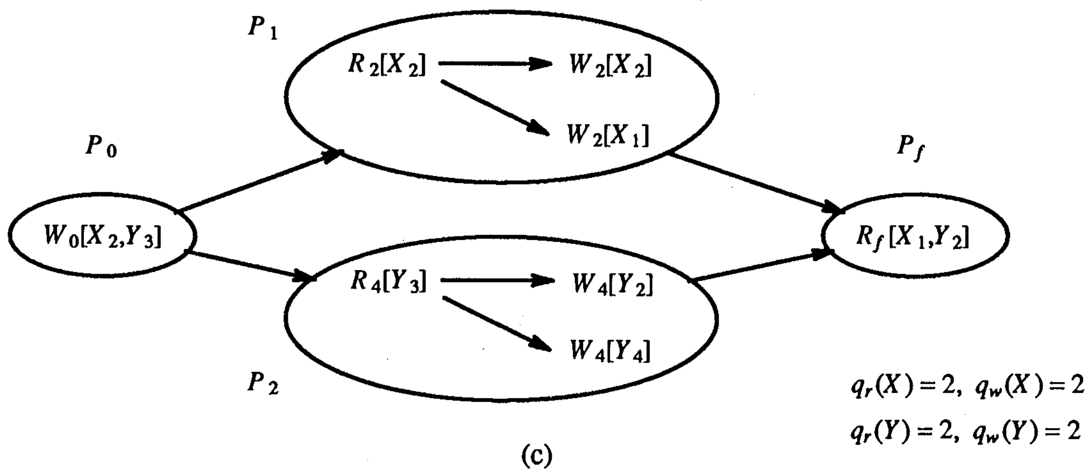
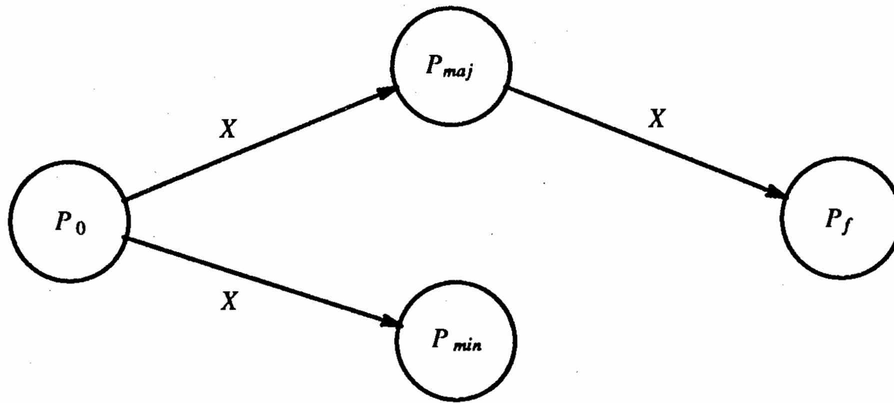
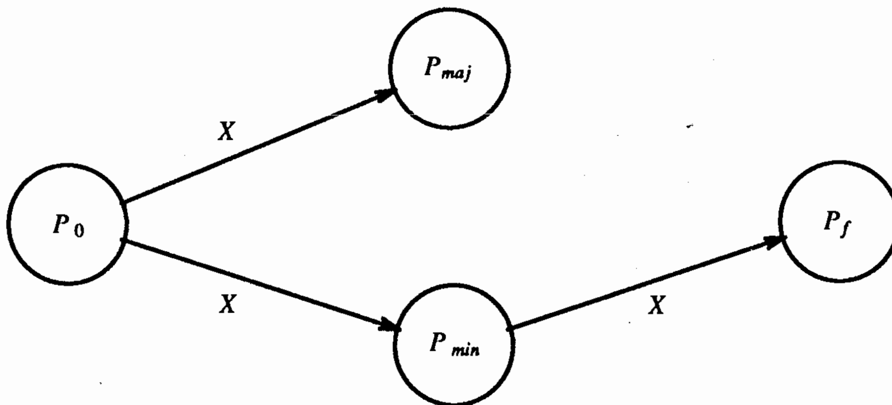


Figure 4.1.1 Acceptance Ratio.

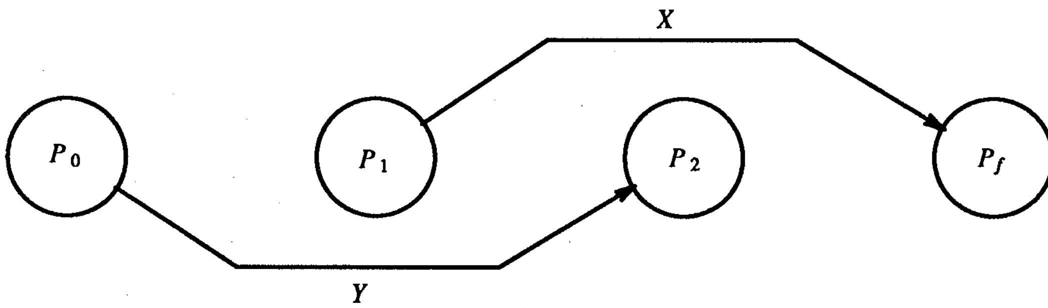


(a)

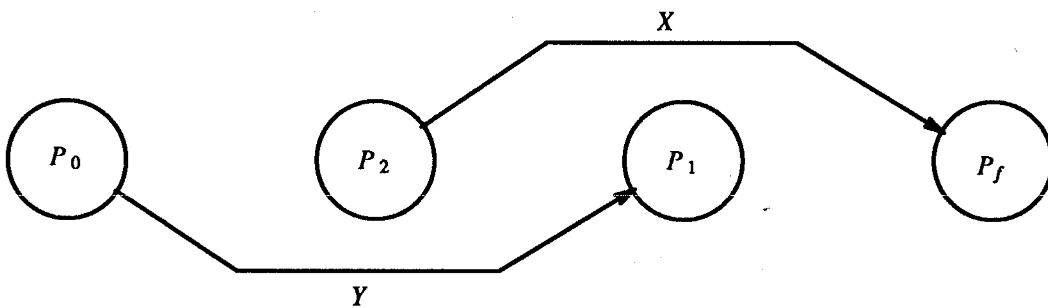


(b)

Figure 4.2.1 Two PIO graphs.



(a) If $P_0P_1P_2P_f$ is a DITS of $PIO(L)$, $X \neq Y$.



(b) If $P_0P_2P_1P_f$ is a DITS of $PIO(L)$, $X \neq Y$.

Figure 4.3.1 Two DITS's for $PIO(L)$.

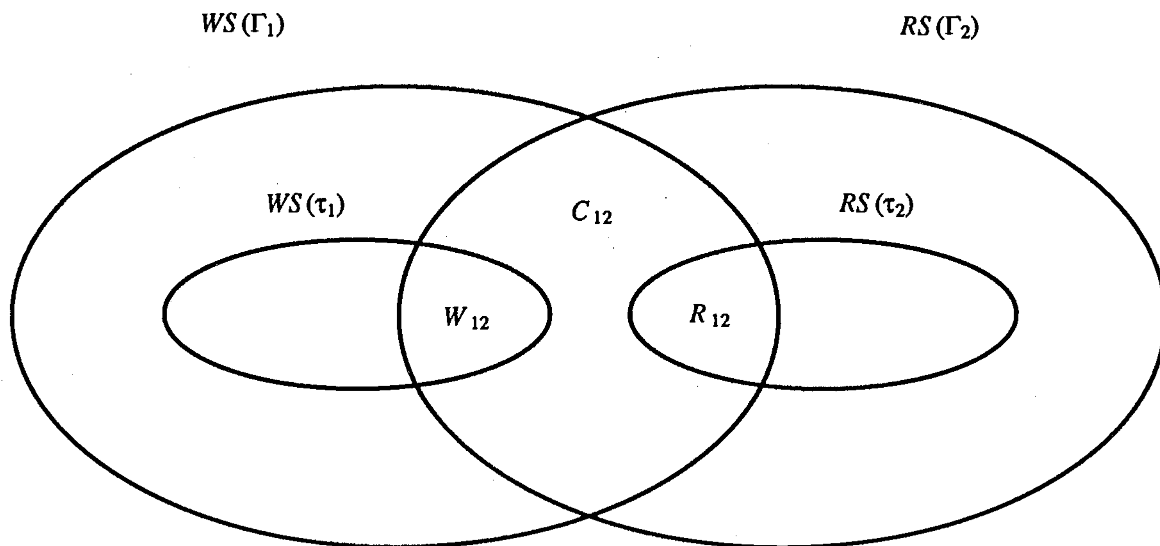


Figure 4.3.2 Inclusion relationships.

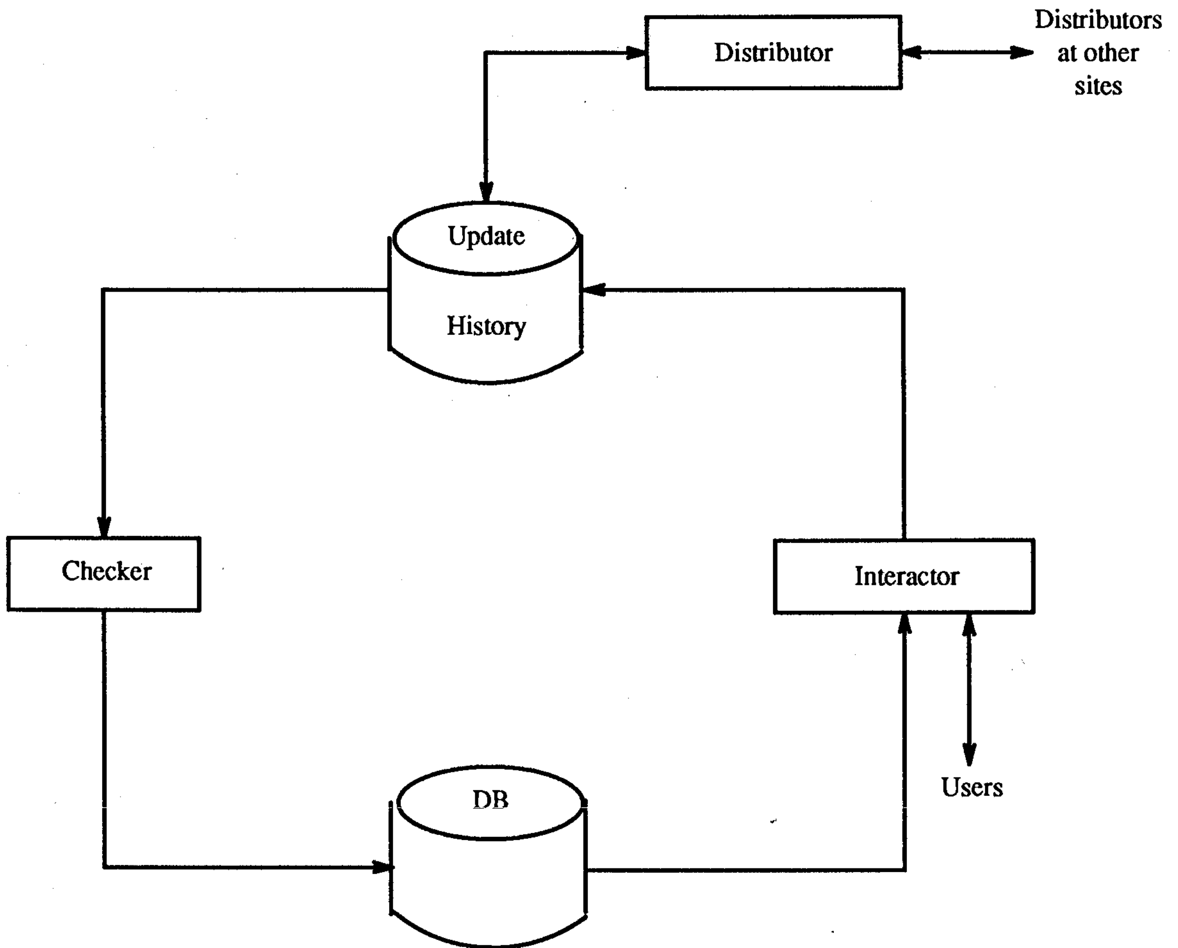


Figure 5.2.1 Architecture of SHARD.

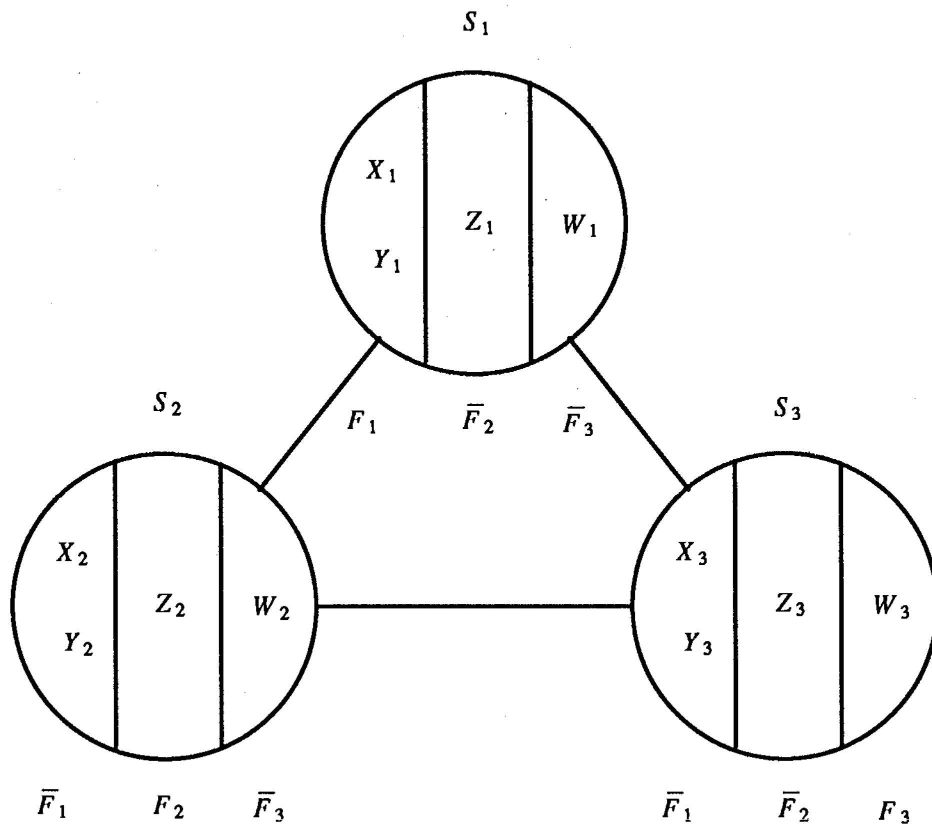
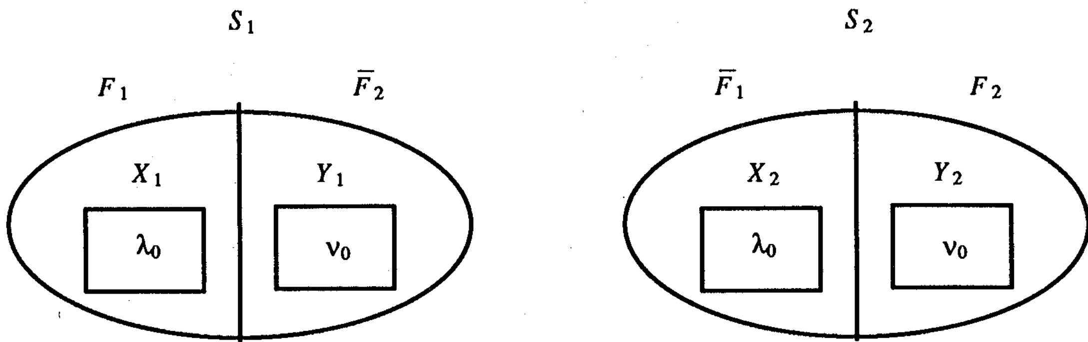


Figure 5.3.1 A fragmented database system.

figure



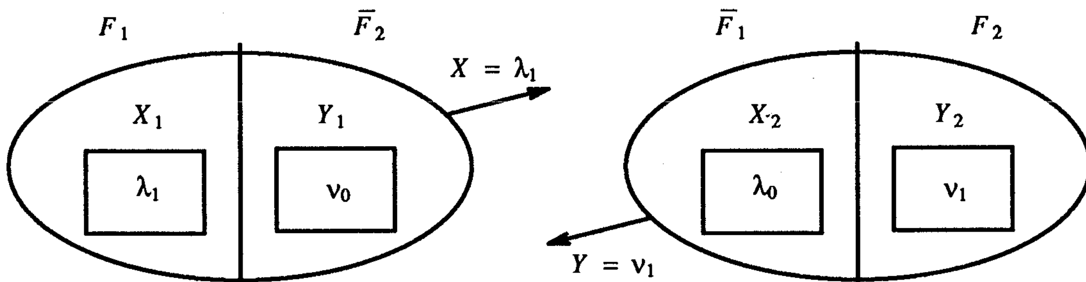
(a)

$T_1 = R_1[X]\,W_1[X]$

(X_1 is updated to λ_1)

$T_2 = R_2[Y]\,W_2[Y]$

(Y_2 is updated to ν_1)



$T_3 = R_3[X]\,R_3[Y]$

values read by T_3 :

$X = \lambda_1$

$Y = \nu_0$

$T_4 = R_4[X]\,R_4[Y]$

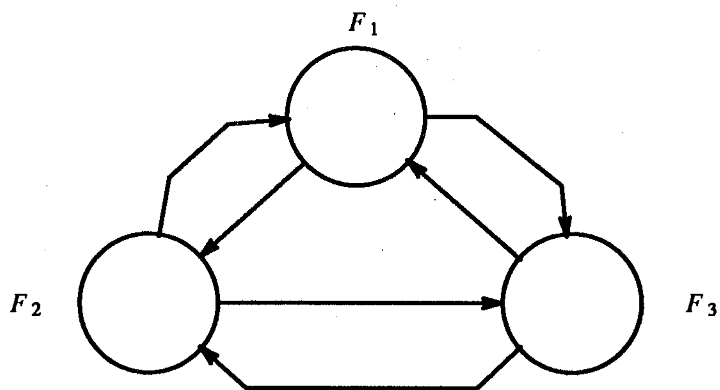
values read by T_4 :

$X = \lambda_0$

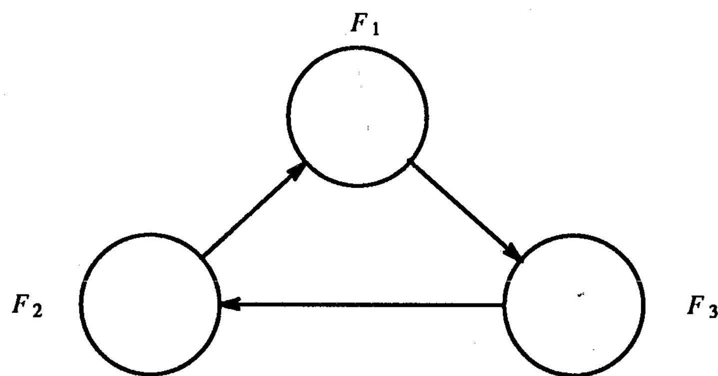
$Y = \nu_1$

(b)

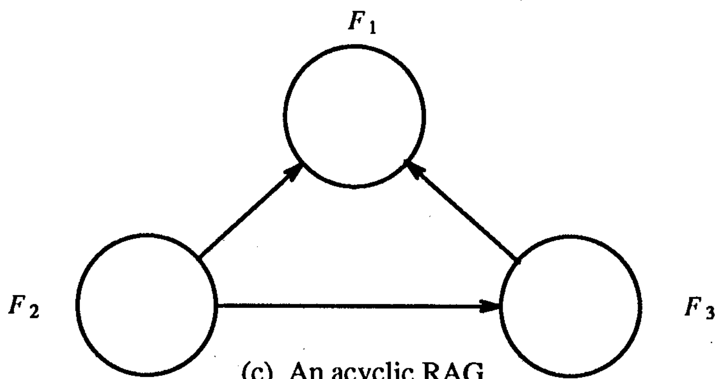
Figure 5.4.1 Illustration for Example 5.4.1.



(a) A complete RAG

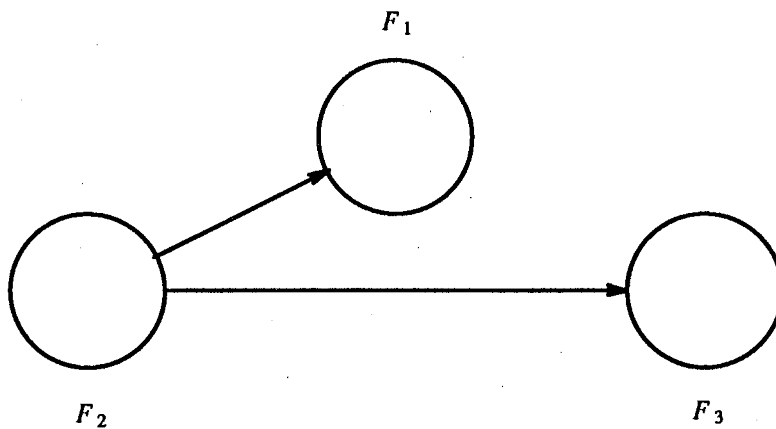


(b) A cyclic RAG



(c) An acyclic RAG

Figure 5.4.2(a-c) Three RAG's.



(d) Loopless RAG

Figure 5.4.2 A RAG.

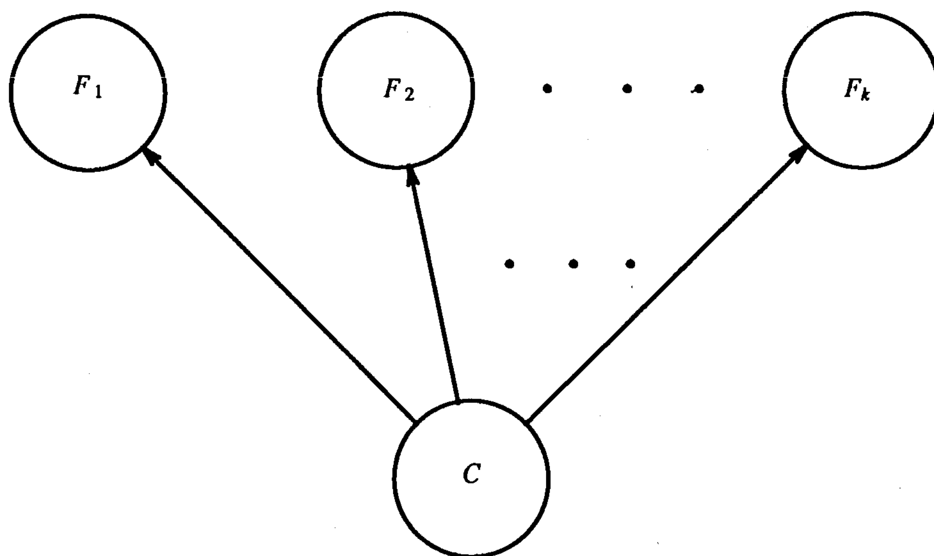
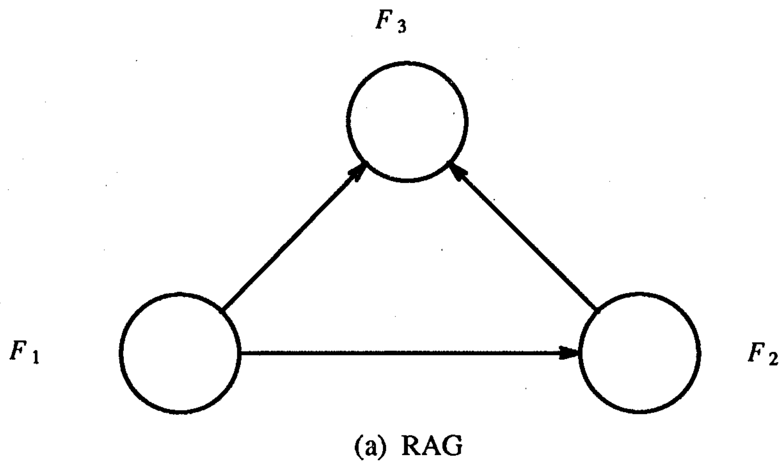
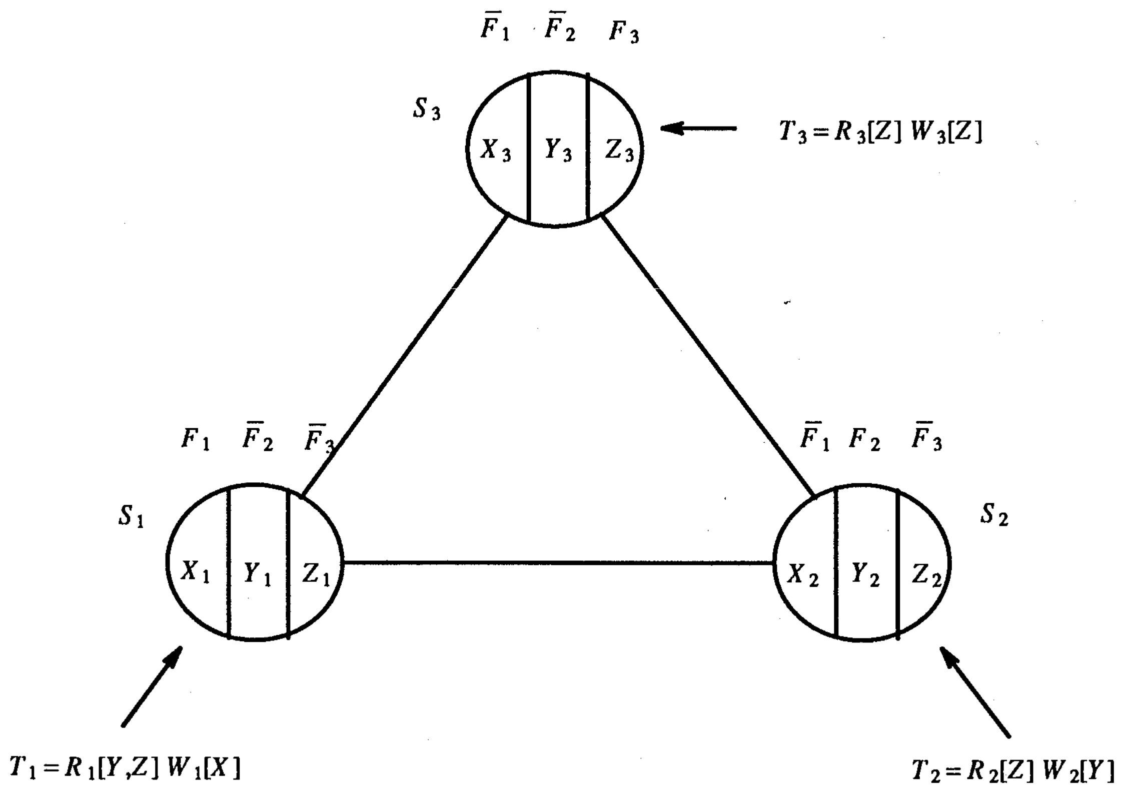


Figure 5.4.3 RAG of Example 5.4.3.

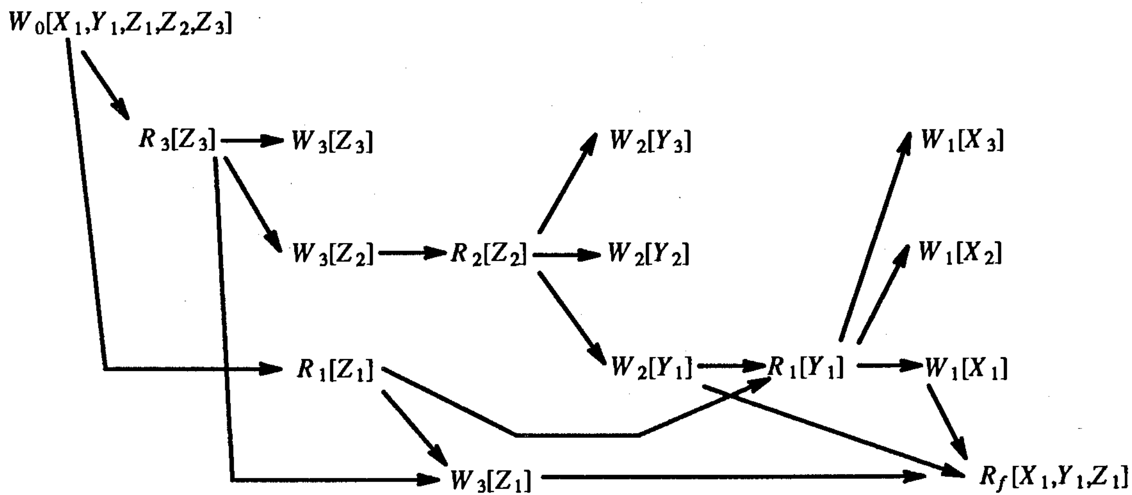


(a) RAG

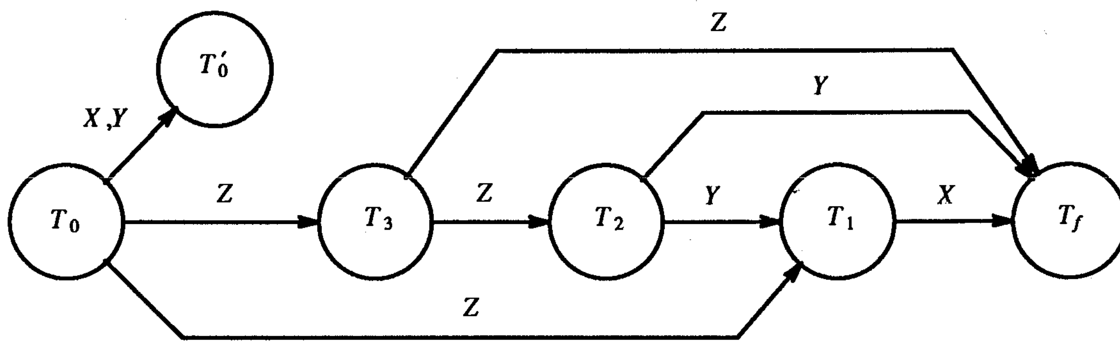


(b) An execution in a fragmented database

Figure 5.4.4(a-b) Illustration for Example 5.4.4.



(c) Rp log L



(d) TIO (L)

Figure 5.4.4(c-d) Illustration for Example 5.4.4.

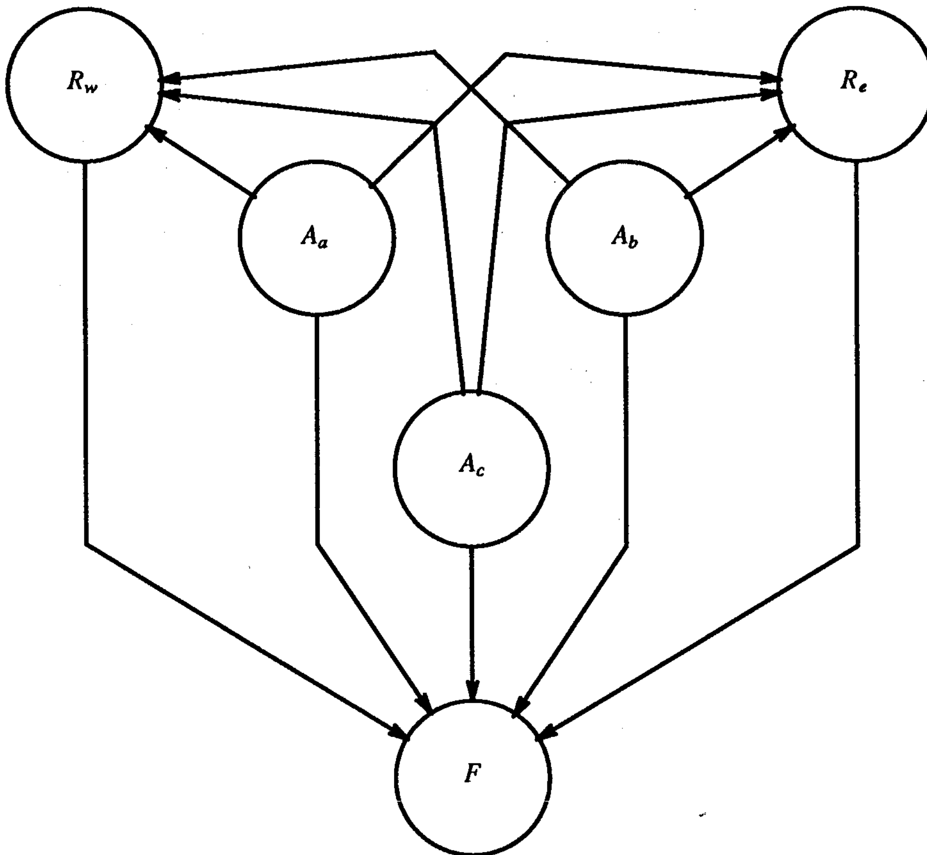


Figure 5.4.5 RAG for a fragmented database.

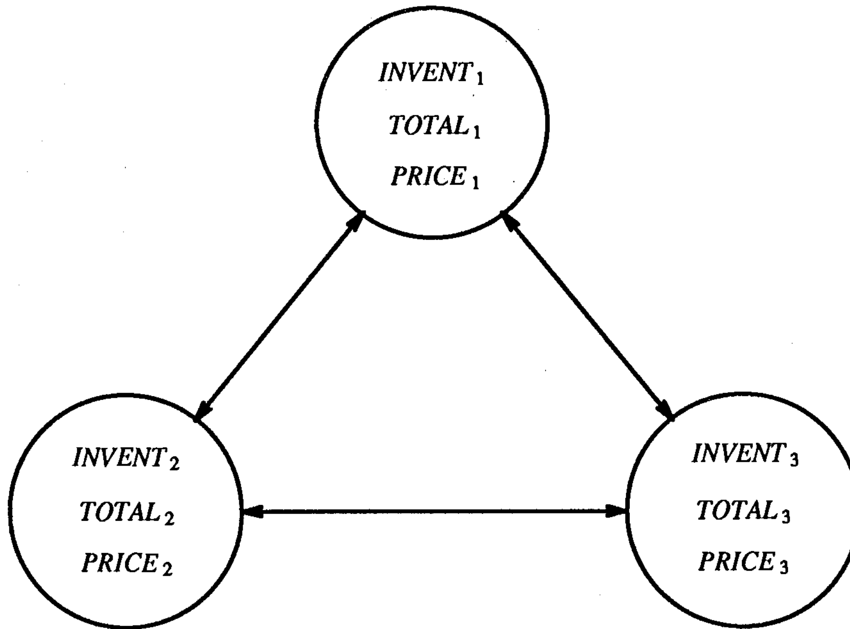
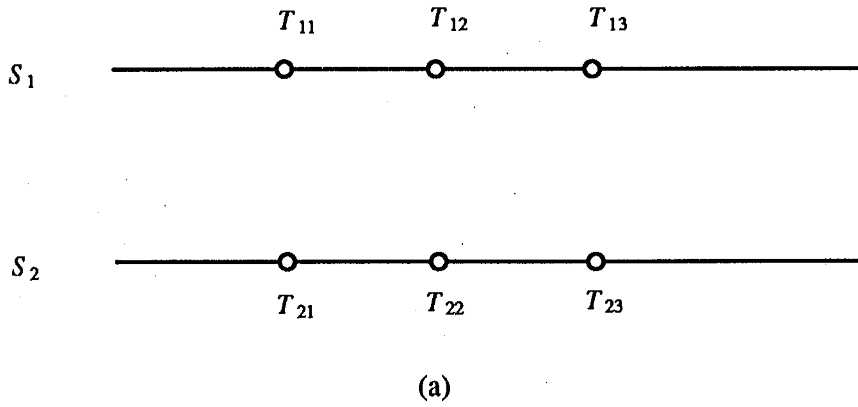
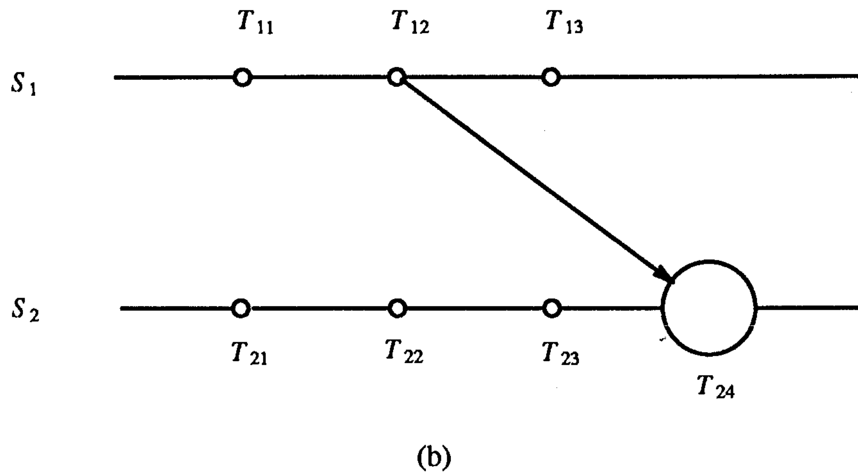


Figure 6.1.1 Illustration for Example 6.1.1.



(a)



(b)

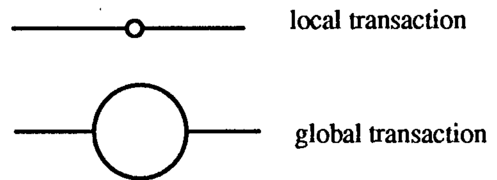
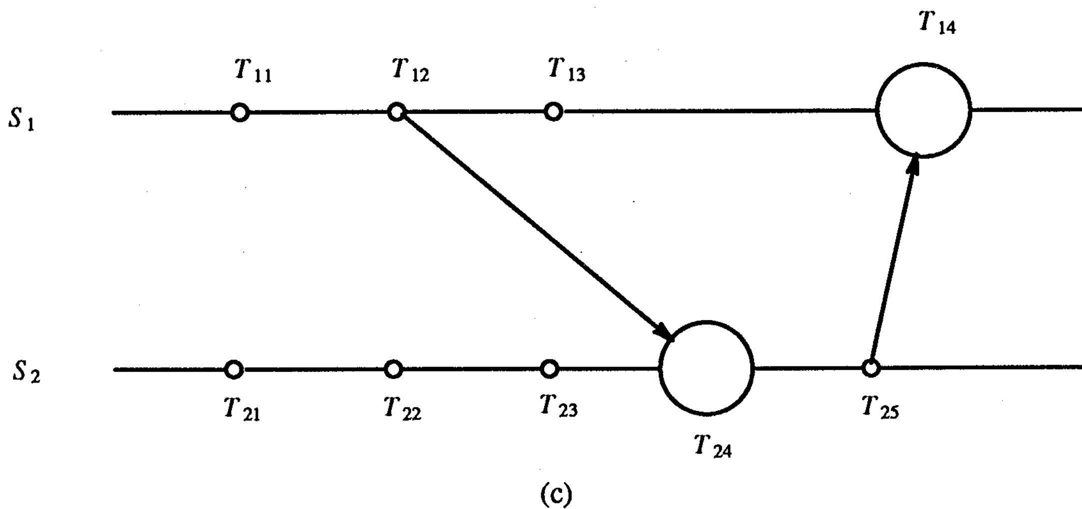
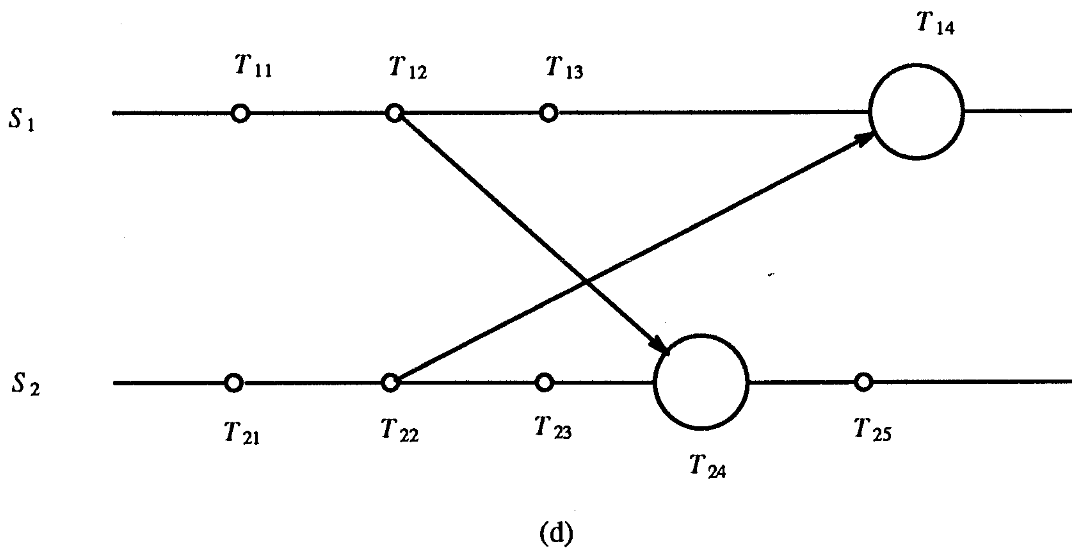


Figure 6.1.2 (a-b) Illustration for Example 6.1.2.



(c)



(d)

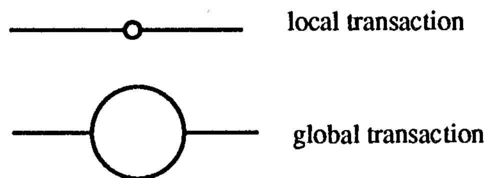


Figure 6.1.2 (c-d) Illustration for Example 6.1.2.

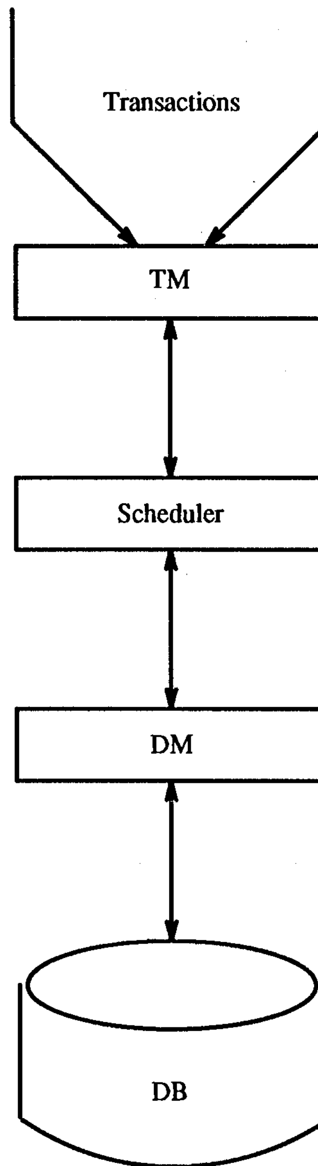


Figure 6.2.1 An architecture for a single-site database.

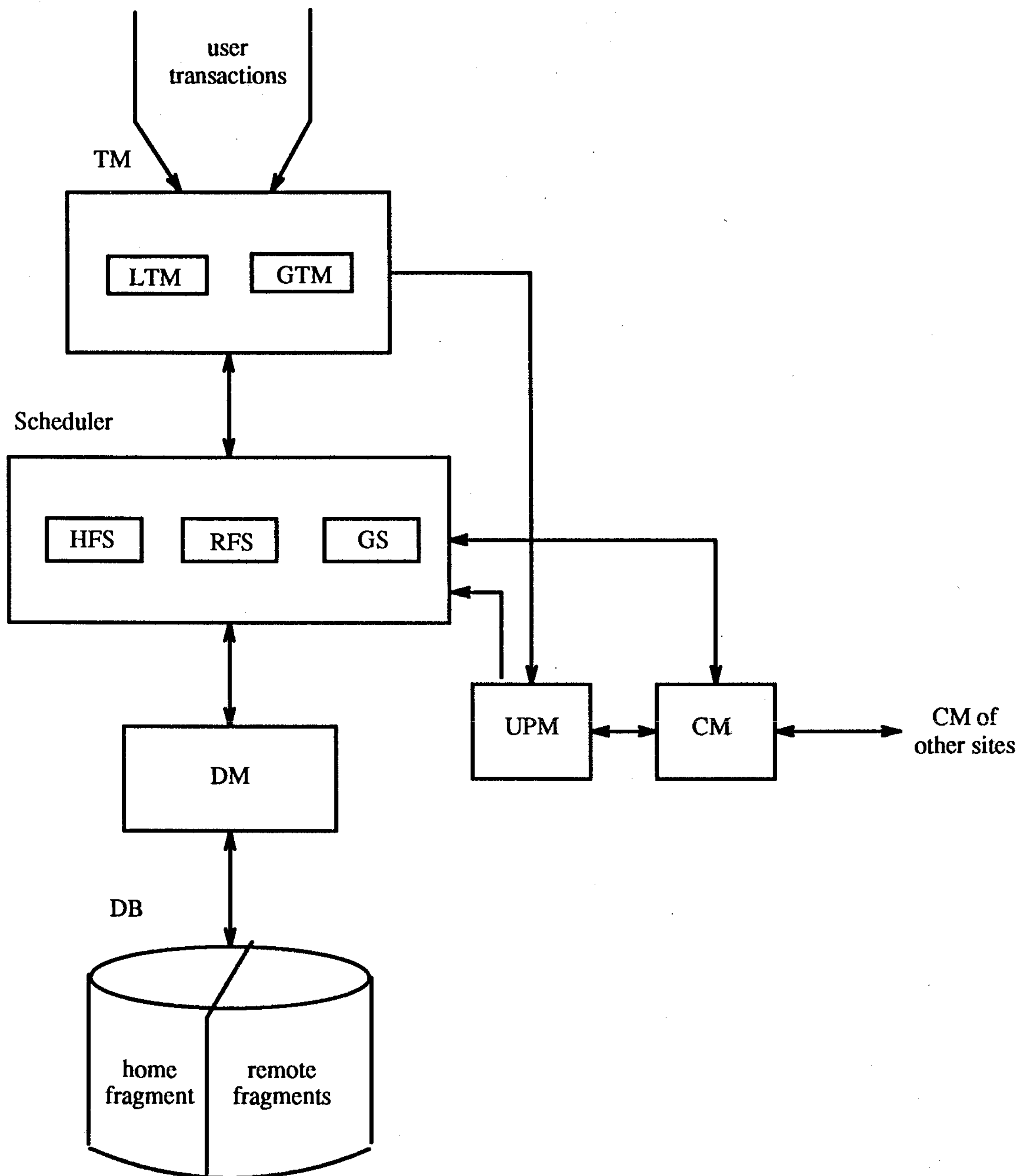
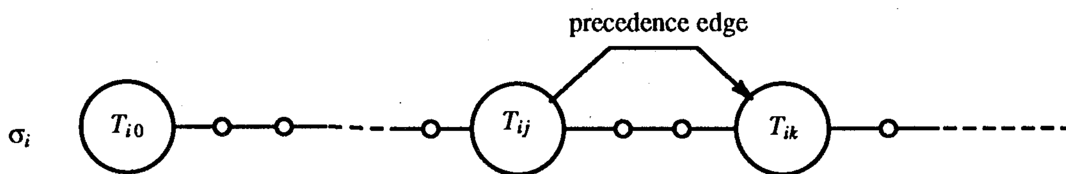
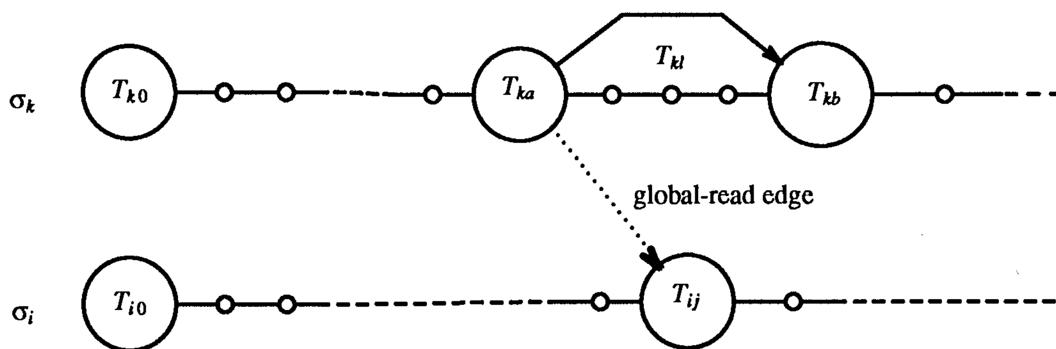


Figure 6.2.2 An architecture for GTOC.



(a) Precedence edge.



(b) Global-read edge.

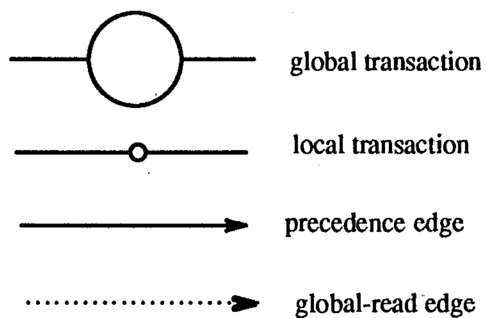
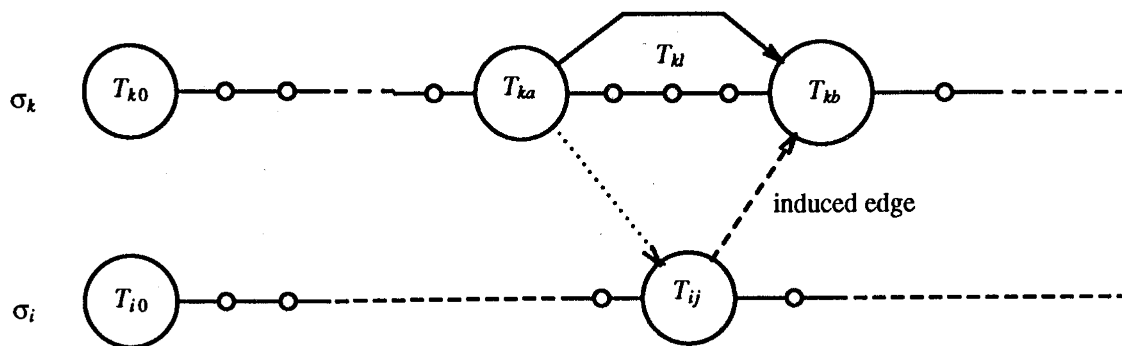
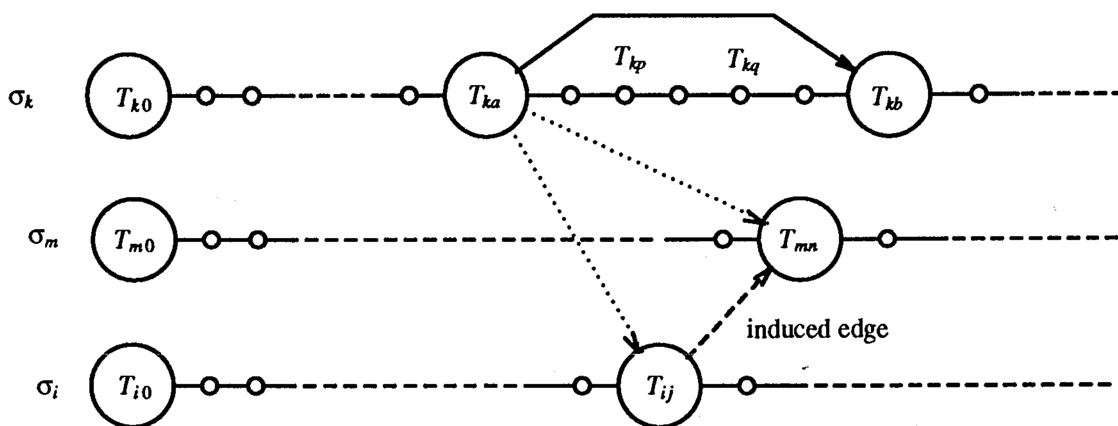


Figure 6.3.1 (a-b) Precedence edge and global-read edge.



(c) Induced edge.



(d) Induced edge.

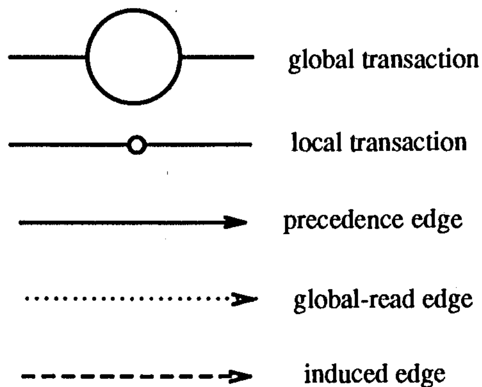


Figure 6.3.1 (c-d) Two kinds of induced edges.

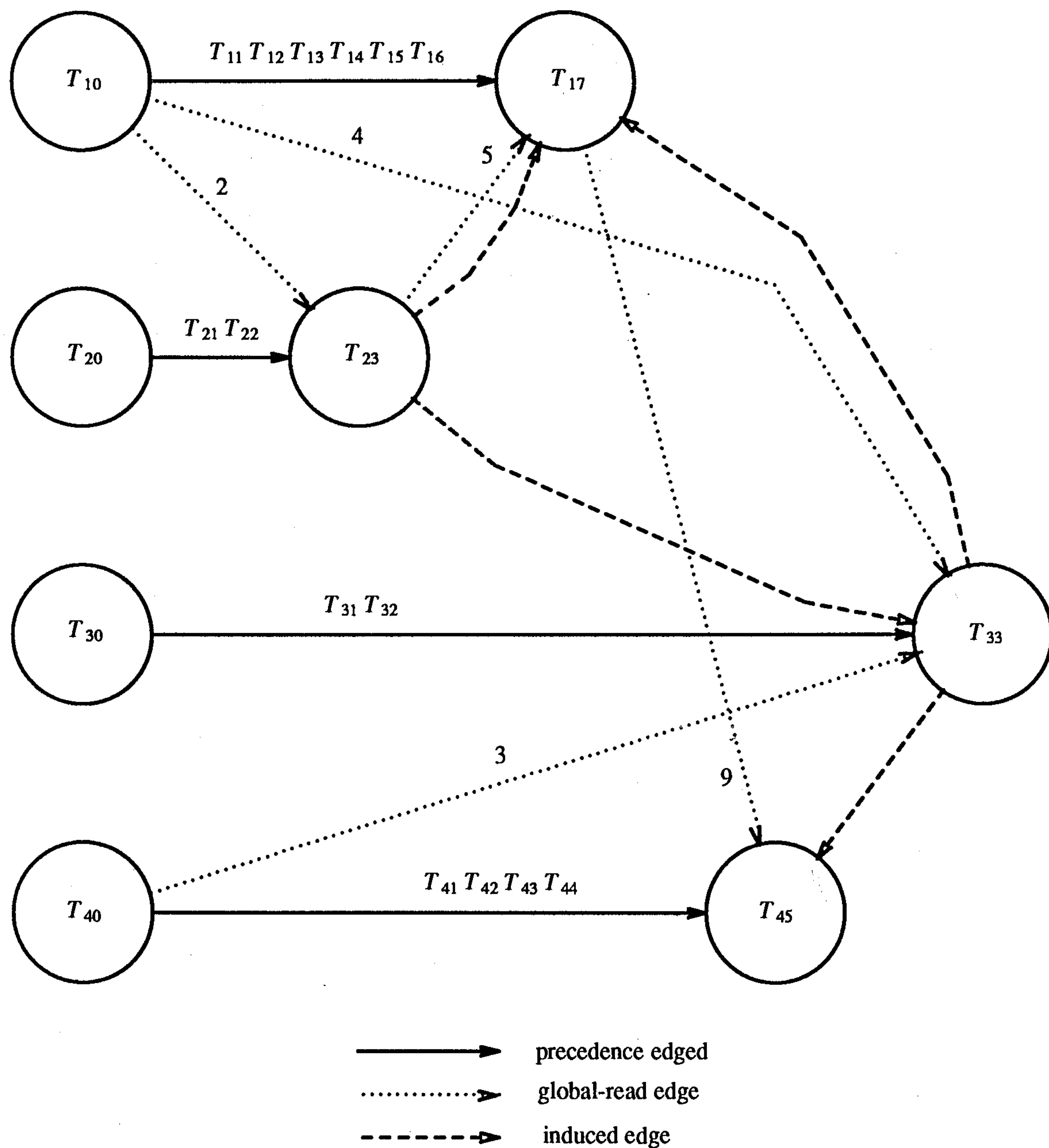


Figure 6.3.2 A GOS graph.

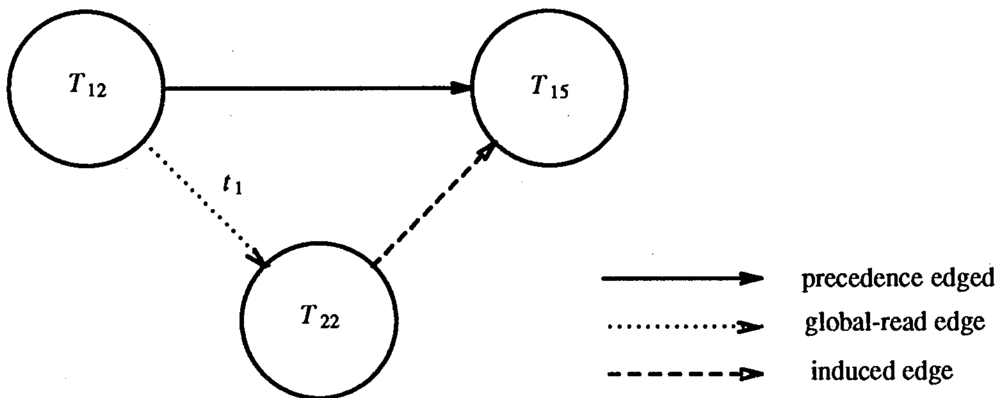


Figure 6.4.1 GOS graph of T_{11}, T_{22} and T_{15}.

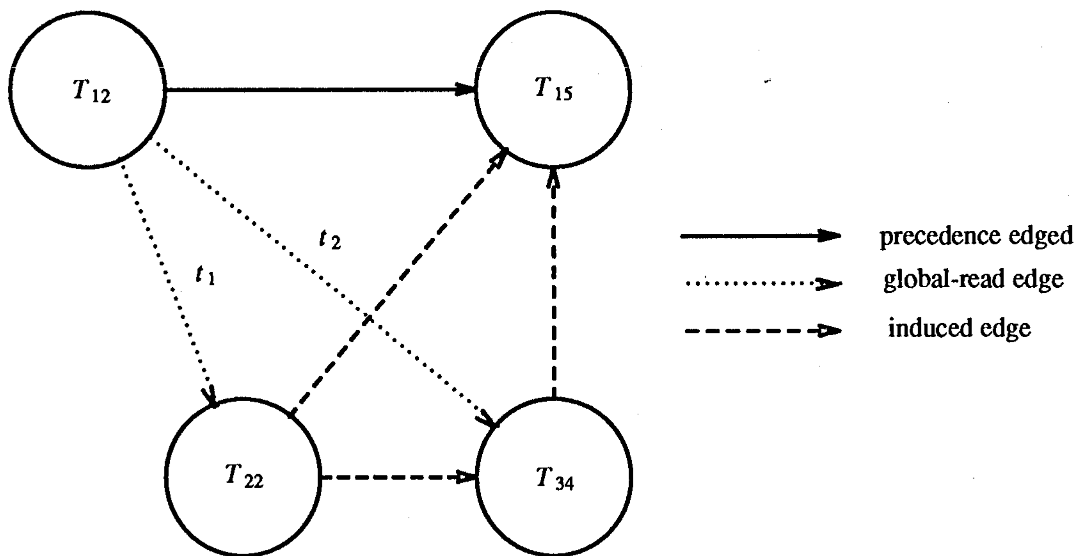
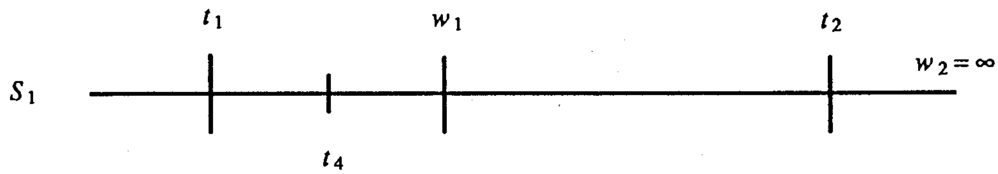


Figure 6.4.2 GOS graph of T_{12}, T_{15}, T_{22} and T_{34}.



$[t_1, t_2]$ -- safe interval of T_{4b} against $COMMIT_1$.
$[w_1, t_2]$ -- safe interval of T_{4b} against $WAIT_1$.
$t_4 = Dts(CR_1(T_{4b}))$.

(a)

$[t'_1, t'_2]$ -- safe interval of T_{3a} against $COMMIT_2$.
$[t'_1, w'_2]$ -- safe interval of T_{3a} against $WAIT_2$.
$t_3 = Dts(CR_2(T_{3a}))$.

(b)

Figure 6.4.3 Deadlock.

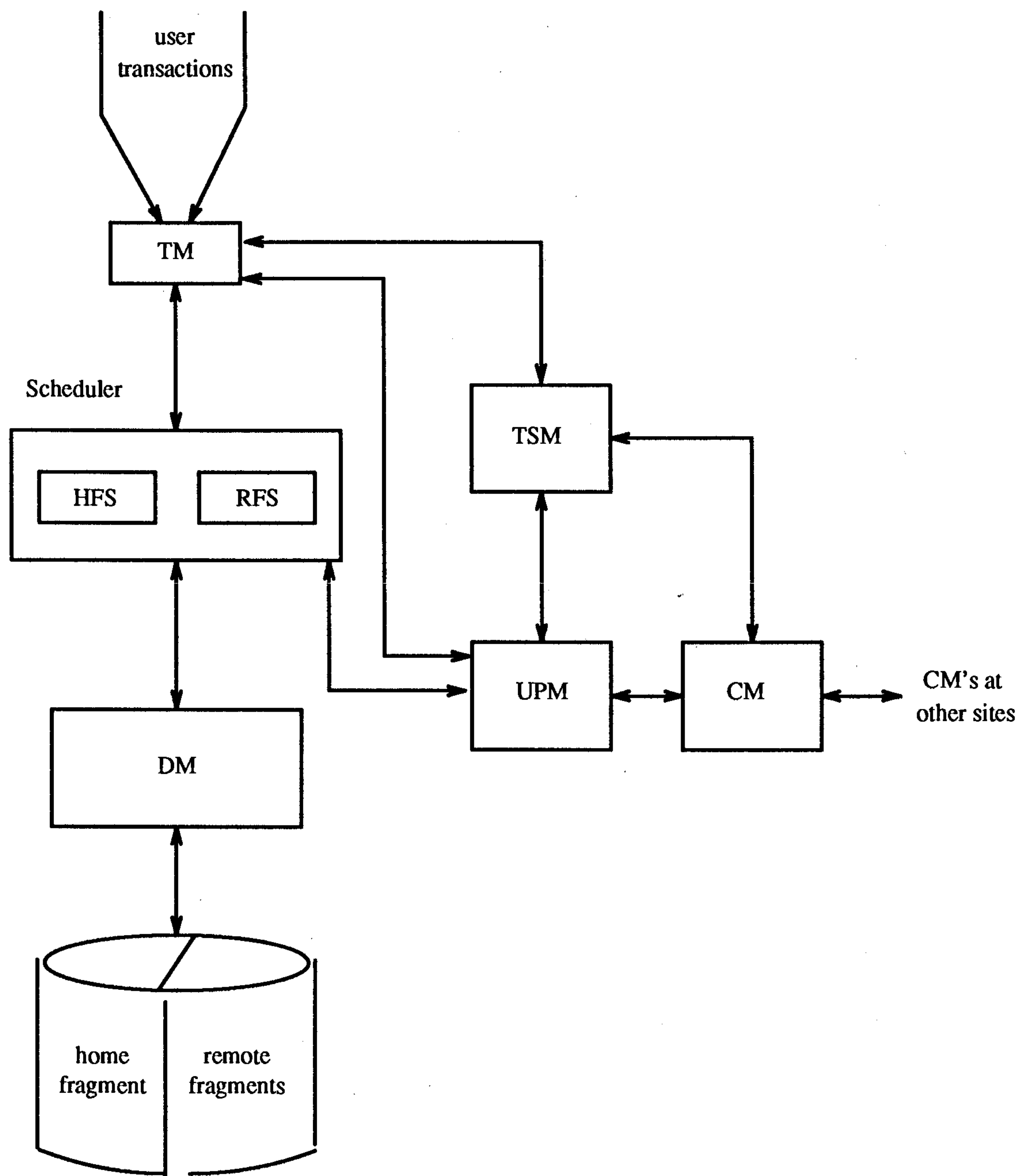
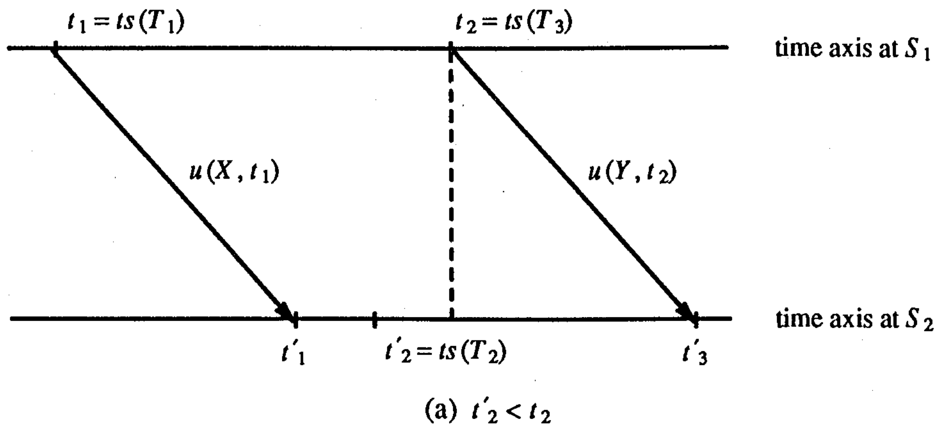


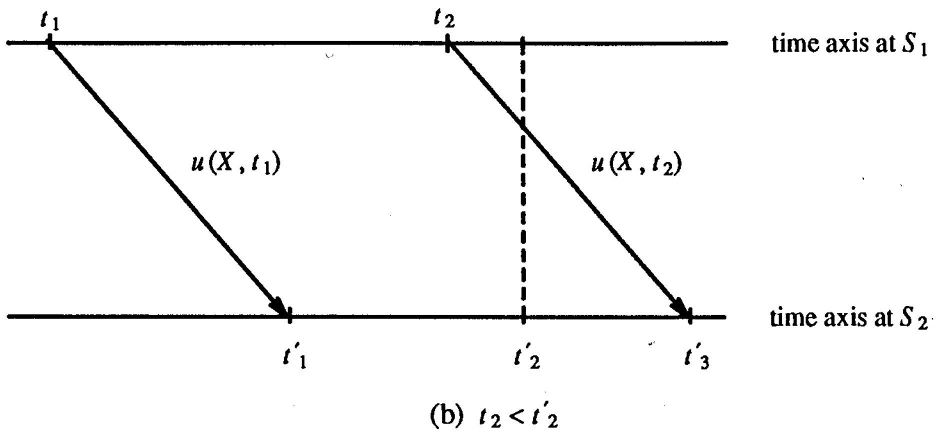
Figure 7.2.1 An architecture for GTOS.

locks held / new request	hr-lock	hw-lock	lr-lock	lw-lock
hr-lock	compatible	incompatible	compatible	preempts
hw-lock	incompatible	incompatible	preempts	preempts
lr-lock	compatible	incompatible	compatible	incompatible
lw-lock	incompatible	incompatible	incompatible	incompatible

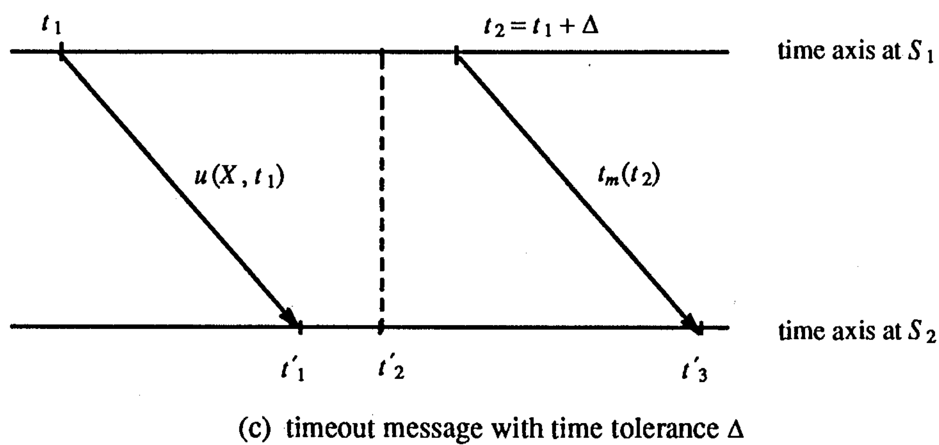
Figure 7.2.2 Compatibility among h-locks and l-locks.



(a) $t'_2 < t_2$

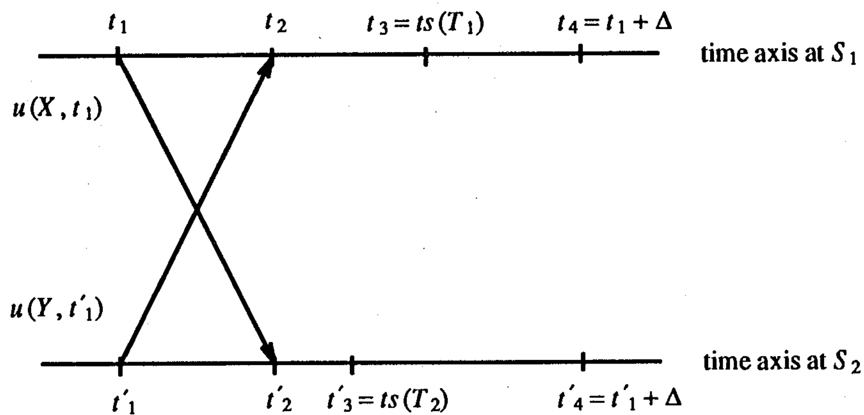


(b) $t_2 < t'_2$

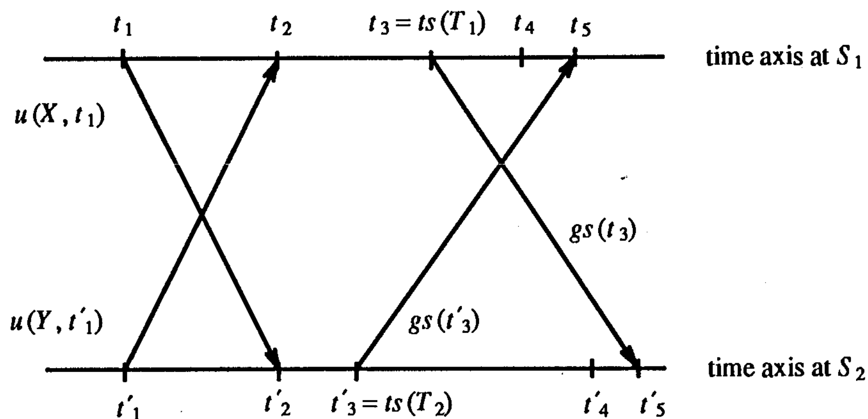


(c) timeout message with time tolerance Δ

Figure 7.3.1 Update and timeout messages.

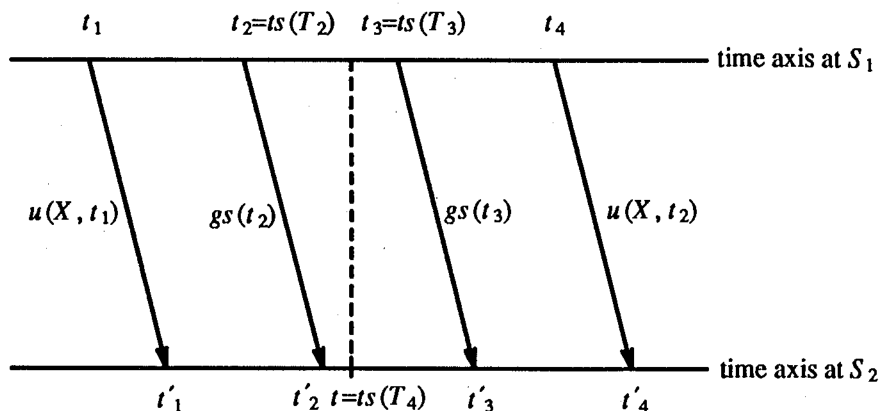


(a) Deadlock.

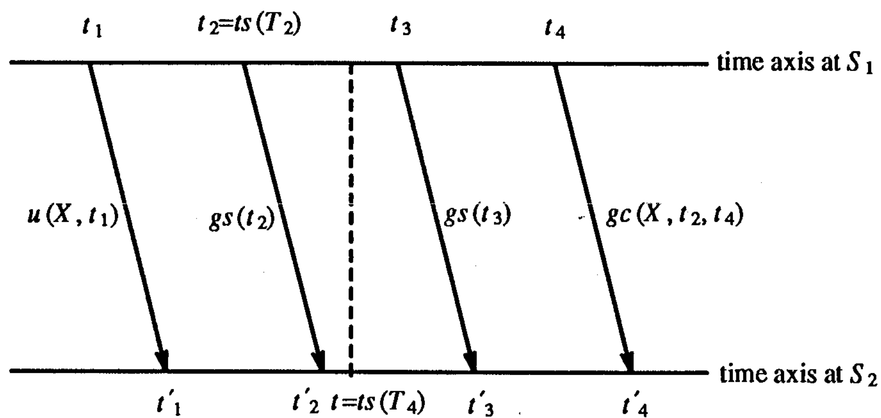


(b) Global-start messages, $gs()$.

Figure 7.3.2 Deadlock and a remedy.



(a) Problem caused by global-start messages.



(b) Remedy.

Figure 7.3.3 Global-start messages and global-completion messages.

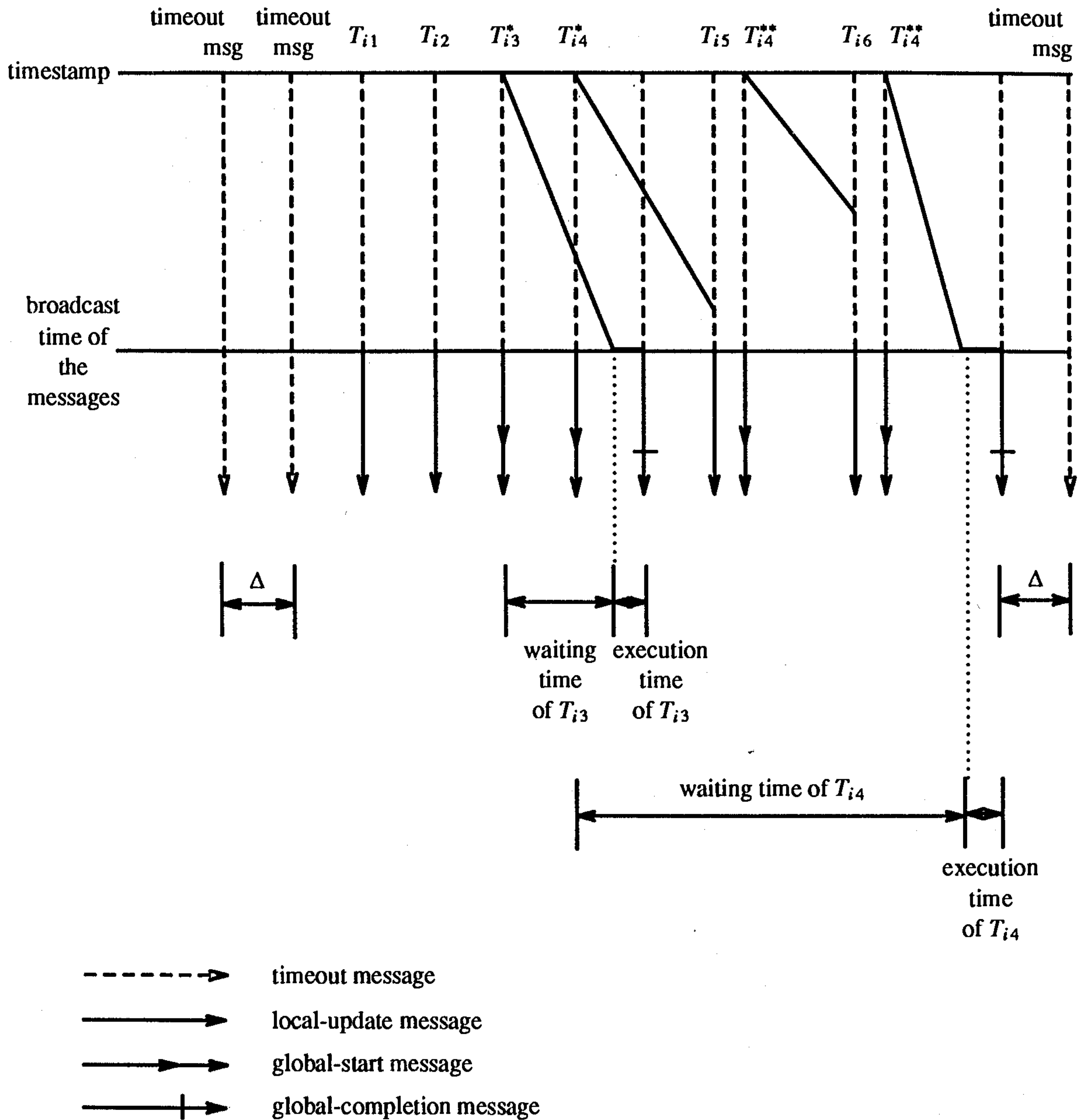


Figure 7.4.1 Time Chart of Transaction Execution.