A NEW PROTOCOL FOR MULTIDATABASE CONCURRENCY CONTROL

By

YONGHO AN

Bachelor of Science

The Ohio State University

Columbus, Ohio

1994

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1997

OKLAHOMA STATE UNIVERSITY

A NEW PROTOCOL FOR MULTIDATABASE

CONCURRENCY CONTROL

Thesis Approved:

H. Lu

Thesis Advisor

R. E. Hedin

J Chandler

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGEMENTS

I would like to express special appreciation to my advisor Dr. Huizhu Lu. She provided essential guidance, inspiration, and supervision through my thesis work. Dr. Huizhu Lu continued to spend endless hours reviewing my work and offerting suggestions for furthur refinement.

I would like to thank my other committee members, Dr. G. E. Hedrick and Dr. J. P. Chandler. Their time and effort are greatly appreciated.

Finally, I would like to give my sincere thanks to my family, my parent, and the second aunt's family for their continued support, and for encouragement at times of difficulty. I could not have done without continued love and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Since information has expanded so rapidly, it has come to have unlimited power in all fields such as the stock market, industries, and the Internet. However, it has been difficult for individuals to locate and to access data within different sites of their own systems. The necessity to access information across several databases, geographically separated but containing homogeneous data, led to the concept of Distributed Database Systems (DDBS). A DDBS is a collection of sites connected by a network. A user of any site can access any data as though the data were stored at the user's own site. Two commonly cited advantages of distributed database systems are *sharability* of data and resources and *local autonomy*. One way to realize these advantages is to build distributed systems in a bottom-up fashion, by putting together existing centralized database managers. This construction gives rise to a *multidatabase system* (MDBS).

Multidatabase systems (also referred to as federated databases or heterogeneous distributed databases) provide a uniform interface for accessing distributed information sources. They allow users to retrieve the correct data from multiple heterogeneous databases transparently. MDBs were inspired by the proliferation of networks and databases and by the need to protect investment in existing systems. MDBSs allow integrated access to heterogeneous, pre-existing databases (referred to as local databases) in a distributed system. Each participating node retains local control of resources and processing. This is called *local autonomy*. Global control and structure are derived from local consent and collaboration.

A Taxonomy of Distributed Database Systems

It uses a classification (Figure 1) [Ozsu and Valduriez 91][Baker 90] which characterizes the system with respect to (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity.

Autonomy refers to the distributed of control and indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems exchange information, whether they can independently execute transactions, and

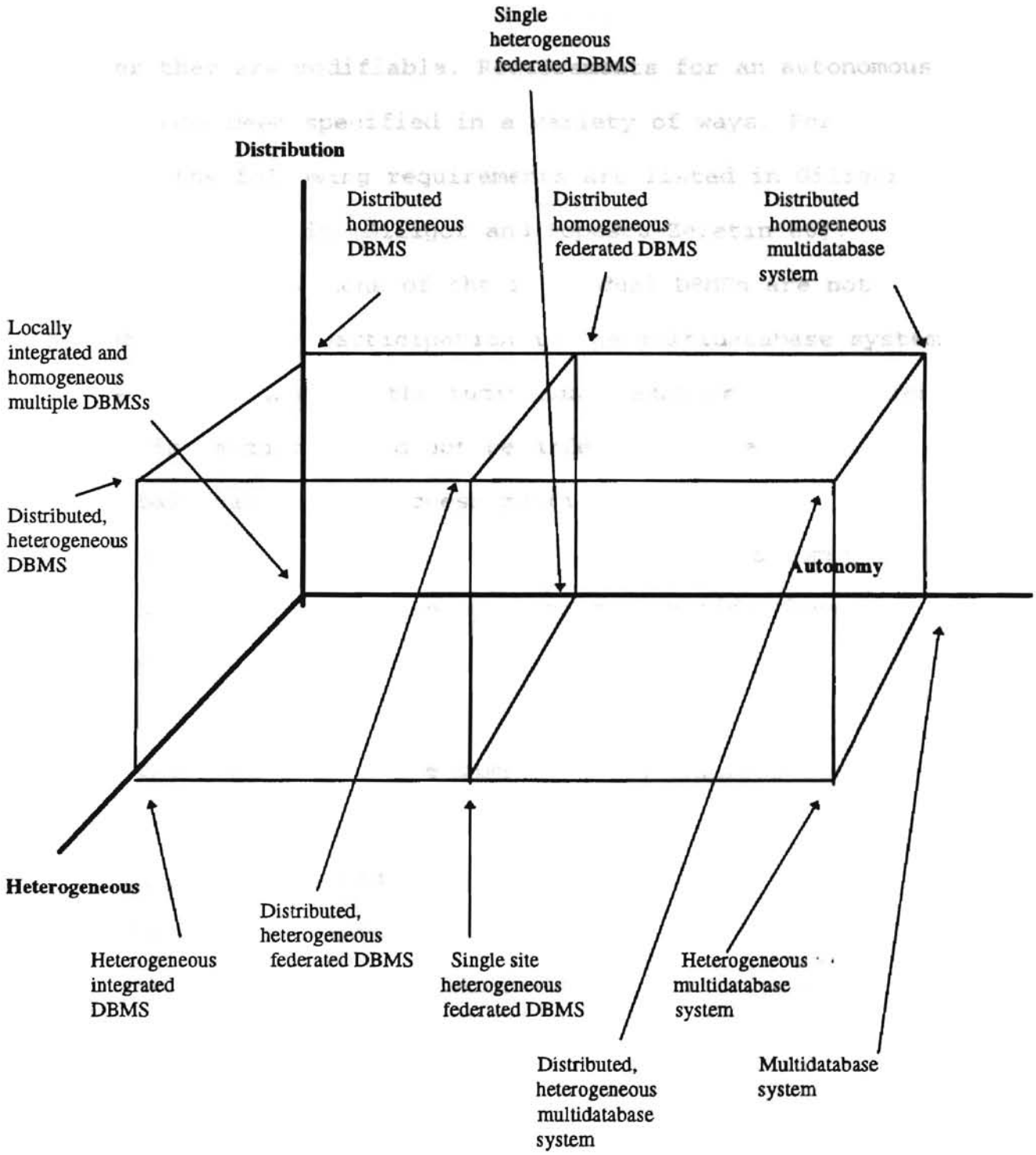

Figure 1. DBMS Implementation Alternatives.

whether they are modifiable. Requirements for an autonomous system have been specified in a variety of ways. For example, the following requirements are listed in Giligor and Popescu-Zeletin [Giligor and Popescu-Zeletin 86].

1. The local operations of the individual DBMSs are not affected by their participation in the multidatabase system.
2. The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.
3. System consistency or operation should not be compromised when individual DBMSs join or leave the multidatabase confederation.

On the other hand, Du and Elmagarmid [Du and Elmagarmid 89] specify the dimensions of autonomy as:

1. Design autonomy: Individual DBMSs can use the data models and transaction management techniques that they prefer.
2. Communication autonomy: Each of the individual DBMSs can make its own decision regarding the type of information it wants to provide to other DBMSs or to the software that controls its global execution.
3. Execution autonomy: Each DBMS can execute the transactions that are submitted to it in itsown way.

A number of alternatives are suggested below. One alternative considered is *tight integration*, where a single-

image of the entire database is available to any user who wants to share the information that may reside in multiple databases. From the user's perspective, the data are logically centralized on a database. In tightly integrated systems, the data managers are implemented such that one of them is in control of the DBMS processing of each user request, even when a request is serviced by more than one data manager. The data managers typically do not operate as independent DBMSs, although they usually have the required functionality.

The second alternative is *semiautonomous* systems, which insist on DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determine what parts of their own databases they will make accessible to users of other DBMSs. They are not fully autonomous systems because they must be modified to permit information exchange.

The final alternative considered is *total isolation*, where the individual systems are stand-alone DBMSs that do not know of the existence of other DBMSs. In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

It is important that the three alternatives considered for autonomous systems are not the only possibilities. They are the three most popular alternatives.

The distribution dimension of the taxonomy deals with data. It is considered in two cases: either the data physically is distributed over multiple sites that communicate with one site over a communication medium, or it is stored at only one site.

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols, to variations in data managers. The important differences considered in this thesis relate to transaction management protocols.

The architectural alternatives are considered in turn. Starting at the origin in Figure 1 and moving along the autonomy dimension, the first class of systems consists of those which are logically integrated. Such systems can be given the generic name *composite systems* [Heimbigner and McLeod 85]. If there is no distribution or heterogeneity, then the system is a set of multiple DBMSs which are logically integrated. Shared-everything multiprocessor environments are an example of such systems. If heterogeneity is introduced, then one has multiple data managers which are heterogeneous but provide an integrate view to the user. In the past, some work was done in this

class where systems were designed to provide integrated access to network, hierarchical, and relational databases residing on a single machine. The more interesting case is where the database is distributed physically even though a logically integrated view of the data is provide to users. This is what is known as a *distributed DBMS* [Ozsu and Valduriez 91]. A distributed DBMS can be homogeneous or heterogeneous.

Next along the autonomy dimension are semiautonomous systems which are commonly called *federated DBMSs* [Heimbigner and McLeod 85]. The component systems in a federated environment have significant autonomy in their execution, but their participation in a federation indicate that they are willing to cooperate with others in executing user requests that access multiple databases. Similar to logically integrated systems, federated systems can be distributed or single-site, homogeneous or heterogeneous.

If one moves to full autonomy, then we get *multidatabase system* architectures. Without heterogeneity or distribution, an MDBS is an interconnected collection of autonomous databases. A multidatabase management system (MDBMS) is the software that manages a collection of autonomous databases and provides transparent access to it. If the individual databases that make up the MDBS are distributed over a number of sites, we have a *distributed*

MDBS. The organization and management of a distributed MDBS are quite different from those of a distributed DBMS.

The fundamental point of the foregoing discussion is that the distribution of databases, their possible heterogeneity, and their autonomy are different issues. It follows that the issues related to multidatabase systems can be investigated without reference to their distribution or heterogeneity. The additional considerations that distribution brings are no different than those of logically integrated distributed database systems for which solutions have been developed [Ozsu and Valduriez 91]. Furthermore, if the issues related to the design of a distributed multidatabase are resolved, introducing heterogeneity may not involve significant additional difficulty. This is true only from the perspective of database management: there may still be significant heterogeneity problems from the perspective of the operating system and the underlying hardware. Therefore, the more important issue is the autonomy of the databases, not their heterogeneity.

The environment considered in this thesis is a multidatabase system, especially transaction management protocols. We assume the optimistic case of fully autonomous DBMSs.

Multidatabase System

A multidatabase system (MDBS) consists of two or more databases, possibly distributed, which are controlled by one or more DBMSs [Brietbart and Silberschatz 87, 88]. A MDBS allows users to manipulate data contained in the databases without modifying current database applications and without migrating the data to a new database. A MDBS also creates the illusion of logical database integration without requiring physical integration of the databases. For simplicity, the intricacies of the DBMSs and data access methods are transparent to the user.

To provide a facility that is acceptable to the end users, as well as the application programmers, an MDBS should adhere to the following principles [Brietbart et al. 90][Brietbart and Silberschatz 88].

1. No modifications to the local DBMS software to accommodate the MDBS are permitted.
2. The autonomy of the local databases are maintained.
3. The MDBS guarantees serializable global transaction execution.
4. The local DBMSs guarantee serializable local transaction execution.
5. No communication exists among the local DBMSs.

Preventing changes to the DBMS software is an important issue. Modifying the DBMSs to interact with the MDBS puts a heavy burden on the MDBS developers when support for a new DBMS is added. These changes may also create difficult problems, both in maintaining current applications and in maintaining the DBMS software.

The concept of local autonomy, a key characteristic of MDB, requires that existing local transactions be allowed to execute as if the MDBS were not present. Local autonomy also requires that DBMS maintenance and performance tuning be allowed to continue as usual. That is, local DBMSs retain fully control over local data and processing. Each local DBMS participates in the multidatabase by sharing some or all of its data. The data to be shared with the global system are defined in a view presented to the local DBMS user interface. To the local DBMS, the MDB appears like any other user because the global system does not dictate local design. So, when the local DBMS gets a request, called *a global subtransaction,* from a global DBMS for data, the local DBMS can accept (commit) or reject (abort) it.

In MDBS, local and global concurrency control must be addressed separately because of local autonomies. Local concurrency controllers guarantee the correctness, using serializability, of the executions of local transaction and global subtransactions at each local site. On the other

hand, the global concurrency controller is responsible for retaining the consistency of the global database.

Although each individual transaction is correct, data consistency can be destroyed during transaction in concurrency control [Ozsu and Valduriez 91]. So, in order to ensure data consistency, the concept of serializability is needed. Each transaction should transfer the system from one consistent state into a new consistent state without any violation. In addition, temporary inconsistency can occur during the execution of a transaction, but the final state should be always consistent.

Objective

The object of this thesis is to propose a new protocol for multidatabase concurrency control to avoid deadlock and to retain serializability by combining advantages of pessimistic and optimistic approaches.

This new protocol is implemented on a sequential machine. The final result should be faster and have higher degree of concurrency in comparison with Thomas' Write Rule.

CHAPTER 2

LITERATURE REVIEW

Most concurrent control approaches have relied on locking of data objects in a *"pessimistic"* sense that assume that the conflicts between transactions are quite frequent. More recently, the methods are used in an *"optimistic"* sense; they rely mainly on transaction back-up as a control mechanism while "hoping" that conflicts between transactions will not occur.

Approaches of Past Work

1. Disadvantages of the locking approach (pessimistic)

1) Lock maintenance represents an overhead that is not present in the sequential case. Even read-only transactions that do not affect the integrity of the data should use locking in order to guarantee that the data being read are not modified by other transactions at the same time.

2) Since the locking approach is not deadlock free, deadlock detection must be considered to be part of locking

maintenance overhead. That is, there is no general-purpose deadlock-free locking protocol for databases that always provide high concurrency.

3) To allow a transaction to abort itself when a mistake occurs, locks cannot be released until the end of the transaction. This may lower concurrency significantly.

4) Locking may be necessary only in the worst case in 'optimistic' sense.

Research directed at finding deadlock-free locking protocols may be seen as an attempt to lower the expense of concurrency control by eliminating transaction backup as a control mechanism. But, if we consider it in the *optimistic* sense that relies for efficiency on the hope that conflicts between transactions will not occur or will be rare. This is called the "*optimistic approach*" [Bernstein and Goodman 81] [Darcy and Boston 83] [Eliezer et al. 91] [Kung 81]. Since locks are not used, it is completely deadlock-free and allows a high level of concurrency [Bernstein and Goodman 81] so that when transaction conflicts are very rare.

2. The idea of the Optimistic Approach

1) Since reading a value never can cause a loss of integrity, reads are completely unrestricted.

2) Writes are severely restricted. Any transaction must consist of three phases: *a read phase, a validation phase,*

and write phase [Barghouti et al. 91] [Bernstein and Goodman 81] [Ozsu and Valduriez 91]. During the read phase, all writes take place on local copies. Then, if it can be established during the validation phase that the changes the transaction made will not cause a loss of integrity, the local copies are made global in the write phase. The step in which it is determined that the transaction will not cause a loss of integrity is called validation.

If validation fails, then the transaction will be blocked-out and restarted as a new transaction. Thus a transaction will have a write phase only if the preceding validation succeeds. On the other hand, optimistic algorithms [Ozsu and Valduriez 91] delay the validation phase until just before the write phase. Thus, an operation submitted to an optimistic scheduler is never delayed comparing with the locking scheduler. The read and write operations of each transaction are processed freely without updating the actual database. Each transaction initially makes its updates on local copies of data. The validation phase consists of checking whether updates on local copies would maintain the consistency of the database. If the answer is affirmative, the changes are made globally. Otherwise, the transaction is aborted and has to restart and that cause starvation. Of course, permitting the transaction exclusive access to the database after a specified number of

trials had been tried for many years, but this try reduced the level of concurrency, the biggest advantage of the optimistic approach. Therefore, the solution of the starvation problem has been one of the most important areas of database in recent years.

An Optimistic Commit Protocol for Distributed Transaction Management [Eliezer et al. 91]

A major disadvantage of the two-phase commit (2 PC) protocol is the potential unbounded delay when a certain transaction failure occurs. By using compensating transactions, [Eliezer et al. 91] is obtained by using revised 2 PC protocol that overcomes theses difficulties. In the revised protocol, locks are released as soon as a site votes to commit a transaction, thereby solving the indefinite blocking problem of 2 PC. If the transaction is to be aborted, then its effects are undone semantically using a compensating transaction. Therefore, semantic, rather than standard, atomicity is guaranteed. But this protocol reduces to a serial protocol when no global transactions are aborted, and excludes unacceptable executions when global transactions fail.

A Time-based Distributed Optimistic Recovery and Concurrency Control Mechanism [Gafni and BapaRao 92]

Optimistic methods of concurrency control can achieve high throughput but impose a space overhead. [Gafni and BapaRao 92] describes a time-based approach to distributed concurrency control and recovery that alleviates the high cost of optimistic methods by combining the solutions to concurrency control, recovery management and localized control into a single flexible yet powerful and efficient mechanism. This approach adapts the object-oriented Timewarp mechanism - it was designed for networks of cooperative processes where all processes belong to one application and accomplish a common task. For this type of application to be correct, all messages have to be processed in strictly increasing order - to handle competing processes rather than the co-operating processes for which it was originally intended. This method assumes that no event synchronization is necessary to allow the transaction steps to proceed; when that assumption fails, a rollback mechanism restores the system to a consistent state. The result is a completely decentralized, nonlocking concurrency and recovery protocol that supports more general features in

corporating desirable features of other distributed applications, such as the use of versioning and active objects. But this method imposes a heavy space overhead, and a high transaction failure rate.

Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types [Herlihy 90]

An optimistic concurrency control technique is one that allows transactions to execute without synchronization, relying on commit-time validation to ensure serializability. More recently, several new optimistic techniques are proved. But these methods have classified operations only as read or write. [Herlihy 90] systematically exploits type-specific properties of objects to validate additional interleaving. Necessary and sufficient validation conditions can be derived directly from an object's data type specification. Herlihy's method is also modular. That is, it can be applied selectively on a per-object basis in conjunction with standard pessimistic techniques such as two-phase locking, permitting optimistic methods to be introduced exactly where they will be most effective[Herlihy 90]. This method enhances the availability of replicated data, circumventing

certain tradeoffs between concurrency and availability imposed by comparable pessimistic techniques.

Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases [Veijalainen and Wolski 92]

[Veijalainen and Wolski 92] shows the algorithms to prepare for certification and to commit certification to protect against serialization errors called global view distortions and local view distortions. View serializable overall histories are guaranteed in the presence of most typical failures. The assumptions are that the participating database systems produce rigorous histories; e.g., by using the strict two-phase locking, and that no local transaction may update the data accessed by a global transaction that is in the prepared state.

Thomas' Write Rule (TWR)

If we suppose a timestamp ordering (TO) scheduler receives write transaction, wi [x], after it has already sent wj [x] to the DM when ts(Ti) < ts(Tj), TO rule rejects wi [x]. But, this rejection is unnecessary if the

scheduler only is concerned with write-write synchronization (ww synchronization). That is, processing a sequence of write transactions in TO produces the same result as processing the single write transaction with maximum timestamp. Late operations can be ignored. This is called *Thomas' Write Rule (TWR)*[Bernstein 87]. It never delays or rejects any operation. When a TWR ww synchronizer receives a write transaction that has arrived too late insofar as the TO rule is concerned, it simply ignores the write transaction but reports its successful completion to the TM.

A simple example is the following:

```
S = A1 : R(x)          W(x);
    A2 :       W(x);
```

Using TWR, write step of A1 is simply ignored.

Multidatabase System Architecture

The component-based architectural model of a multidatabase management system (MDBMS) features full-fledged DBMSs, each of which manage a different DBMS. The MDBMS provides a layer of software that runs on top of these individual DBMSs and allows users to access various databases. Each DBMS has its own transaction processing components. The components are a transaction manager , called *Local Transaction Manager* (LTM), a *Local Scheduler*

(LS), and a *Local Data Manager* (LDM). The function of LTM is to interact with the user and coordinate the atomic execution of the transaction. The LS is responsible for ensuring the correct execution and interleaving of all transactions presented to the LTM. The local recovery manager ensures that the *Local Database* (LDB) contains all of the effects of committed transactions and none of the effects of uncommitted ones.

We assume each autonomous DBMS to be a single database, and the MDBMS layer is simply another "*user*" . The scheduling of transactions which require multiple DBMSs is done by the MDBMS layer. The transaction manager of the MDBMS layer is called the *Global Transaction Manager* (GTM) since it manages the execution of global transaction (Figure 2).

In multidatabase Concurrency Control, correcting conflicting serializability at two levels which are local and global transactions, has been the most difficult problem. Each local scheduler cannot ensure the consistency of global transactions. Even though event controlled by the local scheduler are serializable, their global execution order may be not serializable causing indirect conflicts.

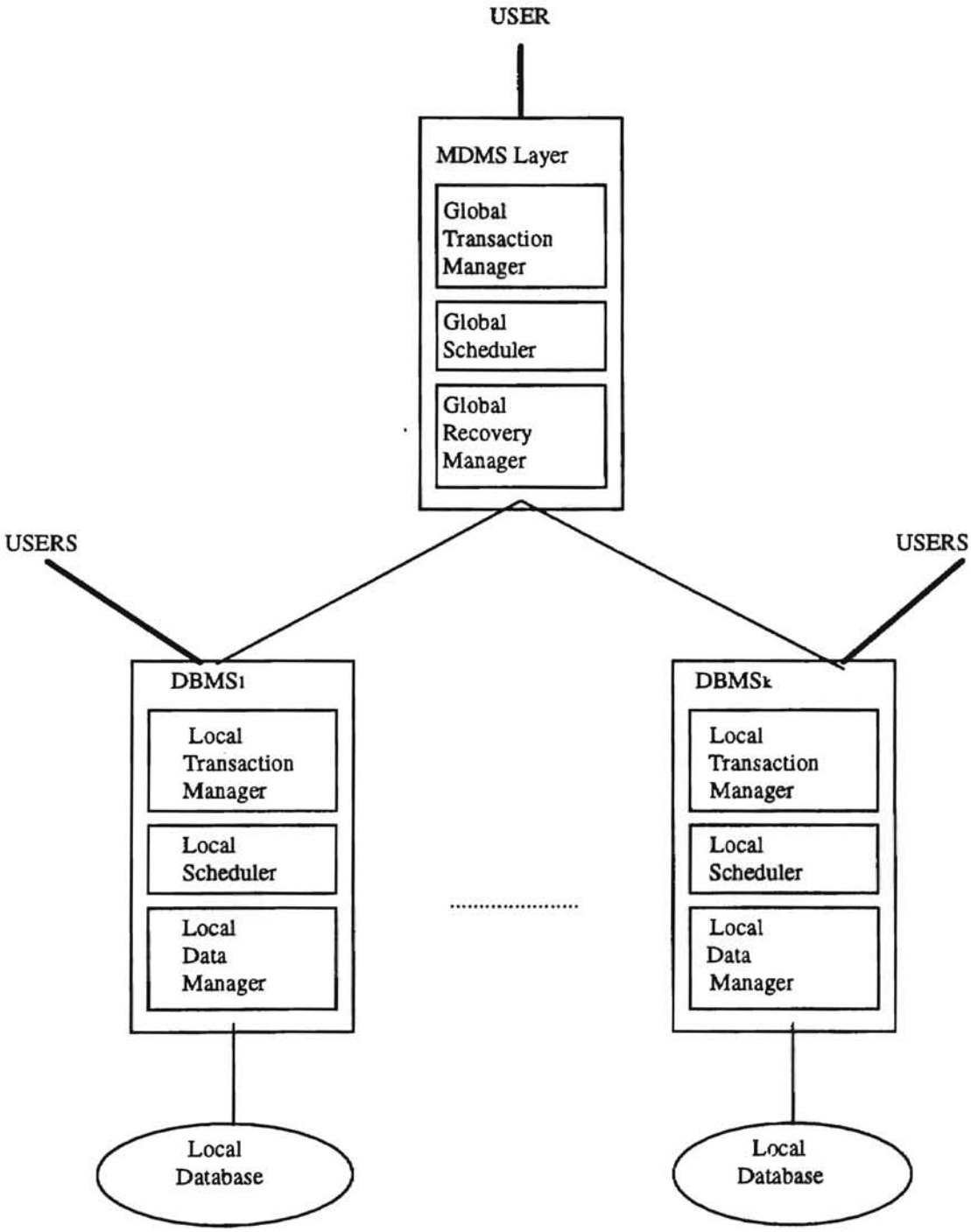

Figure2. MDBS Architecture.

Transactions on Multidatabase System

A transaction Ti is a partial order with ordering relation <i where [Bernstein 87]

1. Ti $\subseteq$ {ri[x], wi[x] | x is a data item} $\cup$ {ai, ci}

2. ai $\in$ Ti, iff ci $\notin$ Ti

3. if t is ci or ai (whichever is in Ti), for any other operation p $\in$ Ti, p <i t and

4. if ri[x], wi[x] $\in$ Ti, then either ri[x] <i wi[x] or wi[x] <i ri[x].

MDBS transactions have two type transactions which are the local and global transactions. The execution of global transaction is co-ordinates by the *global transaction manager* (GTM) that is a software package built on top of the existing DBMSs whose function is to ensure that the concurrent execution of local and global transactions is serializable. Ensuring global serializability in an MDBS is complicated by the fact that each of the participating local DBMSs is a pre-existing database system whose software cannot be modified. As a result (the characteristics of GTM), (Figure 3)

1) The function of GTM has duties for concurrency control (or scheduling) to guarantee serialized execution of transactions by controlling the execution of

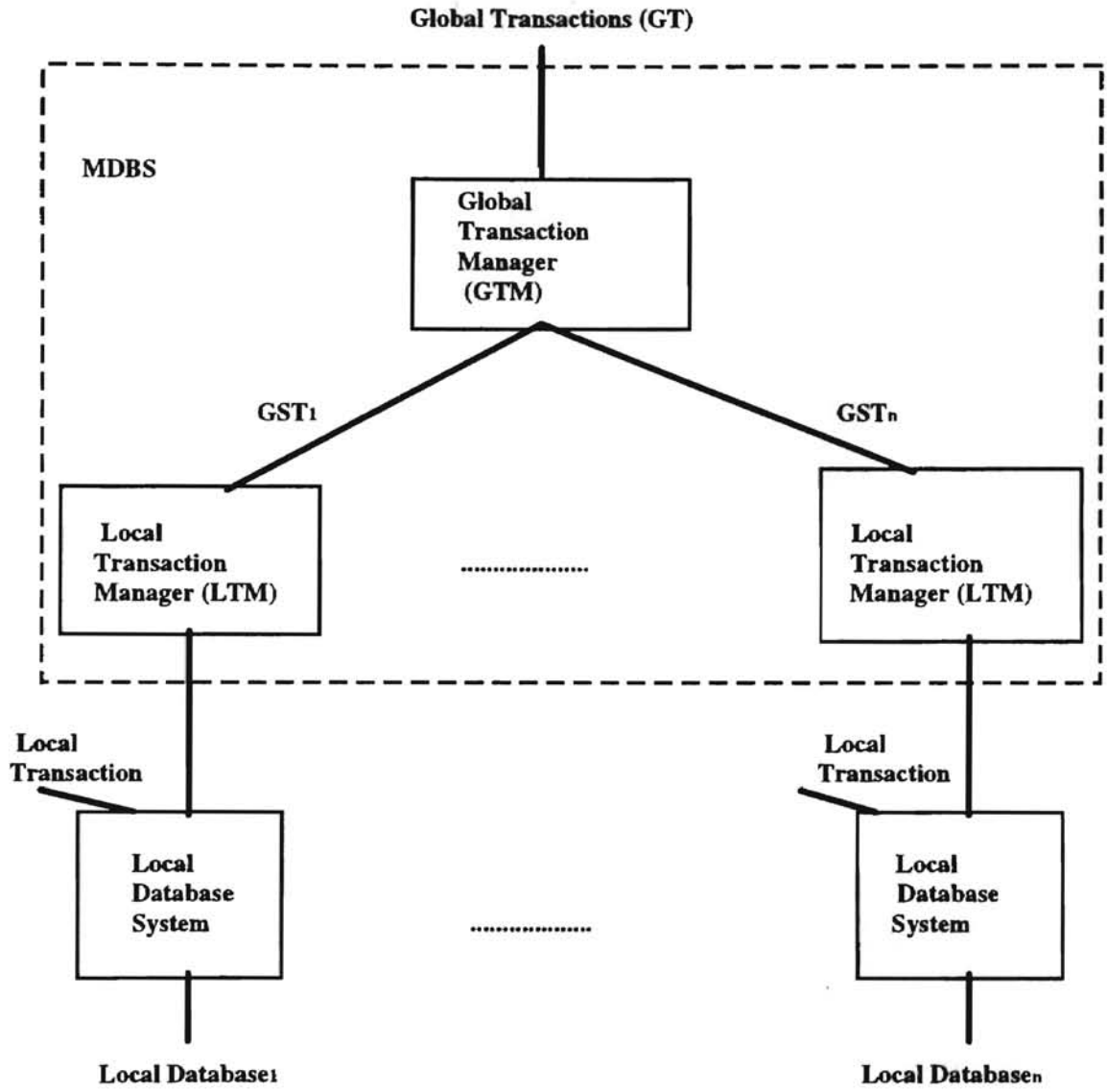

Figure 3. Multidatabase Model

subtransactions, commitment and recovery to achieve atomicity and durability of global transactions in the presence of failures. It allocates one LTM for each of the sites referenced by the global transaction.

2) The LTM is the remote component of the MDBMS that runs directly on top of each LDBS. It receives operations of subtransactions from the GTM, submits them to the LDBMS, and sends the results to the GTM. Once an LTM is allocated, it is not de-allocated until the transaction commits or aborts. On the other words, the GTM is centrally located and controls the execution of global transactions. It communicates with the various local DBMSs by means of LTM per site that execute at each site on top of the local DBMSs, which do acknowledge the completion of operations to be submitted by the LTMs. An LTM has several responsibilities with respect to the execution of a global subtransaction.

2.1) Each local DBMS may follow a different concurrency control protocol.

2.2) Local DBMSs may not communicate any information (e.g., conflict graph) relating to concurrency control to the GTM. They are not aware of each other. On the other words if a local transaction is submitted to a local DBMS, then no other local site is aware of that transaction. Local DBMSs behave as if MDBS does not exist according to the concept of local autonomy.

3) The GTM is unaware of *indirect conflicts* between global transactions due to local transactions at the local DBMSs.

This is due to the fact that the local pre-existing applications make calls to the local DBMS interfaces, and thus the GTM, which is built on top of the local DBMSs, is not involved in the execution of the local transactions. So in order to ensure the correct behavior of the system, the MDBS must be able to synchronize the execution of global transactions with local ones. This is generally not possible to achieve if arbitrary local transactions can be submitted at local sites, since a local transaction may change a value of a replicated data item. To guard against such behavior the MDBS must provide a concurrency control scheme and formulate restrictions on the type of local transactions that can be tolerated by the MDBS concurrency control mechanism

Multidatabase Scheduler

MDBS Serializability is the combination of two types: each *local database scheduler* (LS) produces a serializable execution ordering and the set of committed global transactions are globally serializable; that is LS and *global scheduler* (GS) together create an acyclic graph ordering of the executions. In other words, a global serialization graph [Bernstein 87] for a global schedule *S*

is a directed graph whose nodes represent global transactions and whose arcs are defined by Thompson [87] as follows.

{Ti -> Tj | there exists operation Oi in transaction Ti and operation Oj in transaction Tj, such that Oi conflicts with Oj and Oi occurs before Oj in a global schedule S}.

A global schedule *S* is the set of all operations belonging to local and global transactions with a partial order <s on them. The local schedule at a site k, denoted by Sk, is the set of all operations (belonging to local and global transactions) that execute at k with a total order <k on them. The schedule Sk is a *restriction* [Mehrotra et al. 92] of the global schedule S.

Scheduling of transactions in a MDBS must be accomplished at the global and local levels. Since we assume that each DBMS can generate local execution ordering serializablly, the only requirement of the MDBMS is to submit global subtransactions to each DBMS. In global scheduler, following things become apparent:

1) all operations in global subtransactions must be assumed to conflict if they are submitted to the same DBMS at the same time.

2) Since each subtransaction is dependent upon the ordering of other related subtransactions, global transactions which

access mutually disjoint sets can conflict due to local transactions. It is called *indirect conflict*.

Multidatabase Serializability

Multidatabase serializability is the combination of two types of serializable histories in a multidatabase history. When each local database scheduler produces a serializable history and the set of committed global transactions are globally serializable, the MDBMS is said to have produced an MDB-serializable schedule. This is the same as ensuring that the MDB history is *λ–acyclic* or that for each local history it is equivalent to some serial schedule, and that the MDB schedule is *γ–acyclic* or the global history is equivalent to some serial ordering. Therefore, the proof process is simplified because each type of transaction can be considered separately [Baker 90].

Theorem 1. *(MDB Serializability Theorem)* [Baker 90] A multidatabase history (MH) is MDB-serializable if and only if MSG(MH) is both *γ–acyclic* and *λ–acyclic*. If given a *γ–acyclic* and *λ–acyclic* MSG for a multidatabase history MH, MH is MDB serializable.

Since each DBMS produces only serializable schedules, λ-cycles at a specific DBMS are not possible. Further, the data is not replicated, so λ-arcs are not formed between transactions at different DBMSs. Therefore, λ-cycles are not possible, and the proof is accomplished in *γ–acyclic* as described below.

γ–acyclic: Without loss of generality, assume that MH =< LH, GH > refers to the committed projection of a multidatabase history. Consider the global history GH defined over the set of transactions GT = {GT1 , ... , GTn). Without loss of generality, assume that the committed history (C(GH)) is {GT 1 , ... , GTm}. The Γ–*vertices* of MSG(MH) ({GT1, GT2 , ... ,GTn}) are *γ –acyclic* so they can be *topologically* sorted with respect to *γ–arcs*. Let the permutation i1, i2, ... , in in be a permutation of 1 2 ,..., n such that GTi1, GTi2, ... , GTin in a topological sort of the Γ–*vertices* of MSG(MH). Let GHs be the serial history of GTi1, GTi2, ... , GTin. We will prove that: GH ≡ GHs. Let p ∈ GTi and q ∈ GTj and p and q conflict such that p <GH q. This means that there is a γ–arc GTi -> GTj in MSG(MH). Therefore, in any topological sort of GH, GTi precedes GTj. Thus, all operations of GTi precede all

operations of GTj in any topological sort. Thus GH ≡ GHs. Since GHs is MDB-Serial, GH is MDB-Serializable.

Also, given that the history is MDB-serializable, we will show that the MSG produced must be both γ–acyclic and λ–acyclic.

First note that the set of λ–arcs is subdivided into a number of disjoint subsets, each for one LH. Assume that a cycle exists in one of the subsets of λ–arcs as follows: Ti -> ... -> Tn -> ... -> Ti. This implies that an operation of Ti precedes and conflicts with an operation of Tn and that an operation of Tn precedes and conflicts with an operation of Ti. This means that the DBMS which has generated the particular local history has incorrectly scheduled its transaction, which contradicts the assumption that all local schedulers function correctly. Thus, *λ–cyclicity* cannot occur in a MDB-serializable history.

Suppose MH is serializable. Let MHs be a serial history equivalent to the MDB-serializable history MH. Consider a γ–arc (GTi -> GTj) ∈ MSG(MH). This means that there exist two conflicting operations p ∈ GTi and q ∈ GTj such that p < q in some local history. This is true since both of these operations execute on the same database. Since MHs is equivalent to MH and there is an arc from GTi

-> GTj, all operations of GSTi at site k occur before those of GSTj at site k. Suppose there is a *γ-cycle* in MSG(MH). This implies that there exists a DBMS at site m which scheduled an operation r ∈ GTj before an operation s ∈ GTi. Since this implies that GTj <GH GTi in MHs, an operation of GTj precedes any of GTi's. But, an operation of GTi is known to precede an operation of GTj at DBMS at site k, which is contradictory.

Chapter III

A NEW PROTOCOL FOR MULTIDATABASE CONCURRENCY CONTROL

Implementation Details

Implementation Detail

The primary objective of this thesis is to show that the new protocol has better serializability than other protocols and is deadlock-free. This new protocol is a protocol that is made by combining the advantages of both pessimistic protocols, especially the two phase lock, and optimistic protocols. The first step is to know the advantages of each approach. Then, we can design this new protocol that participates in the multidatabase system. The simulated environment will consist of two separated local database systems each having its own transaction processing system for distributed global control of multidatabase system. Each transaction will be checked on the checking board that shows the transaction order and shows which command has reservation or lock. It also runs on each local database in order to test serializability and freedom from deadlock.

Environment of the Implementation

Platform : Sequent

Language: C

Special command used: Fork

An Algorithm of the New Protocol in MDBS

A new protocol has one reservation before read commands and one lock before write commands. A reservation is not a lock. It is like checking-point or half-lock.

In order to prove the new protocol is serializable, it should prove to be free from conflict. Conflict between transactions may be read-write, write-read, or write-write. For a pair of conflicting operations, the relative order of execution is important [Mahesh 90]. If the order is the same for each pair of conflicting operations from the two transactions, the transactions can be regarded as have been executed in the serial order.

1) *read - read*: before a read command, mark a reservation on the data that does not affect any other read command. This reservation mark will affect an anticipated write command. That is, if a *read - write* is formed, then the read command will be delayed.

2) *write - read*: before locking for a write command, the algorithm checks if there is another write- lock. If there is, then another write-lock will form a *write - write*. Otherwise, put the lock and then process. The next read command will see the write-lock, mark a reservation for the next process, then wait until getting unlock signal.

3) *read - write*: before a read command, reserve and then process read command. The next write command, it will be processed without any regard for read commands. After reading, it compares that read $\cap$ write = $\emptyset$. If it is not equal, then the read command will be delayed until the write command has been processed. Then, the read command will do its process again until read $\cap$ write = $\emptyset$.

4) *write - write*: The first write command locks its data and its process. The second write command waits until the first write command has been processed. Then, it will do its process making all processes serial.

Example 1) Assume that a multidatabase system is composed of two local databases whose contents are: LDB1 = {d, e, f, g} and LDB2 = {s, t, u, v}. Two global transactions are posed to the

GT1 : read (d); read (e); write (s); write (d);

GT2 : read (d); read (u); write (s); write (d);

These generate the following global subtransactions:

GST11 : read (d); read (e); write (d);

GST12 : write (s);

GST21 : read (d); write (d);

GST22 : read (u); write (s);

Further, we introduce local transactions into each DBMS as follows:

LT1 : read (e); write (e); write (d);

LT2 : read (u); write (u);

With the new protocol, we can assume the following local and global histories:

LH1 : read11 (d); read11 (e); write11 (d); readL1 (e); read21 (d); writeL1 (e); writeL1 (d);

LH2 : read22 (u); write22 (s); write12 (s); readL2 (u); writeL2 (u);

The following global subtransaction histories can be derived from these local histories:

GSH1 : read11 (d); read11 (e); write11 (d); read21 (d); write21 (d);

GSH2 : read22 (s); write22 (s); write12(s);

Finally, the global history is the partial order which combines GSH1 and GSH2 as GH = { GSH1 $\cup$ GSH2 }. The multidatabase history is the tuple MH = < { LH1, LH2}, GH >.

Example 2) Comparing the new protocol with Thomas' Wite Rule (TWR) in ww synchronization, it appears there is no difference. But, if they are compared in rw synchronization,

not only ww synchronization, the new protocol is more secure in serialization than TWR. As a example, assume TWR combines a two phase locking (2 PL) rw synchronizer, and T = { t0, t1, t2, t3 }, where t0 = w0(x), w0(z), w0(y);

t1 = r1(x), r1(z), w1(x);

t2 = r2(x), w2(y);

t3 = r3(z), w3(z), w3(y);

and, ts(t1) < ts(t0) < ts(t2) < ts(t3).

Scheduler, S = w0(x), r1(x), w0(z), r1(z), r2(x), w0(y), r3(z), w3(z), w2(y), w1(x), w3(y);

Both two phase locking (2 PL) and the new protocol are serial, but the new protocol is faster. That is, when read-lock in 2 PL is on a data, other read or write commands cannot be applied on that data. But, any command in the new protocol can be processed without any violation. Since the reservation on the data is not a lock. This is similar to a check-point that shows the data has been read, so the next command does not need to wait. If a read command is next command, then unless there is a lock on the data, it is processed simultaneously with earlier commands. If a write command is next, then it processes the data and makes a read-write form. Also, a read command of TWR with 2 PL needs three steps: read-lock, read, unlock. However, the new protocol requires only two steps: reservation, read. Thus, the new protocol is faster than TWR with 2 PL.

By using a pessimistic approach without starvation such as a two phase commit (2 PC), the degree of concurrency is reduced by using a read-lock that is not necessary, causing the deadlock. Also, by using only an optimistic approach, starvation can occur. Consequently, the new protocol is realized using on optimistic approach but using reservation on data. The reservation has the role of protection from starvation and reduction of the degree of concurrency, the biggest advantages of an optimistic approach.

Deadlock-Free

Deadlock is a situation in which each transaction in a set of transactions is blocked waiting for another transaction in the set, and therefore none will become unblocked unless there is external intervention [Bernstein 87].

A useful tool in analyzing deadlocks is a *wait-for graph* (WFG). A WFG is a directed graph that represents the wait -for relationship among transactions. The nodes of this graph represent the concurrent transactions in the system [Ozsu and Valduriez 91]. An arc Ti -> Tj exists in the WFG if transaction Ti is waiting for Tj to release a lock on some entity. It is easier to indicate the condition for the

occurrence of a deadlock. A deadlock occurs when the WFG contains a cycle.

The formation of the WFG is more complicated in multidatabase system., since more than two transactions that participate in a deadlock contain may be running at different sites. It is called a *global deadlock.* In multidatabase system, it is necessary to form a *local wait-for graph* (LWFG) and a *global wait-for graph* (GWFG) which is the union of all the LWFGs.

A LWG consists of only local transactions and global subtransactions at a single site [Ceri and Pelagati 84]. The graphs (Figure 4) at each site on example 2 are maintained by local DBMSs and are unavailable to the MDBS.

Example 3)

```
Site 1: GT1: read(x); write(y);
        LT1: read(x); write(w);
Site 2: GT2: read(y); write(z);
        LT2: write(z);
Site 3: GT3: read(z); write(x);
        LT3: read(z);
```

Suppose all global transactions executes concurrently, with each global transaction issuing its 'read' before any transaction issues its end.

At this point GT1 has read-lock on x

GT2 has read-lock on y

GT3 has read-lock on z.

After processing GT1, GT2, and GT3, local transactions have read-lock on x and write-lock on z and on v. In Figure 4, it shows the allocation of the local transactions and global subtransactions has no problem.

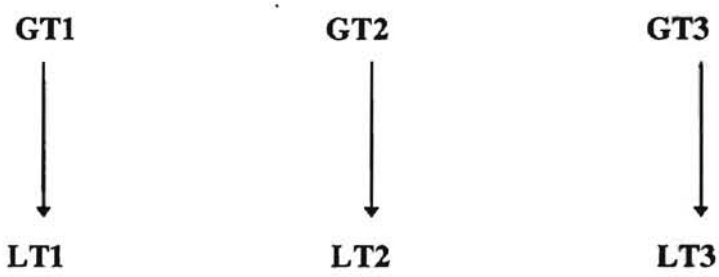

Figure 4. Local wait-for graph.

But, if we construct a GWF by merging the LWGs, it shows the following cycle in general algorithms on Figure 5, especially in pessimistic algorithms.

That is, all global transactions must obtain write-locks:

GT1 requires write-lock on y

GT2 requires write-lock on z

GT3 requires write-lock on x

But,

GT1 cannot get write-lock on y until GT2 releases read-lock

GT2 cannot get write-lock on z until GT3 releases read-lock

GT3 cannot get write-lock on x until GT1 releases read-lock.

Thus, this is deadlock.

Figure 5. Global wait-for graph.

As it is mentioned above, since the new protocol is a locking system in optimistic approach, it is deadlock-free compared to other locking systems. Since this new protocol has a reservation before *read*, not a read-lock, next *write* does not need to be wait for releasing the read-lock of the manner of two phase-commit. Therefore, it is definitely deadlock free.

Outline of Transaction Processing

1. The GTM decomposes every global transaction submitted into as many global subtransactions as the number of sites in which the transaction has to be executed, each of which accesses only one LDB. The GTM maintains a waiting queue to record information about the global subtransactions, and maintains state queue to record state of global subtransactions on each LDBS.
2. The GTM determines an order among the global transactions so that their serialization orders are compatible in all local sites they are executed, and allocates LTM to each subtransction in that order.
3. A global subtransaction executed at the local site is allowed to enter into the waiting queue after receiving READY instruction from the GTM which acts as the coordinator and remains in this state till the coordinator issues COMPLETED or NO COMPLETED instruction for global commit or abort.
4. If one of the global subtransactions fails, the ABORT state is recorded into the state queue, and send the message, ABORT, back to the all LTMs allocated.
5. Communication between the coordinator and local sites is accomplished through the LTMs.

6. The LTM converts the global read/writes to the language understandable by the local DBMS at that site. Also, the LTM keeps recording information about each subtransaction submitted, along with the result of subtransaction execution which is passed on to the GTM.

7. The LTM at each local DBMS ensures local serializability.

8.If there is no local transaction, the global subtransaction does not need to be scheduled with any local transaction.

9. The LTM passes a message from the scheduler to the GTM which results of the transaction commit or abort.

Pseudocode for the GTM

```
DO forever
BEGIN
  initialize the waiting queue, state queue, and all other
  variable;
  On receiving a global transaction DO
  BEGIN
    WHILE
      decompose into subtransaction;
      allocate LTMs for each subtransaction;
      record the global subtransactions and information
      about them into the each LDB storage
```

```
    END WHILE
  END
  On receiving a message from one or more LTMs allocated to
  a transaction DO
  BEGIN
    IF message is NO from at least one LTM allocated THEN
      BEGIN
        record the message into the state queue;
        send the message 'ABORT' to all LTMs allocated;
        go to WAIT;
      END
    IF message is YES from all LTM allocated THEN
      BEGIN
        record the message into the state queue;
        send the message 'READY' to all LTMs allocated;
        go to WAIT;
      END
  END
WAIT: wait the message for the complete schedule from LTMs
IF message is COMPLETED from all LTMs THEN
  BEGIN
    record the message into the state queue;
    make the effects of transaction execution in the
    global database;
```

```
      deallocate all LTMs allocated to the transaction;
    END
  IF message is NO COMPLETED from at least one LTM THEN
    BEGIN
      record the message into the state queue;
      deallocate all LTMs allocated to the transaction;
      restart the transaction;
    END
END
```

Pseudocode for the LTM

```
  DO forever
  BEGIN
    initialize a local data structures;
    get a local transaction;
    On receiving a global subtransaaction DO
    BEGIN
      decompose the global subtransaction into atomic
      operations;
      set and enqueue the operations;
      IF there is no local operation THEN
        BEGIN
          do not need to be scheduled;
          break the loop;
```

```
      END
    ELSE
        BEGIN
            call scheduler;
         END
  END
get a message from the scheduler;
IF message is NO THEN
  BEGIN
      record the message into the state queue;
      send the message 'NO' back to the GTM;
    END
  ELSE IF message is 'YES' THEN
     BEGIN
        record the message into the state queue;
        send the message 'YES' back to the GTM;
      END
get a message from the GTM
IF message is ABORT THEN
  BEGIN
      record the message into the state queue;
      send the message 'NO COMPLETED' back to the GTM;
    END
  ELSE IF message is READY THEN
```

```
    BEGIN
       record the message into the state queue;
       send the message 'COMPLETED' back to the GTM;
     END
END
```

The GTM allocates one server (LTM) to a global transaction for each of the sites referenced by the transaction. A server allocated to a transaction is not released until the transaction has completed execution at each site and the results of the transaction have been committed or aborted by the MDBS.

The global transaction diagram by the proposed new protocol is shown in Figure 6. The GTM sends the global subtransactions to the appropriate servers. If a server is not allocated to a global transaction for a particular site, the GTM allocates a server to the transaction and passes the global subtransactions to the appropriate servers for execution.

When a global transaction completes execution, the GTM instructs the servers allocated to the transaction, to commit the update to the local databases. The MDBS uses the proposed new protocol in communication between the GTM and the LTMs to commit the results of a global transaction. For

example, consider GT has data item x, y, and LT has a data item x.

GT: r(x), w(x), w(y);

LT: r(x) w(x);

GT has a reservation on x and reads x. Then, it puts a write-lock on x during LT reads x. The scheduler calls rw synchronization in this case, so w(x) on GT is blocked until it finished its processing, then r(x) on LT is processed. If we look at the other example such as GT: r(x), r(y); LT: r(y); . Since GT and LT put reservations on data x and y, not locks, there is no conflict, and this new protocol shows more improved degree of concurrency control than Thomas' Write Rule with 2PL because Thomas' Write Rule with 2PL needs read-locks on data x and y, which the proposed protocol does not need.

The proposed new protocol ensures MDBS- serializability and autonomy of component LDBSs. Let $x \in gt_i$ and $y \in gt_j$ and x and y conflicts such that $x <_{GH} y$ where GH is global database history. This means that there is a γ-arc gt_i -> gt_j in MSG(MH) in page 28. Therefore, by the proposed new protocol, gt_i precedes gt_j in any topological sort of GH. So, all operations of gt_i precede all operations of gt_j. Thus, serializability is ensured.

Also, the LDBSs are not required to inform the global concurrency controller about the local transactions executed at the local sites. MDBSs transactions are scheduled by

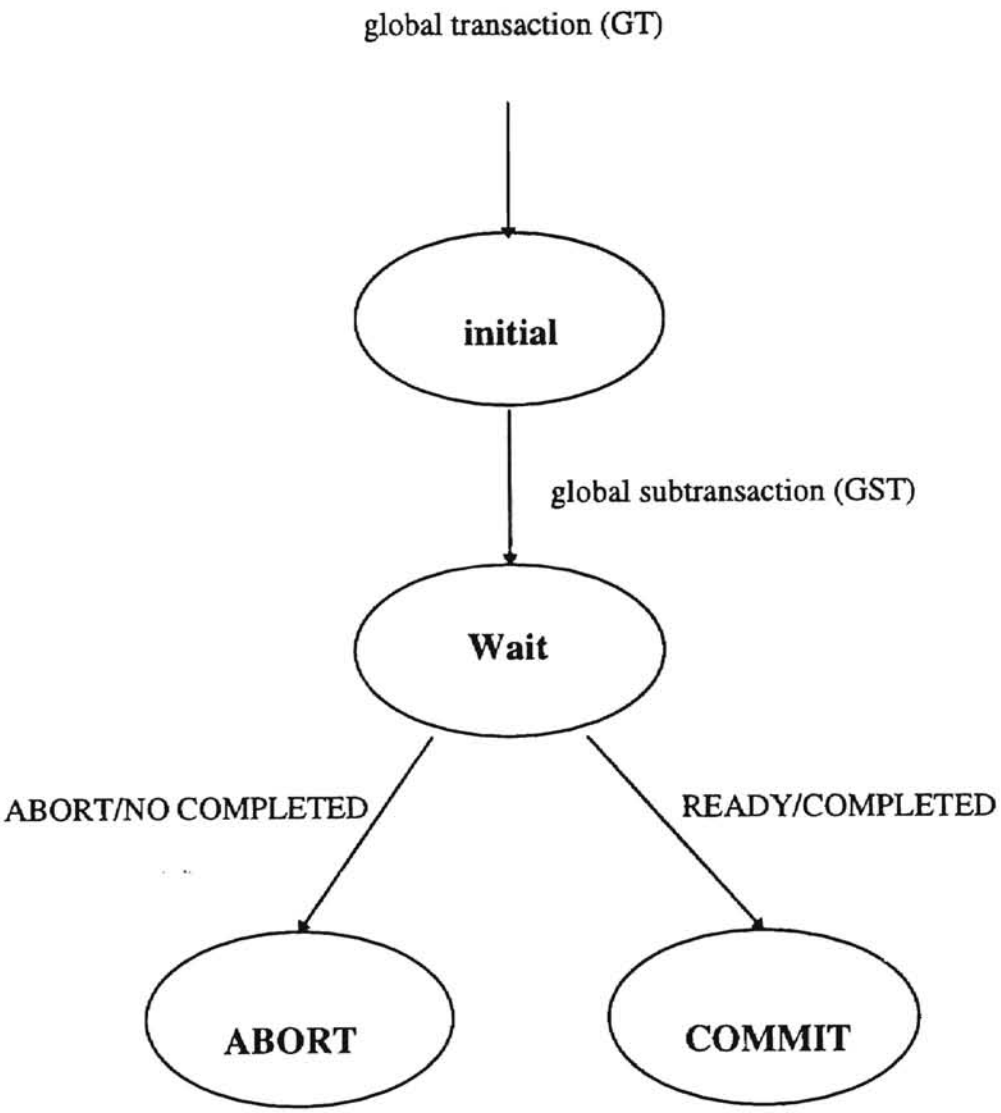

Figure 6. Global Transaction Diagram by the New Protocol.

getting information about which sites contain the data items to be accessed by the global transactions, and unaware of the local transactions, so no modification in the existing LDBMS is demanded by MDBS transactions, and the MDBS does not require any specific commit protocol to be supported by the local DBMSs and assumes that any local DBMS is capable of properly committing the results of local transactions.

After the servers complete commit processing with the local DBMSs, the servers are deallocated from the global transaction and are returned to the pool of available servers. The local scheduler concurrently executes all the transactions submitted to it. All the data items needed for operation of a transaction is checked for availability. If available, the transaction puts appropriate lock (reservation) one the data and accesses it. Theses locks (reservations) are released only after the completion of the transaction.

Comparison of Major Approaches in MDBS

Algorithm	Global Execution Correctness	Local Autonomy	Degree of Concurrency
Quasi-serializability	Guaranteed	Preserved	Low
Distributed cycle detection algorithm	Guaranteed	Not preserved	High
GCC algorithm used in super databases	Guaranteed	Not preserved	High
Optimistic algorithm	Guaranteed	Preserved	Low
Altruistic locking algorithm	Guaranteed	Preserved	Low
Proposed method	Guaranteed	Preserved	High

CHAPTER IV

SUMMARY AND FUTURE WORK

Summary

Advanced databases are widely used nowadays. The particular advanced database, which is multidatabase, without any compromise to its local autonomy, will increase the usability of the heterogeneous distributed database system. Multidatabase is one of the very active database research areas. The problem of managing heterogeneous distributed databases is becoming an increasingly difficult problem due to an ever increasing number of different DBMSs utilized in many corporations. Many retrieve-only MDBSs have been developed that attempt to provide a tool for managing heterogeneous distributed data sources.

In the lierature review chapter we saw several models for distributed control of heterogeneous distributed database system. A multidatabase concurrency control mechanism based on the new protocol concurrency control mechanism was proposed as a solution for the problem of indirect orders between global transactions due to local

transactions, still preserving local autonomy and ensuring global serializability. The degree of concurrency is improved and it ensures serializability by maintaining the new protocol at all sites.

Future Work

Since the data becomes larger such as image data, the object-oriented method has become more important in order to realize the large data through the network system. The Object-oriented multidatabase is the new area which is considering large data as a object or just thing, and realizes inheritance. Object-oriented transactions are defined as open nested transactions. They can be realized in multi-layer transaction systems for open nested transactions. Thus, future investigation is needed in this area.

REFERENCES

[Alonso at al. 87] R. H. Alonso, Gracia-Molina, and, K. Salem, "Concurrency Control and Recovery for Global Procedures in Federated Database System", *Data Engineering Bulletin*, Vol. 10, No. 3, pp. 5-11, 1987.

[Baker 90] K. Baker, "Transaction Management on Multidatabase Systems", *Ph.D. Dissertation and Technical Report, Department of Computer Science*, University of Alberta, Alberta, CA 1990.

[Barghouti et al. 91] N. S. Barghouti, S. Aser, and G. E. Kaiser, "Concurrency control in Advanced Database Applications", *ACM Computing Surveys*, Vol. 23, No. 3, pp. 269-317, September 1991.

[Bernstein 87] P. A. Bernstein, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Co., Reading, MA, 1987.

[Bernstein and Goodman 81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 13, No.2, pp. 185-221, June 1981.

[Booth 81] G. M. Booth, *The Distributed System Environment*, New York: McGraw-Hill, Inc., 1981.

[Brietbart and Silberschatz 87] Y. Brietbart and A. Silberschatz, "An Update Mechanism for Multidatabase Systems", *Data Engineering*, Vol. 10, No. 3, pp. 12-18, September 1987.

[Brietbart and Silberschatz 88] Y. Brietbart and A. Silberschatz, "Multidatabase Update Issues", *Proceedings of ACM SIGMOD International Conference on Management of Data*, Vol. 17, No. 3, pp. 135-142, September 1988.

[Brietbart et al. 90] Y. Brietbart, A. Silberschatz, and G. R. Thompson, "Reliable Transaction Management in a Multidatabase System", *Proceedings of ACM SIGMOD International Conference on Management of Data*, Vol. 5, No. 8, pp. 215-224, September 1990.

[Bukhres and Elmagarmid 95] O. A. Bukhres and A. K. Elmagarmid, *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*, Prentice-Hall,. Inc. Englewood Cliffs, NJ, 1996.

[Ceri and Pelagati 84] S. Ceri and G. Pelagati, *Distributed Database Principles and Systems*, McGraw-Hill, Inc., 1984.

[Chikkanna 94] K. H. Chikkanna, "Concurrency Control in Multidatabases", *M. S. Thesis*, Department of Computer

Science, Oklahoma State University, 1994.

[Darcy and Boston 83] L. Darcy and L. Boston, *Webster's New World Dictionary of Computer Terms*, New York: Simon and Schuster, Inc., 1983.

[Du and Elmagarmid 89] W. Du and A. K. Elmagarmid, "Quasi serializability: a correctness criterion for global concurrency control in InterBase", *Proceedings of 13th International Conference on Very Large Database*, pp. 347-355, August 1989.

[Eliezer et al. 91] L.Eliezer, Henry F. Korth, and A. Silberschatz, "An Optimistic Commit Protocol for Distributed Transaction Management", *ACM Transaction on Database Systems*, Vol. 12, No. 4, pp. 88-97, 1991.

[Elmagarmid and Du 96] A. K. Elmagarmid and W. Du, "A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems", *Sixth International Conference on Data Engineering, IEEE Computer Society*, February 5-9, 1996.

[Gafni and BapaRao 92] A. Gafni and K. V. BapaRao, "A Time-based Distributed Optimistic Recovery and Concurrency Control Mechanism", *IEEE Computer Society*, Vol. 6, No. 3, pp. 498-505, 1992.

[Giligor and Popescu-Zeletin 86] V. Giligor and R. Popescu-Zeletin, "Transaction Management in Distributed

Heterogeneous Database Management Systems", *Information Systems*, Vol. 11, No. 4, pp. 287-297, 1986.

[Heimbigner and McLeod 85] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management", *ACM Transactions on Office Information Systems*, Vol. 2, No, 8, pp. 253-278, July 1985.

[Herlihy 90] M. Herlihy, "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types", *ACM Transactions on Database Systems*, Vol. 15, No. 1, pp. 96-124, March 1990.

[Kung and Robinson 81] H. T. Kung and Jone T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transaction on Database Systems*. Vol. 6, No. 2, pp. 213-226, June, 1981.

[Leu and Elmagarmid 90] Y. Leu and A. K. Elmagarmid, "A Hierarchical Approach to Concurrency Control in Multidatabases", *IEEE Transactions on Database Systems*, Vol. 5, No. 4, pp. 202-210, 1990.

[Mahesh 90] M. J. Ram Mahesh , "Supporting Altruistic Protocol in Multidatabase System", *M. S. Thesis*, Department of Computer Science, Oklahoma State University, 1990.

[Mehrotra et al. 92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz, "The Concurrency

Control Problem in Multidatabases: Characteristic and Solutions", *Proceeding of ACM SIGMOD International Conference on Management of Data*, pp. 288-297, 1992.

[Muth et al. 92] Peter Muth, Wolfgang, and E. J. Neuhold, "How to Handle Global Transactions in Heterogeneous Database Systems", *Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, IEEE Computer Society Technical Committee on Data Engineering*. February 2-3, 1992.

[Ozsu and Valduriez 91] M.T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[Soparkar et al. 91] N. Soparkar, H. F. Korth, and, A. Silberschatz, "Failure Resilient Transaction Management in Multidatabases", *IEEE Computer*, Vol. 9, No. 12, pp. 28-36, December 1991.

[Thompson 87] G. R. Thompson, "Multidatabase Concurrency Control" *Ph.D Dissertation*, Department of Computer Science, Oklahoma State University, 1987.

[Veijalainen and Wolski 92] J. Veijalainen and A. Wolski, "Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases", *IEEE Computer Society*, Vol. 11, No. 2. pp. 470-479, 1992.

APPENDIX A:

Glossary

Conflict : Two operations conflict if their order of execution affects either the state of the database or the value that one of them returns. In the Read-Write model, two operations conflict if they operate on the same data item and at least one of them is a Write [Bernstein 87].

Consistent state : A state of the database that satisfies the database's consistency predicates. Intuitively, this means that data item values are internally consistent with each other [Bernstein 87].

Database System : A collection of hardware and software modules that support database operations and transaction operations [Bernstein 87].

Data Manager (DM) : A composite module of the database system, consisting of a cache manager and recovery manager [Bernstein 87].

Distributed Database System : A collection of sites connected by a computer network, where each site is a centralized database system that stores a portion of the database [Bernstein 87].

Partial order : A partial order L = (Σ , <) consists of a set Σ called the domain of the partial order and an irreflexive, transitive binary relation < on Σ [Bernstein 87].

Prepared State: It is a state of a transaction in which the subtransaction finishes all of its read and computation operations and has all of its updates stored in a stable storage. Such transaction is ready to commit or abort according to a global decision [Leu and Elmagarmid 90].

Restriction: A set P1 with a partial order $<_{p1}$ on its elements is a restriction of a set P2 with a partial order $<_{p2}$ on its elements if P1 $\subset$ P2, and for all e1, e2 $\in$ P1, e1 $<_{p1}$ e2 if and only if e1 $<_{p2}$ e2 [Mehrotra et al. 92].

rw Synchronization : Controlling the order in which Reads execute with respect to conflicting Writes [Bernstein 87].

Scheduler : By delaying or rejecting some of those operations, scheduler is the database system module that controls the relative order in which database operations and transaction operations execute [Bernstein 87].

Serial Execution: For every pair of transactions, all of the operations of one transaction execute before any of the operations of the other [Bernstein 87].

Serializability: An execution is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions [Bernstein 87].

Serialization Order: Partial order of all operations in the execution [Leu and Elmagarmid 90].

Transaction Manager (TM) : The database system module that is the interface between transactions and the rest of the database system. It receives each operation from the transaction, performs any necessary preprocessing of the operation, and then forwards the operation to the appropriate database system module [Bernstein 87].

Two Phase Locking : The locking protocol in which each transaction obtains a read (or write) lock on each data item before it reads (or write) that data item, and does not obtain any locks after it has released some lock [Bernstein 87].

ww Synchronization : Controlling the order in which conflicting Writes execute [Bernstein 87].

APPENDIX B:

Abbreviations and Acronyms

2PC	Two Phase Commit
2PL	Two Phase Locking
DBMS	Database Management System
DDBMS	Distributed Database System
GH	Global Database History
GS	Global Scheduler
GSH	Global Serializability Graph
GT	Global Transaction
GTM	Global Transaction Manager
GWFG	Global Wait-for Graph
LDB	Local Database
LDM	Local Data Manager
LH	Local Database History
LS	Local Scheduler
LTM	Local Transaction Manager
LWFG	Local Wait-for Graph
MDBMS	Multidatabase Management System
MDBS	Multidatabase System

MH	Multidatabase History
MSG	Multidatabase Serializability Graph
rw synchronization	read-Write Synchronization
TO	Timerstamp Ordering
TWR	Thomas' Write Rule
WFG	Wait-for Graph
ww synchronization	Write-Write Synchronization

VITA

YONGHO AN

Candidate for the Degree of

Master of Science

Thesis: A NEW PROTOCOL FOR MULTIDATABASE CONCURRENCY CONTROL

Major Field: Computer Science

Biographical:

Person Data: Born in Seoul, Korea, on March 3, 1970.

Education: Received Bachelor of Science in Computer Science from the Ohio State University, Columbus, Ohio, in August 1994; completed the requirement for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 1997.

Experience: Consultant, Department of Computer Science, the Ohio Sate University, January 1993 to December 1993; Teaching assistant, Department of Computer Science, Oklahoma Sate University, August 1995 to May 1997.

Professional Membership: Korean-American Scientists and Engineering Association.