COPING WITH VALUE DEPENDENCY FOR FAILURE RECOVERY IN MULTIDATABASE SYSTEMS

YONGMEI SUN

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Coping with Value Dependency for Failure Recovery in Multidatabase Systems

by

Yongmei Sun, BSc.

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

November 1997

St. John's Newfoundland

Abstract

Local autonomy is the main impediment to achieving failure atomicity in a multidatabase system since it allows a local database to unilaterally commit or abort a subtransaction. Compensating a committed subtransaction is in general hard to realize due to the complication arising from the propagation of the committed effects. Resubmitting an aborted subtransaction is more realistic since the problems arising from inter-subtransaction dependencies are more predictable than those from propagation of committed effects. However, if such a dependency is cyclic or if it not only involves values but also data items, then the problem becomes more complicated. In this thesis, a failure recovery scheme[1] using resubmission is proposed. The scheme is based on distinguishing the subtransactions into two different types, and employing different strategies for them. As a result, the scheme allows an aborted subtransaction to be restarted. Compared with other failure recovery schemes which also do not rely on compensation, the scheme compromises local autonomy to a lesser extent. In this thesis, different kinds of dependencies are also studied, their impact on the correctness of resubmission method discussed and solutions proposed.

[1] A preliminary version of the scheme was presented in International Conference on Data and Knowledge Systems for Manufacturing and Engineering,Hong Kong, pp 297-306,1994.

This thesis is dedicated to

my mother, ChuanShen Chu

and my father, DaZhen Sun

and my husband, John Yang

for their support and encouragement throughout

the course of my education

Acknowledgements

I wish to express my thanks to my supervisor Dr. Jian Tang for his guidance, interest, suggestion and enthusiasm. Without his help, it would be impossible to give this thesis its current quality.

I would like to thank the system support staff for providing help and assistance while I conducted this research.

I am also very grateful to the administrative staff who have helped in one way or another in the preparation of this thesis.

In addition, I would like to acknowledge the financial support received from the Department of Computer Science and the School of Graduate Studies.

Special thanks are due to my fellow graduate students Zhengqi Lu, Xu He, Zhiming Shi and Chen Hao for their valuable comments and useful suggestions.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Many of today's database systems share information in an organization-wide basis. These database systems are usually developed independently of each other. As a result, they may be *heterogeneous*, indicating the use of different structures, data models, control policies, etc. They may also maintain *local autonomy*, meaning the relationship among them is not coordinator-subordinator oriented. In other words, individual database systems have the freedom of not being controlled by the others.

Sharing information among heterogeneous and autonomous database systems is a complicated task. The complication arises from the fact that it is generally required that both heterogeneity and local autonomy of the individual databases be preserved. A multidatabase approach provides to users a uniform interface by integrating the database systems. This has the advantage that the users view the collection of the databases as a single and powerful database, and therefore are free from the burden of handling various problems caused by heterogeneity and local autonomy.

A multidatabase system (MDBS) is a collection of several databases. An MDBS creates the illusion of a single database system. It allows users to manipulate data contained in the various databases without modifying current applications and without migrating the data to a new database. The MDBS hides from users the intricacies of different DBMS's and different access methods. It provides uniform access to pre-existing databases without requiring the users to know either the location or the characteristics of different databases and their corresponding DBMS's. The MDBS query and data manipulation languages allows users to access multiple pre-existing databases in a single query or application.

A multidatabase is divided logically into two levels, global and local. At the global level is a multidatabase management system (MDBMS) which among other things is responsible for maintaining data consistency across local databases. At the local level is a set of local database management systems (LDBMS), one for each site. A LDBMS ensures data consistency within the corresponding local database.

A multidatabase user requests service through a global transaction, and a local database user through a local transaction. A transaction is simply a sequence of read and write operations defined on a database. A global transaction is a transaction that is submitted to the MDBMS and is executed under the MDBMS control. A local transaction, on the other hand, is a transaction submitted to a local DBMS, outside of the MDBMS control.

It is difficult to design a general multidatabase system that is both correct and

efficient in all cases. In this respect, local autonomy posts many difficult problems. Local autonomy exists in different forms. One aspect of autonomy is the right of every node to commit or abort a transaction at any time.

In this thesis, serializability and atomicity are used to be the criteria for ensuring consistency. Serializability requires that an execution of global and local transactions be equivalent to a serial execution of these transactions. Atomicity requires that a transaction either performs all its write operations or performs none of them. Atomicity is the goal of most failure recovery schemes.

A MDBMS can be thought of as containing two logically separate components, a global concurrency controller (GCC) and a global recovery manager (GRMGR). When a global transaction is submitted to the system, the GCC first schedules the execution of each subtransaction of the global transaction to ensure serializability. The GRMGR is responsible for ensuring atomicity of global transactions.

Most of the work in the area address only concurrency control ignoring failures. The example of concurrency control methods are the site graph method [6], the altruistic locking [28], the cycle detection method [29], the optimistic algorithm of [14], the integration method using observability and controllability [25], the superdatabases [27] and the top down approach [13].

In the current literature, several approaches have been suggested to handle failure recovery in a multidatabase system [5,8,20,34]. One approach tries to achieve true atomicity [5,8,34]. The price to be paid for that is compromising local autonomy to some extent. Another approach is based on the notion of

logical atomicity by compensation [20]. This approach does not compromise local autonomy, but is hard to realize in practical applications since it requires the use of compensating transactions. In a multidatabase system where different databases exhibit heterogeneous and autonomous behaviors, compensating a transaction whose effects have been propagated to the other databases is an extremely difficult task in the general case.

The difficulties in achieving failure atomicity in a multidatabase system are mainly due to two factors. One is that different databases are allowed to commit or abort a transaction unilaterally. The other is the fact that usually there exist various dependencies, aggregately called value dependencies (henceforth simply called dependencies), between different operations of a transaction. If the inter-dependent operations are executed at different databases and some commit while the others abort, then we must either undo the committed or redo the aborted operations. Undoing the committed operations, as mentioned before, is difficult. Redoing the aborted operations, on the other hand, may be undermined by the inter-dependencies between the committed and the aborted operations.

The value dependencies among different subtransactions of the same global transaction may be input/output oriented or dialogue oriented. In input/output oriented dependency, the input of a subtransaction is generated from the output of some other subtransactions which have finished successfully. In dialogue oriented dependency, the write operation of a subtransaction depends on the values of the read operations of other subtransactions. These must be submitted to the

global site even before subtransactions terminate which then forwards them to execute the dependent write operation. Dialogue oriented dependency occurs in those applications where data at different local sites are related by global constraints and therefore requires close interaction among subtransactions. As a result, dialogue oriented dependency requires that the operations in individual subtransactions be coordinated in terms of their execution order. Note that this coordination is inevitable in any concurrency control mechanism which deals with applications where dialogue oriented dependency exists. On the other hand, the existence of dialogue oriented dependency complicates the design of concurrency control and failure recovery protocols. This issue has been studied in several works [14,31,32,33,34]. Another point worth noting is that to preserve dialogue oriented dependency does not contradict execution autonomy (refer to Section 2.2), since each local site has the freedom to choose to abort or commit a subtransaction.

Dialogue oriented dependency may or may not form a cycle. In [34], the authors note that if it does not form a cycle, then some operations of a multidatabase transaction can be committed in an ordered fashion toward failure atomicity. However, if it forms a cycle, the authors suggest using two different transactions to encompass the operations of a multidatabase transaction at a single site. This method may not be feasible if there exist direct dependencies among the operations at a single site which belong to the same multidatabase transaction.

In this thesis, a method is proposed for failure recovery in a multidatabase system where dialogue oriented dependency exists. The method achieves fail-

ure atomicity by properly ordering the commit operations based on the value-dependencies existing in a multidatabase transaction. To resolve the problems caused by cyclic dependency, restrictions are put on a few operations but no restrictions on the others. With this treatment the idea of commit order can still be used in the face of cyclic dependency and at the same time minimize the loss of local autonomy. The proposed protocol is feasible without relying on the way the operations of a transaction depend on each other and does not need expensive compensate transactions. Different types of dependencies are also studied and the impact which they have on the commitment protocols are analyzed.

An expense that is paid by the proposed commitment protocol is that local autonomy is compromised to some extent. However, as will be explained in Section 8.2, such compromises are justified.

The rest of this thesis is organized as follows. In Chapter 2, we discuss the issue of concurrency control in MDBS as well as a special method, 2PL. In Chapter 3, we describe general problems of failure recovery. In Chapter 4, we survey some of the recent research in the area of atomic transaction commitment. In Chapter 5, we first discuss a transaction processing model in a multidatabase system, and then give a description of the essential concepts related to value dependency. In Chapter 6, we discuss the proposed commitment protocol in detail. In Chapter 7, we give a more thorough examination of value-dependency, relax some assumptions we made at Chapter 5 and present the solutions. In Chapter 8, we discuss various

issues related to the protocol, such as performance, local autonomy and implementation. We conclude the thesis by summarizing the main results.

Chapter 2

Concurrency Control of Multidatabase Systems

In this thesis, we concentrates on the issue of failure recovery of multidatabase systems. Our commitment protocol is under the assumption that we use a special concurrency control mechanism, two phase locking (2PL). Since failure recovery is closely related to the concurrency control issue, we discuss the concurrency control problem in this chapter. First, we introduce the basic concept of the concurrency control problem and then we discuss traditional approaches to solve this problem. The multidatabase concurrency control issue is also addressed. We put emphasis on two phase locking mechanism as it is used in our context of failure recovery.

2.1 Traditional Approaches to Concurrency Control

In a database system, several users may read and update information concurrently. Undesirable situations may arise if the operations of various user transactions are improperly interleaved. Concurrency control is an activity that coordinates

concurrently executing operations so that they interleave with each other in an acceptable fashion.

Most traditional approaches follow one of three main approaches to concurrency control:two phase locking (the most popular example of locking protocols), timestamp ordering, and optimistic concurrency control. Some mechanisms add multiple granularities of locking and nesting of transactions. In this section, we give a detailed description of 2PL mechanism as well as strict two phase locking mechanism as they are used in our context of failure recovery.

The idea behind locking is intuitively simple. Each data item has a lock associated with it. Before a transaction T_1 may access a data item, the scheduler first examines the associated lock. If no transaction holds the lock, then the scheduler obtains the lock on behalf of T_1. If another transaction T_2 holds the lock, then T_1 has to wait until T_2 gives up the lock. That is, the scheduler will not give T_1 the lock until T_2 releases it. The scheduler thereby ensures that only one transaction can hold the lock at a time, so only one transaction can access the data item at a time.

Locking can be used by a scheduler to ensure serializability. To present such a locking protocol, the following notation is used.

Transactions access data items either for reading or for writing them. We therefore associate two types of locks with data items: read locks and write locks. Here rl_i[x] (or wl_i[x]) is used to indicate that transaction T_i has obtained a read (or write) lock on x. We use the letters o, p, and q to denote an arbitrary type of

operation, that is, a Read(r) or Write(w). We use ol_i[x] to denote a lock of type o by T_i on x.

Two locks pl_i[x] and ql_j[y] *conflict* if $x = y$, $i \neq j$, and operations p and q are of conflicting types. Two locks conflict if they are issued by different transactions, and one or both of them are write locks. Thus, two locks on different data items do not conflict, nor do two locks that are on the same data item and are owned by the same transaction, even if they are of conflicting types.

We use ru_i[x] (or wu_i[x]) to denote the operation by which T_i release its read (or write) lock on x. In this case, we say T_i unlocks x (the u in ru and wu means unlock).

Here are the rules according to which a basic 2PL scheduler manages and uses its locks:

1. When it receives an operation p_i[x], the scheduler tests if pl_i[x] conflicts with some ql_j[x] that is already set. If so, it delays p_i[x], forcing T_i to wait until it can set the lock it needs. If not, then the scheduler sets pl_i[x] , and then sends p_i[x] to execute.

2. Once the scheduler has set a lock for T_i, say pl_i[x], it may not release that lock at least until after the corresponding operation p_i[x] has been processed.

3. Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item).

Rule 1 prevents two transactions from concurrently accessing a data item in

conflicting modes. Thus, conflicting operations are scheduled in the same order in which the corresponding locks are obtained. Rule 2 supplements Rule 1 by ensuring that operations on a data item are processed in the order that the scheduler submits them. Rule 3 guarantees that all pairs of conflicting operations of two transactions are scheduled in the same order.

Almost all implementations of 2PL use a variant called Strict 2PL. This differs from the Basic 2PL scheduler in that it requires the scheduler to release all of a transaction's locks only when the transaction terminates.

There are two reasons why a strict 2PL is necessary in practical applications. First, consider when a 2PL scheduler can release some $ol_i[x]$. To do so the scheduler must know the following:

1. T_i has set all of the locks it will ever need, and

2. T_i will not subsequently issue operations that refer to x.

One point in time at which the scheduler can be sure of 1 and 2 is when T_i terminates, that is, when the scheduler receives the c_i or a_i operation. In fact, in the absence of any information, this is the earliest time at which the scheduler can be assured that 1 and 2 hold.

A second reason for the scheduler to keep a transaction's locks until it ends, and specifically until after the transaction's Commit or Abort is processed, is to guarantee a strict execution [16].

Executions are called strict when they satisfy the condition that both Reads

and Writes for x are delayed until all transactions that have previously written x are committed or aborted.

Strict histories have nice properties. For this reason, 2PL implementations usually take the form of Strict 2PL scheduler, rather than the Basic 2PL schedulers.

2.2 The Problem of Concurrency Control in Multidatabase Environments

The problem of concurrency control in multidatabase environments is different from that in traditional distributed database systems. Furthermore, most efforts attempting to generalize the classical concurrency control strategies for multidatabase systems are only partially successful. For example, many concurrency control protocols proposed for MDBSs either violate local autonomy or do not maintain global serializability.

Designing a concurrency control strategy for a heterogeneous database environment is more difficult than in its homogeneous counterpart, primarily because we must deal not only with the data distribution but also with heterogeneity and autonomy of underlying databases. In a homogeneous distributed database system, local database management systems use the same concurrency control stratagem and the global concurrency controller has access to all information it needs to produce and/or certify the schedules. In addition, the global concurrency controller normally has control over all transactions running in the system.

In contrast, in a multidatabase systems, we must deal with the following problems caused by autonomy of the local systems:

1. Local concurrency controllers are designed in such a way that they are totally unaware of other LDBSs or of the integration process. This type of autonomy is defined as design autonomy and indicates that each of the LDBSs is free to use whatever algorithms it wishes. When this LDBS is incorporated into a multidatabase system, design autonomy specifies that we cannot retrofit its algorithms.

2. The Global Concurrency Controller (GCC) needs information regarding local executions in order to maintain global database consistency. However, the GCC has no direct access to this information and can not force the Local Concurrency Controllers (LCCs) to supply it. This type of autonomy is defined as communication autonomy which means that an LCC is allowed to make independent decisions as to what information to provide.

3. LCCs make decisions regarding transaction commitments based entirely on their own considerations. LCCs do not know or care whether the commitment of a particular transaction will introduce global database inconsistency. In addition, a GCC has no control over LCCs at all. For example, a GCC can not force an LCC to restart a local transaction even if the commitment of this local transaction will introduce global database inconsistency. We call this type of autonomy the execution autonomy; it says that each of the LCCs is free to commit or restart any transaction running under its control.

There has been a flurry of research to develop new approaches to transaction management that meet the requirements of consistency of multidatabase system. In the following section, some of them are discussed.

2.3 A Brief Review of Existing MDBS Concurrency Control Techniques

Breitbart proposed a global concurrency control protocol based on site graphs [6]. The protocol works as follows. Before a global transaction is submitted, the GCC analyzes its read and write operations, trying to select some sites to execute them without creating cycles in the site graph. The acyclicity of the site graph will guarantee the correctness (serializability) of the global execution. This algorithm preserves local autonomy, and does not abort any global transactions. The problem with this algorithm, however, is that it allows a low concurrency degree for global transactions. One observation is that a global transaction with operations at all sites blocks the execution of other global transactions until it is committed. Another observation is that no two global transactions can access multiple sites concurrently.

The altruistic locking algorithm [28] is a lock based algorithm. The locking granularity is that of a local site. In other words, a local site can execute at most one global subtransaction and many local transactions at a time. In order to access a local database, the global procedure has to lock the local site and then issue a global subtransaction to the local transaction manager of that site. The

locking or releasing of local sites must follow specific rules. This algorithm has the following two problems: First, since the granularity is a site, the degree of concurrency for the global procedures is very low. Second, since the effect of the local transactions is not considered, the global serializability is not guaranteed.

The optimistic algorithm [14] is based on a centralized controller and uses operating system robust processes (STUBs). One of the key ideas is the STUB process. It ensures the successful execution of a global subtransaction even though it may get aborted or restarted repeatedly by the local concurrency controller. This algorithm does not violate local autonomy. However, because of the lack of consideration for local transactions, this algorithm may generate a non-serializable schedule.

The basic idea of superdatabases algorithm [27] is as follows. Every LDBS reports to the GCC the serialization order, o-element, of each global subtransaction executed on it. The GCC uses these o-elements to construct an o-vector for each global transaction. It then validates the execution of a global transaction against the set of recently committed global transactions. It does this by trying to find a consistent o-vector position among the o-vectors of the recently committed global transactions attempting to commit. The problem is that it is unclear how the GCC could get these o-elements in an autonomous environment.

In the distributed cycle detection algorithm [29], each local site keeps a local serialization graph for the transactions executed on it. The local serialization graph is kept acyclic by the local concurrency controller. The GCC validates

the execution of a global transaction by invoking a distributed cycle detection algorithm to make sure that the commitment of this global transaction will not create a global cycle among the local serialization graphs. However, this algorithm violates local autonomy by requiring the local sites to keep the local serialization graphs for the GCC.

All of the protocols discussed above either violate local autonomy in a certain way, allow low concurrency degree, or fail to maintain global serializability.

2.4 Using 2PL to Achieve the Serializability of MDBS

As mentioned above, all of the protocols discussed above have their limitations. The common assumption made in the above protocols is that the local concurrency control mechanism of a participating database system is unknown. This can be accommodated only by the global concurrency controller with very low concurrency level. Additionally, with the notable exceptions of the trivial site graph method [6] and the top down approach [13], the methods require modifications to participating database management systems.

To ensure the correctness of our failure recovery scheme, each local LDBMS is required to use a strict two phase locking mechanism for concurrency control. Global concurrency controller is required to use the two phase locking mechanism. The requirement that each local LDBMS use the strict two phase lock mechanism is practical because most of the commercially available systems use the strict two

phase locking policy.

As concerned with the applicability of strict 2PL to distributed databases, it has been shown that, in the absence of failures, a global transaction scheduler using the distributed two phase locking protocol produces strict histories for global transactions [4]. It has been also proved that this is true for a mix of global and local transactions [7].

Chapter 3

General Problems of Failure Recovery

3.1 Concept of Failure Recovery

Computer systems fail in many ways. It is not realistic to expect to build DBSs that can tolerate all possible faults. However, a good system must be capable of recovering from the most common types of failures automatically, that is, without human intervention.

There are two types of failures that are most common in databases, known as *transaction failures*, and *system failures*. A transaction failure occurs when a transaction aborts. A system failure refers to the loss or corruption of the contents of volatile storage (i.e., main memory).

In a database system, if failure occurs and a transaction cannot be completed correctly, an abort operation is issued to the transaction. When a transaction aborts, the DBMS wipes out all of its effects. The prospect that a transaction may be aborted calls for the ability to determine a point in time after which the

DBMS guarantees to the user that the transaction will not be aborted and its effects will be permanent. The commit operation accomplishes this guarantee. Its invocation signifies that a transaction terminated "normally" and that its effects should be permanent. Executing a transaction's commit operation constitutes a guarantee by the DBMS that it will not abort the transaction and that the transaction's effects will survive subsequent failures of the system. A transaction that has issued its Start operation but is not yet committed or aborted is called active. A transaction is uncommitted if it is aborted or active.

The objective of failure recovery is to bring the database to a consistent state, removing effects of uncommitted transactions and applying missing effects of committed ones. To be more precise, define the *last committed value* of a data item x in some execution to be the value last written into x in that execution by a committed transaction. Define the *committed database state* with respect to a given execution to be the state in which each data item contains its last committed value. The goal of failure recovery is to restore the database into its committed state with respect to the execution up to the system failure.

The data recovery manager is primarily responsible for ensuring that the database contains all of the effects of committed transactions and none of the effects of aborted ones. The data recovery manager is normally designed to be resilient to failures in which the entire contents of volatile memory are lost. After a system failure, the only information the data recovery manager has available is the contents of stable storage. Since the data recovery manager never knows when a

system failure might occur, it must be very careful about moving data between volatile and stable storage.

Failure recovery of a distributed database system is more complicated than that of a centralized database system because of the following fact. A transaction T in a distributed system has operations in various sites of the distributed system. When the Commit operation of T comes, to which sites should this operation be forwarded? A Commit operation concerns all sites involved in the processing of T. The same is true for Abort. Thus, the processing of a logically single operation (Commit or Abort) must take place in multiple places in a distributed database system.

The problem is more subtle than it may appear at first. Having sent Commit operations to all other sites is not enough. It is possible that a Commit is sent, but a local site rejects it and aborts the transaction. In this case, if the transaction is distributed, it should abort at all other sites where it accessed data items.

In a distributed system, we can have partial failures, that is, some sites may be working while others have failed. We must ensure that a single logical action (Commit or Abort) is consistently carried out at multiple sites when partial failures occur.

The simplest and most popular algorithm that ensures this consistency is called two phase commit (2PC) protocol. It will be described in detail in the next section.

3.2 Two Phase Commit Protocol

Assuming no failure, the two phase commit protocol goes roughly as follows:

1. The coordinator sends a VOTE-REQ (i.e., vote request) message to all participants.

2. When a participant receives a VOTE-REQ, it responds by sending to the coordinator a message containing that participant's vote: YES or NO. If the participant votes No, it decides Abort and stops.

3. The coordinator collects the vote messages from all participants. If all of them were YES and the coordinator's vote is also Yes, then the coordinator decides Commit and sends COMMIT messages to all participants. Otherwise, the coordinator decides Abort and sends ABORT messages to all participants that voted Yes (those that voted No already decided Abort in step (2)). In either case, the coordinator then stops.

4. Each participant that voted Yes waits for a COMMIT or ABORT message from the coordinator. When it receives the message, it decides accordingly and stops.

The two phases of 2PC are the voting phase (step (1) and(2)) and the decision phase (step (3) and (4)). A participant's *uncertainty period* starts when it sends a YES to the coordinator (step (2)) and ends when it receives a COMMIT or ABORT (step (4)). The coordinator has no uncertainty period since it decides as

soon as it votes - with the knowledge, of course, of the participants' votes (step (3)).

In step (4), a participant p that voted Yes is waiting for a COMMIT or ABORT from the coordinator. At this point p is uncertain. In this case the participant must consult with other processes to find out what to decide. This consultation is carried out in a termination protocol (for 2PC).

2PC protocol is widely used in distributed database systems. However, the autonomy and heterogeneity of the local database systems that participate in a MDBS causes several new problems, so 2PC can not be used in a MDBS.

3.3 Problems in Multidatabase Recovery

The objective of multidatabase recovery is to maintain the atomicity and durability of global transactions in the presence of failure. Here multidatabase recovery from site and subtransaction failures is discussed. Subtransaction failures occur when subtransactions of global transactions are unilaterally aborted by the LDBSs (e.g., to resolve local deadlocks). While many of the recovery principles used in distributed database systems can be applied in multidatabase systems, the autonomy and heterogeneity of the local database systems that participate in a MDBS causes several new problems that do not exist in other distributed database systems.

We say that a multidatabase transaction G is globally committed when it commits at the MDBS. To complete a globally committed multidatabase trans-

action G, the MDBS has to commit all its subtransactions at their LDBSs. A globally committed multidatabase transaction G becomes locally committed at LDBSi when the multidatabase system commits the subtransaction g_i of G at LDBSi. Multidatabase recovery must deal with the following problems which are due to the autonomy of the LDBSs:

1. The LDBSs cannot distinguish locally uncommitted subtransactions that belong to globally committed multidatabase transactions from uncommitted local transactions. When a LDBS comes up after a site failure, its local recovery procedures roll back all locally uncommitted subtransactions, even if they belong to globally committed multidatabase transactions.

2. MDBS recovery actions at each LDBS constitute new transactions. From the point of view of the LDBS, recovery transactions have no connection to the failed subtransactions they are supposed to complete.

3 Global transactions which have a locally committed subtransaction cannot be rolled back. The MDBS has either to complete the failed subtransactions of the globally committed transaction or compensate for the locally committed subtransacion.

Chapter 4

Literature Review of MDBS Failure Recovery

In this chapter, some approaches for failure recovery in multidatabase systems are reviewed, It will be shown that most solutions proposed either allow incorrect results or place severe restrictions on global and local transactions.

4.1 2PC Agent Method

In distributed database systems, the database consistency is attained by means of the basic two-phase commit protocol (2PC) or its variations and related recovery protocols. In the basic 2PC scheme, a coordinator responsible for the transaction commitment communicates with participants executing the operations. It is typical of the scheme that every participant has to move a subtransaction to a recoverable prepared state before the transaction is finally committed. In this state the unilateral aborts are no longer allowed at a participating DBMS, albeit that they are allowed before it.

Systems supporting the prepared state are called two-phased DBMSs, whereas

systems without an appropriate 2PC interface will be called single-phased DBMSs. It is evident that if arbitrary single-phased DBMSs are used as participating DBMSs and, additionally, submission of local transactions is allowed, then the objective to guarantee database consistency cannot be met in general. The fact that most of the existing systems are single-phased, and thus neither support the prepared state nor have an external interface for participating in the 2PC protocol, is a major obstacle in the way of heterogeneous DBMS integration.

Wolski and Veijalainen in [32] addressed the problem of failure recovery in multidatabase system assuming a MDBS in which the participating LDBSs use two phase locking (2PL) and permit only rigorous [9] schedules. They suggest a system as follows: The coordinator decomposes global transactions into global subtransactions, submits the corresponding commands to the Participating Sites and returns the results to the application. Upon receiving the global Commit, the Coordinator starts the distributed commitment procedure according to the basic 2PC protocol. The participant role is played by the 2PC Agent (2PCA) modules (unless there is a two-phased DBMS).

The important assumptions about the 2PC Agent interface described in [32] are:

1. Upon receiving the Prepare command, the 2PC Agent votes, e.g. it responses either with the REFUSE or READY message; In the latter case the corresponding global subtransaction enters the prepared state, meaning that it may either become committed or aborted at the Coordinator's subsequent request, de-

spite certain failures that might have occurred at the Participating Site in the meantime.

2. For any subtransaction in the prepared state, the 2PC Agent has to accept and confirm the COMMIT/ABORT message,i.e. has to execute the corresponding Commit/Abort command. Thus, it does not have execution autonomy with respect to these messages.

The scheme of this method is based on the idea of subtransaction resubmission, which is a repeated execution of all the commands belonging to a global subtransaction when a corresponding local subtransaction had been aborted by the local database. A transaction resubmission results in a new local subtransaction. In the process of recovery, the 2PCA may generate many local subtransactions for a given global subtransaction. Then, all but the last one (in the history) are in the aborted state. The last one may be incomplete, aborted or committed.

4.2 Variation of 2PC Protocol Using Prepared to Commit State

The basic requirement to develop a variation of the 2PC protocol for a multidatabase system is the availability of a visible prepared to commit state for all sutransactions of global transactions [22]. A subtransaction enters its prepared to commit state when it completes the execution of its operations and leaves this state when it is committed or aborted. Only when a transaction is committed, can its updates be observed by other transactions. The prepared state is visible if

the primitives the local DBMS provides in its interface allow the MDBS to decide whether the sutransaction should commit or abort.

Many DBMSs support a visible prepared to commit state (e.g., SYBASE) and can directly participate in a multidatabase 2PC. However, there are DBMSs which do not provide a prepared state or a corresponding primitive in their interface.

An approach which does not require the modification of the LDBSs is to simulate the prepared to commit state [33]. According to this approach, the MDBS has to determine whether all operations issued by the subtransaction have been successfully completed. This can be accomplished as follows:

1. Many DBMSs designed using the client-server architecture provide primitives to request the status of outstanding operations. For example, the Remote Database Access (RDA) standard and the DBMSs that comply with it provide an inquire operation which can be used by the MDBS to determine whether all operations of a subtransaction have been completed.

2. The MDBS may force a handshake after each operation. According to this approach, the MDBS submits the operations of each subtransaction once at a time and waits for the completion of the previous database operation before it submits the next one. Subtransactions are reduced to collections of totally ordered operations.

However these approaches only have limited application domain since not all DBMS applications provide such primitives.

4.3 Excluding Local Transaction

Centralized and distributed database recovery has been under extensive investigation. Solutions used for these environments are not directly applicable to multidatabase systems. The difficulty results from the autonomy of individual DBMSs which only know how to recover their own local databases. Individual DBMSs cannot distinguish between local transacions and transactions which are a part of a large global transaction that must be recovered so that multidatabase consistency is maintained.

The typical recovery procedure in a centralized DBMS requires the following:

1. Restart the DBMS (upon instructions from the operating system).

2. Recover the database by using information kept in the stable database log.

3. Open the DBMS for user access, that is, permit new transactions to be submitted.

4. Terminate the local restart process.

Ken Barker and M.Tamer Ozsu in [5] proposed that the local recovery process can be revised to accommodate the recovery of the global transactions. A method must be devised to inform the MDBS about the fate of the global subransactions. This is accomplished by modifying Step 2 described above as follows:

2a. Recover the database by using information kept in the stable database log.

2b. Open the database so that the MDBS has exclusive access.

2c. Establish a handshake with the MDBS to notify it that the database is recovered. Wait until the MDBS responds and recovers the global subtransactions.

2d. MDBS relinquishes exclusive access.

This modification permits the MDBS to recover global transactions by resubmitting all GSTs that were ready to commit at the time of the failure. The MDBS must maintain its own log for global transactions to perform this recovery and that log is called the global log. Information in the global log enables MDBS to determine the global subtransaction's current state and which global transactions are still active.

Concerning the violation of local autonomy, they argue as follows: When the DBMS has failed and is being recovered, it is not operational nor is it acting autonomously since the recovery is being performed in consort with the database administrator. The DBMS will not actually become operational, and therefore autonomous, until the local restart process terminates at Step 4. Since this is only a modification to the restart process involving interaction with the administrator it cannot be considered a violation of autonomy.

The revised recovery protocol raises two pragmatic issues. First, it is necessary to define the tasks performed by the MDBS when it has exclusive access. Secondly, it is necessary to demonstrate that establishing such a handshaking is feasible.

When the MDBS acquires exclusive access it determines the status of all global subtransactions at the time of the failure and resubmits any of those which were READY. Others which had not yet reached the READY state are aborted together with the global transactions of which they are a part. All of this is facilitated by the information recorded in the global log and by the simulation of the READY

state.

The practicality of establishing a handshaking between an autonomous DBMS and the MDBS can be demonstrated by noting that the DBMS views the MDBS as a user application. Therefore, the question becomes whether it is possible for a DBMS to restart and restrict the access to the database to only one user (or one class of users). They point out that their investigation into a number of commercial DBMSs (SYbase, Oracle, Ingres) has revealed that all of them provide means for establishing such exclusive access.

4.4 Failure Recovery by Compensating Transactions

In traditional data recovery, when a transaction is aborted for some reason, all the changes that it introduced are undone and the database is returned to the state that existed before the transaction began. This operation is called rollback. The concept of rollback is not applicable to multidatabase because multidatabase permits other transactions to change the same objects that its committed subtransactions have changed. Thus, it would not be possible to restore the database to its state before the aborted global transaction started without cascaded aborts of all the committed transactions that viewed the partial results of the aborted transaction. Instead, user-supplied compensation functions are executed to compensate for each transaction that was committed at the time of failure or automatic abort.

A compensation function undoes the actions performed by a transaction from a semantic point of view. For example, if a transaction reserves a seat on a flight, its compensation function would cancel the reservation. We cannot say, however, that the database was returned to the state that existed before the transaction started, because, in the meantime, another transaction could have reserved another seat and thus the number of seats that are reserved would not be the same as it was before the transaction.

This approach does not compromise local autonomy, but is hard to realize in practical applications since it requires the use of compensating transactions. In a multidatabase system where different databases exhibit heterogeneous and autonomous behaviors, compensating a transaction whose effects have been propagated to the other databases is an extremely difficult task in the general case.

Chapter 5

A System Model for the Proposed Protocol

5.1 Transaction Processing

A MDBMS can be thought of as containing two logically separate components, a global concurrency controller (GCC) and a global recovery manager (GRMGR). The former is responsible for ensuring serializability and the latter atomicity. When a global transaction is submitted to the system, the GCC schedules the execution of each subtransaction of the global transaction to ensure serializability. As first discussed in Chapter 1, some operations of a local subtransaction may depend on the value of read operations of other local subtransactions. The GCC schedules the execution of each subtransaction properly so that only when the read operations which the write operation depends upon are finished and values forwarded, can the write operation be executed. As mentioned in Chapter 1, such an enforcement of execution order is unavoidable for the applications in which dialogue oriented dependencies exist.

When all of the operations of the global transaction complete successfully, the global transaction is said to have reached its commit point. Once a global transaction reaches its commit point, the GRMGR initiates a commitment process to ensure atomicity. It is assumed that the GCC uses a 2PL at the global level. Thus a subtransaction will not release the locks before the global transaction commits/aborts. (Note that in the proposed commitment protocol which will be presented in the subsequent chapters, individual subtransactions may commit (abort) before the global transaction commits (aborts), the global locks a committed (aborted) subtransaction holds will not be released before the global transaction commits (aborts).)

Due to the isolation assumption described above, it is allowed to discuss a commitment process based only on a single global transaction.

We make the following assumptions:

1. Each global transaction contains at most one subtransaction at any local site.

2. The data items accessed at each local site can be identified by the MDBMS.

3. Each local LDBMS uses a strict two phase locking for concurrency control.

4. A subtransaction will not be aborted at a local site if it has finished all its operations and if there is no failure at the site.

Assumptions 1 and 2 are essentially conditions 3 and 4 in [23]. The first assumption is necessary since otherwise a global transaction may not preserve its

consistency even if it is executed alone. The second assumption is necessary for the MDBMS to determine when synchronization is needed. The third assumption requires that a LDBMS ensure a property called 'strong recoverability' [10] for local executions. Strong recoverability has been considered to be essential to the transaction processing in most of today's database systems. The fourth assumption looks a little strong, but it is quite realistic. Since once a transaction has acquired all the locks it needs, it will never get into a deadlock and is unlikely to be aborted if there is no failure. (A transaction may be aborted for reasons other than deadlock such as operator-initiated abortion, for example. However, the issue of how to handle this kind of abortion is orthogonal to the discussion in this paper.) [18] gives a more detailed justification for an assumption which is duplicated by assumption 4.

5.2 Inter-dependency of Subtransactions

Among the subtransactions of a global transaction, some may write into a local database the values which depend on the values read by some other subtransactions. Specifically, let G be a global transaction, and $G = \{G_1, G_2, \cdots, G_n\}$ where each G_i is a subtransaction at site i. If G_i contains a write operation $w_i(v)$, and the value written is calculated based on the values returned by the read operations $r_{j_1}, \cdots, r_{j_k}$ from $G_{j_1}, \cdots, G_{j_k}$, respectively, then we say that $w_i(v)$ depends on $r_{j_1}, \cdots, r_{j_k}$, and denote this by $r_{j_1} \rightarrow w_i(v), \cdots, r_{j_k} \rightarrow w_i(v)$. We also say that G_i has a dependency on $G_{j_1}, \cdots, G_{j_k}$, and denote this by $G_{j_1} \rightarrow G_i, \cdots, G_{j_k} \rightarrow G_i$.

Definition: A dependency graph for a global transaction G is a directed graph $D(G) = (V, E)$ where V contains all subtransactions of G and an arc from G_i to G_j is in V if and only if G_j has a dependency on G_i.

Thus, a node in a dependency graph denotes a subtransaction. (In the following, when the issues relating to dependency graph is discussed, 'nodes' and 'subtransactions' will be used exchangeably.) If a dependency graph does not contain cycles, then there is a partial order where G_i precedes G_j if and only if there is a path from G_i to G_j in the graph. In this thesis, the following terminologies will be used. In an acyclic dependency graph, G_i is a *parent* of G_j (G_j is a *child* of G_i) if there is an arc from G_i to G_j. G_i is a predecessor of G_j if there is a path from G_i to G_j. G_i is *minimal* if it does not have any incoming arcs, and *maximal* if it does not have any outgoing arcs. All these terminologies can be rephrased in terms of the partial order, and are omitted here.

The value dependency relation among the subtransactions of a global transaction can be pushed even further to *access dependency*. An access dependency implies that the data items accessed by a subtransaction may vary depending upon the values read by another subtransaction. For example, assume G_j is a subtransaction that makes deposits into either the checking or the saving account in a bank for a customer. Whether the deposit will be made into the checking account or the saving account depends on the balance in the third account in a different bank. An access dependency posts some additional problems to a resubmission-based recovery protocol. In the following discussion, it is first assumed that the access

dependency does not occur. In later chapters, this restriction by addressing the problems caused by access dependencies will be relaxed and the solutions will be presented.

Chapter 6

The Protocol

6.1 The Commit Sequence Based on Value Dependency Graph

As mentioned before, the atomic commitment of a global transaction in a multidatabase is difficult to achieve since local sites can unilaterally commit or abort a subtransaction. Life would be much easier if whenever a subtransaction aborts, it could be resubmitted at some later time without destroying the semantics of the global transaction. Unfortunately this in general is not possible. The problem is as follows. Suppose in a global transaction G, a subtransaction G_i aborts while another subtransaction G_j which depends on G_i commits. If we resubmit G_i, then some of the read operations in G_i may read different values from what they read previously. Thus G_j would have written a different value than the one committed should it also be resubmitted alongside with G_i. This has the effect that G_j has written a value that is calculated based on a value G_i does not read, a violation of the semantics of G.

The above problem can be resolved by requiring that G_i commits first and

G_j starts committing only if G_i has committed. In this way, when G_i aborts, G_j can also be aborted and then both G_i and G_j are resubmitted at the same time. Clearly, this solution works only if there is no cyclic dependency between G_i and G_j. In general, if the dependency graph of a global transaction does not contain a cycle, then the partial order defined in Section 5.2 can be used to define the commit order of the subtransactions. In this case, a subtransaction can commit if and only if all its parents have committed. If a subtransaction aborts, its program can be resubmitted later without compromising the correctness since none of its descendants has committed, and hence they can all be resubmitted at the same time.

6.2 Dealing with a Cyclic Dependency Graph

If the dependency graph of global transaction G contains cycles, no partial order exists in the dependency graph. Thus the commit sequence described in the last section cannot be used. To resolve this problem, we break the cycles by choosing some nodes in the dependency graph and deleting all arcs emanating from them. Since the resulting graph no longer contains cycles, the commit order described above can be used.

Definition: Given a dependency graph M, a set K of nodes is a *key set* of M if by deleting the outgoing edges from the nodes in K all cycles can be broken. The resulting acyclic graph M' is called a *relaxed* dependency graph generated by K. The nodes in K are called *firm* nodes for M'.

Note that for any dependency graph, there always exists at least one key set, i.e., the set of all nodes in the dependency graph. It is also easy to see that any superset of a key set is still a key set in a dependency graph. As will be seen in later chapters, the nodes in a key set involve higher cost than the other nodes in the commitment protocol. Thus we are always in favor of small key sets.

Definition: For any dependency graph, a minimal key set is a key set such that none of its proper subsets is a key set.

Example 1: Given in Figure 7.3 c (page 68) is a dependency graph which contains five cycles. There are three minimal key sets, $\{G_2, G_3\}$,$\{G_1, G_4\}$ and $\{G_1, G_2\}$. The superset of any of them is also a key set. The relaxed dependency graph generated by the first two minimal keys are shown in Figure 7.4 a and b.

In the following discussion, we will be interested only in a minimal key set. For easy presentation, 'key set' will be used simply for 'minimal key set'.

Note that in general different key sets generate different relaxed dependency graphs. Also note that a firm node (i.e., subtransaction) is well defined only if it is related to a specific key set, since a firm node for one key set may not be firm for the other. Thus in the following discussion, whenever we mention a firm subtransaction without at the same time mentioning the related key set, it should be understood that a key set has been (implicitly) related.

Since a relaxed dependency graph is generated by deleting all outgoing edges of the firm nodes in the initial dependency graph, a firm node in the corresponding

relaxed dependency graph does not have any child. The children of a firm node in the initial dependency graph are called the *hidden children* of that firm node (or the firm node is the hidden parent of those children) in the relaxed dependency graph. Note any hidden child of a firm node still depends on that node. Thus even if the relaxed dependency graph has been (artificially) made acyclic, we still must cope with the problems arising from those dependencies when we attempt to use the partial order in the relaxed dependency graph as the commit order for subtransactions.

The point is that a hidden child G_j can never be a descendent of its hidden parent G_i in the relaxed dependency graph. Thus G_j may commit before G_i. If site i fails after G_j commits but before G_i does, then G_i will be aborted locally by site i when the failure is repaired. Since G_j has a dependency on G_i and G_j has committed, we must not let G_i read different values than it did before. To achieve this goal, two approaches are possible. One is to force G_i to read the old values each time it is restarted. The other is to restore the values G_i writes before the failure by initiating a *restore transaction*. However, both methods are subject to interference by local transactions, as demonstrated by the following example.

Example 2: Assume $G_1 = r_1(a)w_1(b)$, $G_2 = w_2(c)$ and a dependency $r_1(a) \rightarrow w_2(c)$. Suppose G_1 has been chosen as a firm node. Thus G_2 is the hidden child of G_1 in the relaxed dependency graph. After G_2 commits, a site failure aborts G_1. After the site is repaired, we attempt to restore the value written by $w_1(b)$. But before a restore transaction $R_{G_1} = w_r(b)$ is submitted, a local transaction

$L = w_\ell(a)w_\ell(b)$ is submitted and committed. Then R_{G_1} commits. We have the following execution: $r_1(a)w_\ell(a)w_\ell(b)w_r(b)$. This is equivalent to unserializable execution $r_1(a)w_\ell(a)w_\ell(b)w_1(b)$.

Example 2 shows how restoration of a subtransaction may introduce inconsistency in the presence of local transactions. A similar scenario can occur for the first approach, i.e., forcing a subtransaction to read the old values it read previously. To make either approach work, proper provisions must be made to handle the local transactions after the failure.

6.3 Handling Local Transactions in the Recovery Period

To avoid the problems like that described in Example 1, we may use a method similar to the one suggested in [19], which excludes indiscriminately all local transactions before the restoration completes. Although it is simple, this method may sacrifice performance unnecessarily since only a portion of the local transactions may actually interfere with the recovery of subtransactions. Alternatively, we choose to prevent only those local transactions which try to modify the data items read by the subtransaction being recovered.

Let F be a firm subtransaction at site i which is undergoing a recovery after the site failure. If a local transaction L is submitted before the end of the recovery period, it is under the scrutiny of a process called L-HANDLER which determines if L can be accepted.

Process L-HANDLER

{In the following R_F denotes the recovery transaction for firm subtransaction F. M denotes the set of all firm subtransactions whose recovery transactions are yet to commit at this site.}

Get a local transaction L;

accept← true;

for each $F \in M$ do

if $r(F) \cap w(L) \neq \phi$ then

accept← false

end if

end for

if *accept*=false then reject L else accept L

Since a local transaction is accepted only if its write set does not intersect the read set of any firm subtransaction under the recovery, the problem in Example 2 will not occur.

Note that logically, process L-HANDLER is not considered as part of the GRMGR. However, it will interact with the GRMGR. For example, the values in M will be updated whenever the recovery for a firm subtransaction commits.

6.4 Distributed Commitment

In this section, the overall structure of the commitment protocol are described. Assume a global transaction G is submitted to site i. (As usual, the term coordi-

nator is used to represent the site where the global transaction is submitted and participants the sites where subtransactions are executed.) G will be analyzed first and its dependency graph determined. If the dependency graph contains a cycle, a key set of nodes will be chosen for the purpose of breaking the cycles. The coordinator then deletes all arcs emanating from them in the dependency graph to generate a relaxed dependency graph. When the coordinator submits the subtransactions to the local sites, for each subtransaction in the key set, it includes into the messages a flag indicating the subtransaction is firm, together with the read set of that firm subtransaction. The latter information will be necessary if the firm subtransaction later must undergo a recovery due to site failures. If the global transaction can reach its commit point, then the coordinator will initiate the commitment protocol, otherwise it aborts the entire global transaction. Shown below is a description of the actions taken by the coordinator in the commitment protocol.

Coordinator

1. Create dependency graph M of G;

2. If M contains cycles, choose a key set K of nodes, and delete all arcs emanating from the firm nodes, i.e., the nodes in K. Let M' be a variable which is initialized to be the relaxed dependency graph generated by K. For each $G_j \in K$, send START-G_j-FIRM, as well as $r(G_j)$ to site j. For each $G_j \notin K$, send START-G_j to site j.

{G is executed here. During the execution, store the values read by each subtransaction into the log at the coordinator site. If any subtransaction aborts before the global transaction reaches its commit point, then abort G_j for all j, otherwise execute the following steps for commitment. }

3. $P_0 \leftarrow \phi$;

4. $P \leftarrow \{Z : Z$ is unmarked and is a minimal node in M' and $Z \notin P_0\}$;

5. If $P \neq \phi$, then for each $G_j \in P$, $P_0 \leftarrow P_0 \cup \{G_j\}$ and send COMMIT-G_j to site j;

6. If $M' = \phi$, stop; (The commitment protocol finishes.) otherwise wait until one of the following events occur:

   - G_j-COMMIT is received from site j:

     a . Delete G_j and the adjacent arcs from M';

     b . $P_0 \leftarrow P_0 - \{G_j\}$;

     c . Go to step 4;

   - G_j-ABORT is received from site j:

     a . Activate the restart process to restart G_j and all its descendants in M' and mark them in M';

     b . $P_0 \leftarrow P_0 - \{G_j\}$;

     c . Go to step 6;

   - G_j restart finished: unmark G_j in M', go to step 4;

The message START-G_j-FIRM indicates the start of a firm subtransaction G_j. The reason that the values read or written by each subtransaction are required to be stored in the log is that those values may be used later by the execution of some subtransactions when they have to be restarted. (See Section 6.6.) Variable P_0 denotes the set of all subtransactions which have not been aborted since the last time the coordinator sent the commit signal. P is the set of the subtransactions whose commit operations can be initiated as a result of the commitment of some of its parents. A subtransaction in M' being marked signifies it is currently being restarted and therefore should not get started for commitment.

For the sake of clarity, the restart process is logically separated from the commitment protocol. The coordinator can activate the restart process for a subtransaction whenever it wishes and be informed of the final status. The completion of the restart of a subtransaction is signified by the completion of the last operation of that subtransaction. When the restart of a subtransaction is finished the coordinator is informed, which then sends a commit signal to the corresponding site.

The following is the actions taken by a participant at site j.

Participant j

1. When START-G_j-FIRM, together with $r(G_j)$ are received, store G_j-START and G_j-FIRM, as well as $r(G_j)$ in the log; if START-G_j is received, store G_j-START into the log;

{Here the operations of G_j are delivered to the LDBMS. For each write operation of G_j where G_j-FIRM is in the log, store the value to be written into the log.}

2. When COMMIT-G_j is received, perform the following operations in the order specified:

   a . Store G_j-COMMIT into the log;

   b . Request the LDBMS to commit G_j. One of the following events must occur

   - The LDBMS commits the G_j: Send G_j-COMMIT to the coordinator;
   - Site failure occurs: After the failure is repaired, perform the recovery operations. (See section 6.5);

3. When ABORT-G_j is received, store G_j-ABORT into the log, inform the LDBMS to abort G_j;

The messages a participant stores in the log indicate the status and the types of a subtransaction. When the participant receives COMMIT-G_j, it must confirm the commit of G_j to the coordinator. The confirmation may make the coordinator to commit more subtransactions which follow G_j in the partial order.

Recall that a participant can receive an abort message only if one or more subtransactions abort before the global transaction reaches its commit point. In this

case, the coordinator must have aborted the entire global transaction. Thus whenever site j receives ABORT-G_j, it must abort G_j locally. Note that the actions described here are those taken in the normal case. When failures occur, special failure recovery actions must be taken at the global level, which are discussed in the subsequent sections.

6.5 The Global Failure Recovery

When a site failure occurs, the execution of an ongoing subtransaction may be intercepted by the failure. The fate of the transaction after the site is repaired depends on its status and the status of the global transaction to which it belongs at the time of the failure. The global transaction may or may not have reached its commit point before the failure occurs. In the latter case, the subtransaction will be aborted. In the former case, the subtransaction will undergo different recovery procedures, depending upon its status and types. These have been shown in the process named G-RECOVERY.

G-RECOVERY

Enable L-HANDLER;

For each G_j such that G_j-START is in the log:

Case 1. G_j-FIRM is not in the log:

1. if G_j-COMMIT is not in the log, then store G_j-ABORT in the log, send G_j-ABORT to the coordinator and **exit**;

2. if G_j-COMMIT is in the log, then inquire the LDBMS about the status of G_j. If G_j aborts, replace G_j-COMMIT by G_j-ABORT, send G_j-ABORT to the coordinator and **exit**, otherwise send G_j-COMMIT to the coordinator and **exit**;

Case 2. G_j-FIRM is in the log:

1. if G_j-COMMIT is not in the log, then contact the coordinator. If the global transaction has been aborted, then store G_j-ABORT into the log and **exit**, otherwise store G_j-ABORT into the log, send G_j-ABORT to the coordinator and record G_j as recovering.

2. if G_j-COMMIT is in the log, then inquire the LDBMS about the status of G_j. If G_j aborts, record G_j as recovering and repeatedly submit R_{G_j} to the LDBMS until it commits,then record G_j as recovered, send G_j-COMMIT to the coordinator, and **exit**, otherwise send G_j-COMMIT to the coordinator and **exit**.

If no transaction is recovering, disable L-HANDLER.

In the algorithm, 'exit' is used to denote the end of the recovery for G_j. After that point, G_j will switch to the normal mode. The two cases correspond to two different types of subtransactions, non-firm and firm. Note that G_j-COMMIT being in the log does not necessarily mean G_j has been committed locally, since site j may fail after it receives COMMIT-G_j from the coordinator and inserts G_j-COMMIT into the log, but before the LDBMS actually commits G_j. However, the presence of G_j-COMMIT in the log does indicate all G_j's parents have committed. Thus R_{G_j}, the restore transaction for G_j, can be initiated if G_j has been found aborted. On the other hand, if G_j-COMMIT is not in the log, then G_j has been aborted for sure by the LDBMS during the local recovery. This is implied by the order at step 2 in the commitment protocol for a participant. Thus the GRMGR stores G_j-ABORT into the log, which indicates G_j has been aborted locally. Note that when G_j is aborted locally but the global transaction is not aborted globally, G_j will switch to the normal mode if it is non-firm, and stay in failure mode if it is firm. The reason they are treated differently in this respect is because the restart process of a non-firm subtransaction is viewed as a normal case action. In other words, for a non-firm subtransaction G_j, after the GRMGR sends G_j-ABORT, G_j will return to the normal mode. Thus the restart process for G_j will run in normal mode. On the other hand, the restart process of a firm subtransaction is viewed as a failure case action. This treatment is necessitated by the fact that the restart of a firm subtransaction always triggers the execution of L-HANDLER, which is

regarded as an exception handler. Thus it should run only for a subtransaction which is in the failure mode.

6.6 Restart a Subtransaction

Under two circumstances a subtransaction must be restarted: 1. it aborts but one of its predecessors commits; 2. one of its parents restarts. The first condition implies that a subtransaction is restarted only if the global transaction has passed its commit point. Note that from assumption 4 in Section 5.1, this implies that the subtransaction is aborted due to failure. The second condition implies that once a subtransaction restarts, all its descendents must restart as well. In this case we say that the restarts of the descendants are *caused* by that subtransaction, and that they are under the *same restart process.* If the restart of a subtransaction is not caused by any of its predecessors, we say that it is the *originator* of that restart. Clearly, if a subtransaction is the originator of a restart, then the restart must result from its being aborted (due to failures).

Note that the commitment process will restart a subtransaction only if it is certain that the subtransaction falls into one of the above two cases. For the first case, this means that the coordinator has received an abort message for the subtransaction and a commit message for its predecessor. For the second case it means that the coordinator has initiated the restart process for one of its predecessors.

When the restart process is activated for a subtransaction G_i at site i, the

specific actions it takes depend upon whether or not G_i is firm. If G_i is not firm, then the restart process will resubmit the program for G_i as if G_i were a new subtransaction. This has two implications. One is that all information about G_i previously stored in the log becomes void. (These include, among other things, the values previously read by G_i.) The other is that the LDBMS will be informed to abort G_i, and therefore release all the locks obtained by G_i.

If G_i is firm, the restart process only *back track* the program for G_i. This means the following. Firstly, the coordinator views all the values which G_i previously read as still valid. This is necessary since those values will be used by G_i itself for calculation. (See the explanation below.) Secondly, if G_i is in normal mode when it is restarted, then it will not be aborted, and therefore still hold whatever locks it had obtained before. This is essential for the correctness since in the normal mode, L-HANDLER is not active, and therefore releasing the locks could possibly let the local transactions introduce inconsistency. Finally, if G_i is restarted after the failure of site i, then when it should execute a read operation, the coordinator picks up the value for that read operation from the log and delivers it to site i, rather than letting G_i reread it from the local database. (Note that this is possible since our protocol requires the coordinator log the value read by each read operation of the global transaction.) This is necessary since after site i is repaired, all the read locks initially obtained by G_i are released. If G_i rereads the corresponding data items, it may risk reading different values than it did before. On the other hand, recall our assumption that G_i has no access dependency on

any of its parents. Thus if the restart of G_i is caused by its parents, then had it actually performed reread it would not read different data items from those stored in the log[1]. Thus obtaining the values from the log for a firm subtransaction is justified.

One more detail is worth mentioning here. When subtransaction G_i, whether firm or not, is restarted, either it is the originator of this restart, or the restart is caused by some of its parents. In the former case, when G_i requires the values from its parents, those values cannot be read by the parents at run time since the parents have already finished their operations (or may have even committed). However, this posts no difficulties since all the values read by any subtransactions are logged by the coordinator. Thus the coordinator can get those values from the global log, without affecting the execution of G_i. In the latter case, on the other hand, the values will be read at run time by those parents whose restarts cause G_i to be restarted. In summary, our algorithm for restarting a subtransaction is the following.

Process Restart

{Assume G_i is the subtransaction under the restart.}

Case 1 : If G_i is not firm, then mark all information in the log about G_i as void, inform the LDBMS to abort G_i and resubmit G_i to the LDBS. When G_i needs values from a parent, the values will be read at run time if G_i and

[1]Actually this method is not necessary if G_i is in the normal mode when being restarted. Since it still holds the read locks, G_i can read the same values anyway as it did before. For simplicity this case will not be treated differently.

that parent are under the same restart, otherwise the coordinator fetches them from the log;

Case 2 : If G_i is firm, then reevaluate the write operations only, but get the value for each read operation from the log at the coordinator site. When G_i needs values from a parent, do the same thing as that in case 1. When G_i finishes, record G_i as recovered.

6.7 An Example

In this section, an example is used to show how the proposed protocol works in a typical scenario.

Example 3: Let G be defined as $G \equiv r(e)r(a)r(c)w(f)w(b)w(d)$ where a and b are stored at site 1, c and d at site 2, e and f at site 3. Let $G_1 \equiv r_1(a)w_1(b)$, $G_2 \equiv r_2(c)w_2(d)$, $G_3 \equiv r_3(e)w_3(f)$.

Suppose the dependencies among the operations are as follows: $r_3(e) \rightarrow w_1(b)$,$r_1(a) \rightarrow w_2(d)$, $r_2(c) \rightarrow w_3(f)$. The dependency graph for G is shown in Figure 6.1 a.

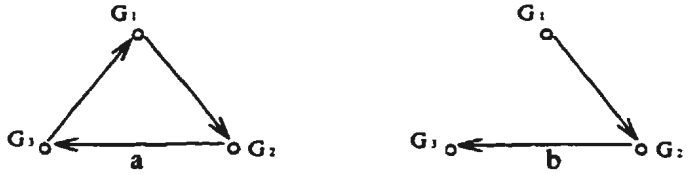


Figure 6.1. dependency graph and relaxed dependency graph of G

Suppose $\{G_3\}$ has been chosen as the key set, thus G_3 is a firm subtransaction.

The relaxed dependency graph is shown in Figure 6.1 b. Assume site 1 is the coordinator and initiates the execution. The log at each site when G reaches its commit point is shown in Figure 6.2. ($(r_i(p), q)$ means $r_i(p)$ reads value q from data item p and $(w_i(p), q)$ means $w_i(p)$ writes value q into data item p.) Note that since G_3 is a firm subtransaction, its read set $\{e\}$ and the value u written by its write operation have also been logged. Since site 1 is the coordinator, it also stores into the log the values read by each subtransaction.

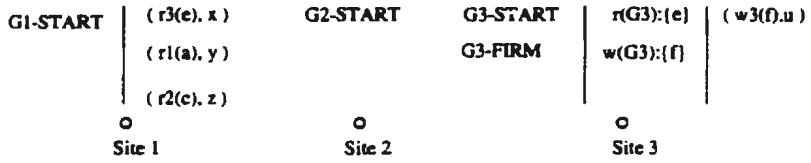


Figure 6.2. The logs when G reaches commit point

Now, site 1 starts commitment process. It starts committing G_1 first since G_1 is a minimal node in the relaxed dependency graph. Suppose site 1 is notified of the commit of G_1, it modifies the dependency graph as shown in Figure 6.3. It then sends COMMIT-G_2 to site 2.

Figure 6.3. Dependency graph when G1 commits

Suppose site 2 fails before COMMIT-G_2 arrives. When site 2 is repaired,G_2-START is the only message in the log. Thus site 2 inserts G_2-ABORT into the log, resulting in Figure 6.4.

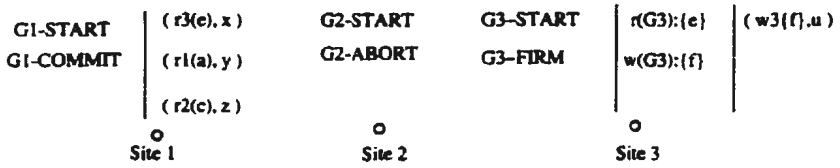


Figure 6.4. The logs after site 2 is repaired

Site 2 then sends G_2-ABORT to site 1, the coordinator. Upon receiving the message, site 1 initiates the restart process for G_2 and G_3. G_2 will be resubmitted as a new transaction. During its execution, when G_2 needs the values from $r_1(a)$, the coordinator will fetch y from the log. However, when G_3 needs value from $r_2(c)$, the value is read from c by $r_2(c)$ at run time since both transactions are under the same restart. Suppose the new value returned is z'. Based on z' $w_3(f)$ writes a value, say u' into f. When $r_3(e)$ is about to be executed, the coordinator fetches x from the log. Note that x is exactly the value based on which the committed value written by $w_1(b)$ has been calculated.

When the execution of G_2 and G_3 reaches the commit point, the coordinator will start another cycle of commitment as it did before, but only for G_2 and G_3.

Now, suppose site 2 notifies site 1 that G_2 commits successfully, site 1 then starts committing G_3 by sending COMMIT-G_3 to site 3. But before the message arrives, site 3 fails.

When site 3 is repaired, G_3-START is the only message in the log. Thus the GRMGR at site 3 inserts G_3-ABORT into the log, and sends G_3-ABORT to the coordinator. The log at each site at this moment is shown in Figure 6.5.

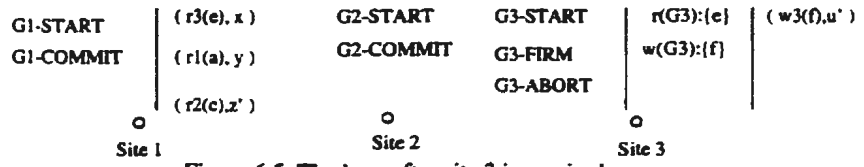


Figure 6.5. The logs after site 3 is repaired

Now the coordinator initiates the restart process for G_3, which requests the value for $r_2(c)$ to carry out $w_3(f)$. Note that G_3 now is the originator of this restart. Thus the coordinator gets z' from the log, making $w_3(f)$ write into f the same value as it did last time, namely, u'. When G_3 should execute $r_3(e)$, the coordinator fetches x from the log. This again guarantees the validity of the value committed by $w_1(b)$. (Note that now L-HANDLER is in action to prevent any local transaction from modifying e.) Suppose G_3 finishes the restart. The coordinator then sends COMMIT-G_3 to site 3. Site 3 receives it and stores G_3-COMMIT into the log. The log at this point is shown in Figure 6.6.

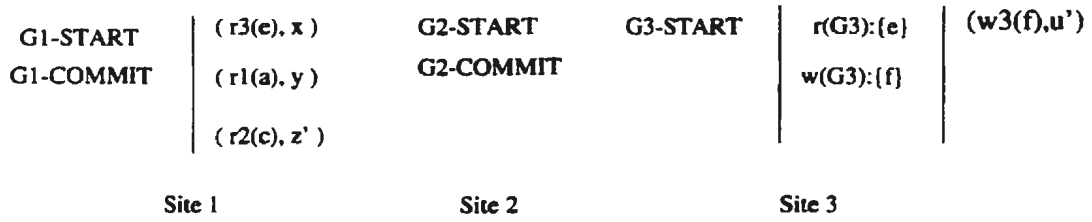


Figure 6.6. The logs after the second failure of site 3

Now suppose before the GRMGR informs the LDBMS to commit G_3 locally, site 3 fails again. When site 3 is repaired, G_3-COMMIT is in the log. Thus $w_r(f)$,

the restore transaction for G_3, is initiated to restore value u'. When R_{G_3} commits, site 3 sends G_3-COMMIT to the coordinator. This completes the commitment for G. □

Chapter 7

An Informal Discussion about Access Dependency

7.1 The Problems Caused by Access Dependency

As mentioned before, access dependency means that the data items, not just the values of a data item, that a subtransaction accesses depend on the values read by some other subtransactions. In this chapter, this problem will be introduced and the idea behind the approaches to solving it will be discussed in a semi-formal manner. The reason is twofold. First, the theme of the thesis is centered on value dependency. Access dependency is inherently a different problem in nature, and therefore a discussion on this topic is largely orthogonal to the main theme of the thesis. Second, since its formation is very similar to value dependency, it is interesting to get some insight into the problem and its solution.

If the set of data items which a subtransaction G_j reads from (write into) may vary depending upon the values read by another subtransaction G_i, we say that G_j has an *r-dependency* (*w-dependency*) on G_i. In the following we will use the

notation $G_i \xrightarrow{ac} G_j$ to denote that G_j has an access dependency on G_i.

If a firm subtransaction has an access dependency on its parents, then the correctness of the protocol cannot be guaranteed, as demonstrated in the following example.

Example 4: A global transaction G is defined as follows.

if b=100 then

    if c=0 then e:=5 else e:=10;

else

    if d=0 then e:=15 else e:=20;

if f=0 then a:=0 else a:=1

Suppose data item a and b are stored at site 1, c and d site 2, and e and f site 3. The subtransactions are defined as follows. $G_1 = r_1(b)w_1(a)$, $G_2 = r_2(c)$ or $r_2(d)$ depending upon the values read by G_1, and $G_3 = w_3(e)r_3(f)$. Thus we have $G_1 \xrightarrow{ac} G_2$. In addition, G_2 r-depends on G_1. The dependency graph for G is shown in Figure 7.1 a. (The darkened line denotes an access dependency.) Now we choose $\{G_2\}$ to be the key set. Thus G_2 is the firm subtransaction and G_3 is its hidden child. The relaxed dependency graph is shown in Figure 7.1 b.

a.Dependency graph b. Relaxed dependency graph generated by key set {G2}

Figure 7.1. The graphs for Example 4

The commit order is G_3, G_1, G_2. Suppose initially, $b = 100$ at site 1, $c = 0$ and $d = 0$ at site 2, and $f = 0$ at site 3. Thus, G_1 reads 100 from b, G_2 reads 0 from c and G_3 writes 5 into e. Now, suppose G_3 commits but G_1 aborts due to site failure. Thus G_1 and G_2 will be restarted. Suppose when G_1 is resubmitted, the value in b has been modified by a local transaction to 101. This would require G_2 to execute $r_2(d)$, not $r_2(c)$. In other words, operation $r_2(c)$ which was performed last time becomes *illegal* should the program for G_2 be executed in the current restart. Thus G_3 should write into e either 15 or 20, instead of 5. Since value 5 in e has already committed, we have had an incorrect execution.

The above example shows how an r-dependency may jeopardize the correctness of the proposed protocol. A w-dependency can also cause the same problem. To see how, suppose G_j is a firm subtransaction which has a w-dependency on G_i. Further assume that its hidden child G_k has committed. Now assume G_i fails but G_j does not, then both G_i and G_j will be restarted. Note that in the execution for the resubmission, G_j still holds all the locks it initially obtained. Since G_j w-depends on G_i, it may perform different write operations in the restart and therefore request new write locks. Thus local deadlock may occur and, to break

the deadlock, G_j may be aborted by the LDBMS. When this happens, the LDBMS will release all the locks held by G_j and reschedule it at some later time. Thus when it is rescheduled, G_j may read a value that has been modified by some local transactions and make the values committed by G_k invalid.

7.2 Approaches

It is easy to see that the problems arising from either r-dependency or w-dependency are from the same source. That is, due to an access dependency, a value read or a read operation itself in the previous execution of a firm subtransaction becomes illegal should the program of the firm subtransaction be executed in the current restart. In the following, we partially formalize this idea. (A full formalization would require a formal model for the transaction execution using resubmission, and is beyond the scope of this thesis.)

Definition: Let G_j be a firm subtransaction which is currently under a restart. Let $p(a)$ be an operation, where $p \equiv r$ or w, which was performed in the last submission of G_j. Then we say that $p(a)$ is legal in the current restart if the program for G_j had been executed in the restart, the execution would contain $p(a)$.

Definition: Let G_j and $p(a)$ be the same as in the above definition. Let x be the value which $p(a)$ read from or wrote into a in the last submission of G_j. We say x is legal for $p(a)$ in the current restart of G_j if 1. $p(a)$ is legal in the current restart; 2. $p(a)$ reads x from or writes x into a should the program for G_j be

executed in the current restart.

From the above definition, the read operation $r_2(c)$ is not legal when G_2 is restarted. Thus the value 0 for $r_2(c)$ is also not legal.

If the values previously read by G_j become illegal, then the values calculated based on those illegal values which G_k commits also become illegal. In other words, those values committed by G_k have to be *invalidated*. For instance, in Example 4, since $r_2(c)$ becomes illegal when G_2 is restarted, the committed value 5 in e by G_3 is invalidated.

Note that in a multidatabase, only those values which are committed by a subtransaction that is a hidden child of some firm transaction can possibly be invalidated. This is because if a subtransaction is not a hidden child of any firm subtransaction, then it can never happen that the subtransaction commits before its parents. In other words, once it commits, all of its parents must have committed, and therefore will never be restarted. Thus all we need is to find a way of avoiding invalidating the values committed by a subtransaction that is a hidden child of some firm subtransaction.

From the previous discussion, we observe that for a firm subtransaction G_j and its hidden child G_k, the restart of G_j will not invalidate the committed values of G_k if once G_k commits, the read operations and their values stored in the log for G_j are always legal when G_j is restarted. Interestingly, this goal can be achieved by only slightly modifying a dependency graph. The method is based on a theorem presented below. Before the theorem is given, a lemma which is

essential for the theorem will be present.

Lemma 1: Assume a dependency graph M contains two paths: $G_i \rightarrow G_j \rightarrow G_k$ and $G_i \rightarrow G_k$. Let M' be the relaxed dependency graph of M generated by key set K. Then at least one of the following conditions is true:

1. G_j is not a firm subtransaction;

2. both G_i and G_j are firm subtransactions;

3. G_i is a parent of G_k in M'.

proof: Suffices it to show that if the first and the second conditions are not true, then the third condition must be true.

The fact that the first two conditions are not true implies that G_i is not a firm subtransaction. From the way a relaxed dependency graph is obtained from the dependency graph, all of the outgoing arcs of G_i contained by M are presented in M'. Thus M' contains arc $G_i \rightarrow G_k$. This means that G_i is a parent of G_k in M'. □

The commitment protocol is slightly modified as follows. In the protocol presented in section 6.4, when a firm subtransaction is restarted, the values to be read by its read operations are always obtained by the coordinator from the log, regardless of whether or not any of its hidden children has committed. Now It is required that this action be taken only if at least one of the hidden children of the firm subtransaction has committed, otherwise its restart procedure must be the same as that for a non-firm subtransaction, namely, being aborted locally

and resubmitted as if it were a new transaction. This is necessary since if a sub-transaction has an access dependency on its parents, then when its parents are restarted, those read operations in the log may become illegal, as it is explained previously.

The following theorem establishes a basis for the method.

Theorem 1: Let M be a dependency graph for a global transaction G in which G_j has an access dependency on G_i, and M' be a relaxed dependency graph. If M contains two paths: $G_i \rightarrow G_j \rightarrow G_k$ and $G_i \rightarrow G_k$ and G_j is firm, then once G_k commits, the read operations and their values stored in the log for G_j are always legal if either the restart of G_j is caused only by G_i or G_j is the originator of the restart.

proof: Note that if G_j is the originator of a restart, then this restart will not make the the read operations or their values stored in the log illegal. This is because the restart of G_j is not caused by its parents. Thus whatever values it gets from its parents are fetched from the log, and hence never change. Therefore, a restart of G_j caused by G_i will only be considered.

By Lemma 1, one of the three conditions specified there must be true. Since G_j is assumed to be firm, the last two conditions will only be considered. Suppose condition 2 is true. Since G_i is firm, G_k must be its hidden child in M'. Since G_k has committed, during the restart of G_i, its read operations get the values from the log. This implies that they get identical values to what they got last time. Thus the corresponding restart of G_j performs the same read and write operations

as it did before. This means its read operations are legal. In addition, since it performs the same write operations, the situation described in the last paragraph of Section 7.1 will not occur. This means the values stored in the log for those read operations of G_i are also legal. Now suppose condition 3 is true. Since G_k is a child of G_i in M', it commits after G_i. Thus if a restart of G_j occurs after G_k commits, it occurs also after G_i commits. Thus G_j must be the originator of that restart which, as it is explained earlier, cannot make either its read operations or their values illegal. □

From Theorem 1, in the face of access dependency, additional arcs can be added to a dependency graph to prevent any committed values from being invalidated. The following algorithm is used for this purpose.

Algorithm MODIFY

Input: a dependency graph M and an access dependency relation among the nodes in M;
output: a modified dependency graph used by the commitment protocol to ensure atomic commitment;

while there is a path $G_i \stackrel{ac}{\rightarrow} G_j \rightarrow G_k \in M$ and G_j is in a cycle of M
and $G_i \rightarrow G_k \notin M$ do
    add $G_i \rightarrow G_k$ to M
end while

The reason that we check if G_j is in a cycle is because access dependency $G_i \stackrel{ac}{\rightarrow}$

G_j may invalidate G_k only if G_j can possibly be chosen as a firm subtransaction. This can happen only if G_j is involved in a cycle. The following theorem establishes the correctness of the method for handling the access dependency. A formal proof of the theorem will not be given, but the idea will be illustrated.

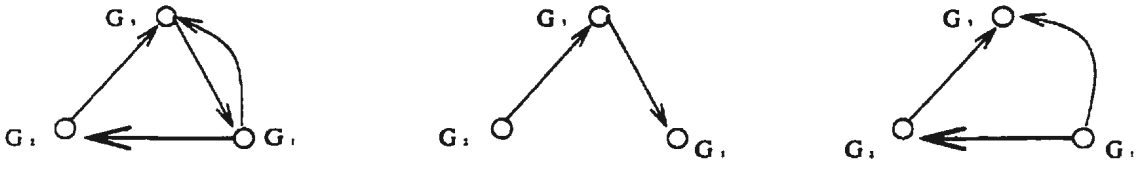
Theorem 2: Let G be a global transaction, N be the dependency graph for G, M be the modified dependency graph generated by applying algorithm MODIFY to N, K be a key set in M which generates a relaxed dependency graph M', and $G_1 \in K$ be a firm node of which Q is a hidden child. Then once Q commits, the values it writes will never be invalidated by any restart.

proof. We consider any path in M with the form $p = G_n \stackrel{ac}{\rightarrow} G_{n-1}, \cdots, \stackrel{ac}{\rightarrow} G_1$ where for all $i, 1 \leq i \leq n$, G_i is a firm node and G_n has no access dependency on any of its parents. We must prove that for any such path, the values committed by the hidden children of G_1 will not be invalidated by any restart. To this end, we only need to prove that for all G_i, the read operations and their values stored in the log are legal once Q commits. (Note that it is not sufficient to prove such a legality only for G_1, since if G_m where $2 \leq m \leq n$ has illegal read operations, then all the values committed by Q may still be invalid since they depend on the read operations and their values for G_1, which depend transitively on the read operations and the values for G_m.) Firstly, assume $n = 1$. For any parent P of G_1, If $P \rightarrow G_n$ is not an access dependency, then once Q commits, the read operations stored in the log for G_1 are always legal. If $P \rightarrow G_n$ is an access dependency, from the definition of algorithm MODIFY, there must be an arc $P \rightarrow Q$ in M'.

By Theorem 1, once Q commits, the read operations and their values stored in the log for G_1 are legal. An easy induction can prove the theorem in the general case. □

Here example is used to show how the protocol works in a typical scenario.

Example 5: We apply MODIFY to the dependency graph in Figure 7.1 a. (The darkened line denotes an access dependency.) The modified dependency graph is shown in Figure 7.2 a. It is easy to see that there are only two possible key sets for this modified graph, $\{G_1\}$ and $\{G_3\}$. The former generates the relaxed dependency graphs in Figure 7.2 b and the latter generates that in Figure 7.2 c.



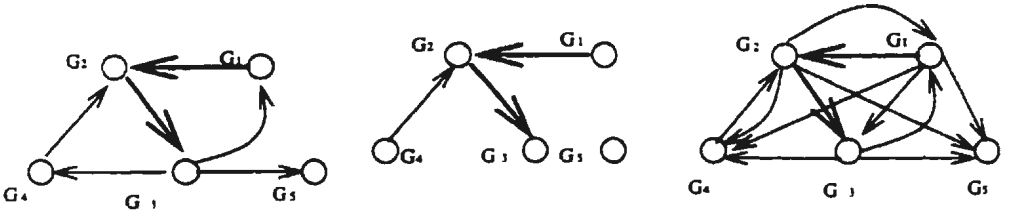
a. Modified dependency graph b. Relaxed dep.graph generated by {G1} c. Relaxed dep. graph generated by {G3}

Figure 7.2. DG and relaxed DGs after applying MODIFY to the DG in Example 4

When $\{G_1\}$ is the key set, G_1 is the firm node whose hidden children are G_2 and G_3, and parent is G_3. The read operations and their values for G_1 stored in the log are always legal since the only dependency, $G_3 \rightarrow G_1$, G_1 has on its parent is not an access dependency. A similar situation occurs when $\{G_3\}$ is the key set. In this case, the firm subtransaction G_3 has two dependencies, $G_1 \rightarrow G_3$ and $G_2 \rightarrow G_3$, neither being an access dependency.

To give more intuitions for the method, a slightly more complicated example in the following will be given as a conclusion of the section.

Example 6: Suppose a global transaction G contains five subtransactions, and there are two access dependencies, $G_1 \stackrel{ac}{\rightarrow} G_2$ and $G_2 \stackrel{ac}{\rightarrow} G_3$. The dependency graph for G is shown in Figure 7.3 a. (A darkened line in the graph represents an access dependency.) This dependency graph contains two cycles. Obviously, $\{G_3\}$ is a key set, which generates the relaxed dependency graph in Figure 7.3 b. G_3 has three hidden children, G_4, G_5 and G_1 in the relaxed dependency graph. They can commit before the rest of the nodes. Suppose G_4 and G_5 commit but G_1 aborts due to failure. Since $G_1 \rightarrow G_2$ is an access dependency, when G_1 is restarted, the read operations stored in the log for G_2 may become illegal, which in turn may cause the read operations and their values in the log for G_3 to be illegal. Thus the committed the values by G_4 and G_5 may be invalidated.



a. Initial DP. b. Relaxed DP generated by {G3} c. Modified DP by applying MODIFY to the initial DP

Figure 7.3. DP and modified DP for Example 6.

Now we apply MODIFY to Figure 7.3 a. The resulting dependency graph is shown in Figure 7.3 c. Note that in this new dependency graph, $\{G_3\}$ is no longer a key set. There are three choices for the key sets. These are $\{G_2, G_3\}$, $\{G_1, G_4\}$ and $\{G_1, G_2\}$. The construction of the relaxed dependency graphs for these key sets are straightforward. Shown in Figure 7.4 a and b are those gener-

ated by the first two key sets. (The number inside a circle indicates the commit order for the corresponding subtransaction. A boldfaced name represents a firm subtransaction).

a. Relaxed DP generated by {G2,G3}

b. Relaxed DP generated by {G1,G4}

Figure 7.4. The relaxed DPs

To see why the committed values by any hidden children of a firm node will not be invalidated by any restart, let us consider the case where $\{G_2, G_3\}$ is the key set. In the dependency graph in Figure 7.3 c, there are two paths, $G_2 \stackrel{ag}{\rightarrow} G_3$ and G_2, which have the form of the path in the proof for Theorem 2, with $n = 2$ and $n = 1$, respectively. We now consider the first path. Take the hidden child G_4 of G_3 as an example. Once G_4 commits, G_1 must have committed. We first examine the legality of the read operations and their values in the log for G_2. Suppose a restart of G_2 occurs after G_4 commits. This restart cannot possibly be caused by G_1, since G_1 has already committed. In this example it cannot be caused by G_4 either, since G_4 has also already committed. Thus the only possibility is that G_2 itself is the originator of the restart. This will not make G_2 perform different operations from those stored in the log. Thus the read operations and their values in the log for G_2 are legal once G_4 commits. Using the similar arguments, we can

say the same thing about G_3. This means once G_4 commits, the values it installed will not be invalidated.

It is illustrative to compare Figure 7.3 b and Figure 7.4 a to see how the application of MODIFY resolves the problem. In both situations, G_3 is a firm node with G_4 and G_5 being two of its hidden children. In Figure 7.3 b G_1 can be restarted after G_4 and G_5 commit. We have seen this may make the read operations of G_3 illegal. However, in Figure 7.4 a, G_1 is a parent of G_4 and G_5, and hence must commit before them. In other words, after G_4 and G_5 commit, G_1 cannot possibly be restarted.

Chapter 8

Discussion

8.1 Performance

In the proposed protocol, the execution of the commit operations of the subtransactions of a global transaction follows the partial order in the relaxed dependency graph. In committing a subtransaction, the coordinator first sends the commit message to the site where the subtransaction is executed, and then waits for the confirmation. It then sends the commit signal to the next eligible ones based on the partial order. This is similar to a two phase commit protocol in the sense that after it sends the first message the coordinator must wait for the response to decide what actions it must take next. The difference is that in a two phase commit protocol all the participants execute the commit operations in parallel while in the proposed protocol the participants execute the commit operations in a predefined (partial) order. Thus in committing a global transaction the proposed protocol may be slower than those which simulate the traditional two phase commitment. On the other hand, in the proposed protocol the coordinator sends

only one message per participant (i.e., commit signal) while in a two phase protocol the coordinator sends two messages per participant (i.e., prepare and commit signals). Thus the proposed protocol has better message overhead.

A merit of our protocol is that it never needs to run compensation transactions. As mentioned before, even if compensating an already committed subtransaction is possible in some specific context, it is comprehensible that developing such compensating transactions is an expensive task. This is because at the time when a committed subtransaction need to be compensated for, its effects may have already been propagated to the other local databases. Compared with compensation, resubmitting an aborted subtransaction is more realistic and cost-effective. As shown in the previous context, a non-firm subtransaction requires almost no recovery actions. It is perceivable that in most cases, non-firm subtransactions account for a majority number of the subtransactions in a global transaction.

From the specification the proposed protocol is non-blocking for any non-firm subtransactions. It may or may not be blocking for a firm subtransaction depending upon the time the site failure occurs. If the site fails after it receives the commit signal from the coordinator, then after the site comes up the subtransaction will be either recovered or committed without consulting with the coordinator. If the site failure occurs before it receives the commit signal, then after the site is repaired the GRMGR must wait until the communication with the coordinator is resumed. In the mean time, a local transaction may be denied the access to some data items if it can possibly introduce an inconsistency.

8.2 Local Autonomy

One of the expenses that are paid by the commitment protocol is that in two occasions local autonomy is compromised to some extent. One occasion is when process L-HANDLER is used to restrict the access of local transactions to the data items during the recovery period. The other is when a site is repaired from the failure, and the commit signal is found in the global log for a subtransaction, the GRMGR must inquire the LDBMS about the status of that subtransaction. In both occasions, however, it is believed that the compromises are essential and not unreasonable. For the first case, based on the study on issues relating to failure recovery, it has become increasingly clear that to ensure atomicity some kind of restrictions must be imposed on the access to certain data items by local transactions during the global recovery period. This principle underlies almost all the protocols developed so far that handle failure recovery, (except for those using compensation). Compared with those protocols which divide the data items into the globally updatable and the locally updatable, or those that exclude all local transactions from accessing the local database, the violation of local autonomy by the proposed protocol is to a lesser extent. This is due to two salient features of our protocol. First, the violation of local autonomy mentioned above only happens in the recovery of a firm subtransaction, which in most cases should account for only a small minority. Second, for a firm subtransaction that is under the recovery, only those local transactions whose write sets intersect with the read set of the firm subtransaction will be rejected.

In the second case, such a 'violation' should be even more acceptable than the first one since all that is required by the GRMGR is the information about the status of a subtransaction. Supplying this piece of information should post no problem to the LDBMS since in the normal case, it is supplied anyway by the LDBMS whenever the subtransaction commits or aborts locally. In the face of failures, the information supplied by the LDBMS may be intercepted by a failure, forcing the GRMGR to make such an inquiry after the failure is repaired. (We say that this is a violation of local autonomy since the interface provided by an LDBMS may have to be expended to allow a GRMGR to make such an inquiry.)

8.3 Implementation of L-HANDLER

As specified in Section 6.3, L-HANDLER enforces access control over local transactions using the information about the write set of a local transaction or the read set of a firm subtransaction. In general, a global transaction is written in a multidatabase language but a local transaction is written in a language provided by the local database. Usually these two kinds of languages are different, (both syntactically and semantically.) Thus in order to compare the read sets of subtransactions with the write sets of local transactions, we must either translate the local transaction into a transaction written in the multidatabase language or translate the subtransaction into a transaction written in local database languages. Among these two approaches, We are in favor of the latter. This is because the former not only violates the local autonomy, but also is hard to achieve since the

local database model underlying a local database language is usually semantically poorer than that underlying a multidatabase language. A translation from the local database level to multidatabase level would require additional mechanism to *enrich* the semantics of a local data mode. On the other hand, a translation from the multidatabase level to local database level is relatively easier to achieve, and this must be done anyway before a subtransaction can be executed at the local database, whether or not we use an L-HANDLER.

Chapter 9

Conclusion

In this thesis, a failure recovery scheme for multidatabase systems is proposed. The scheme enforces a commit order among subtransactions which can be derived from the dependency relations over the subtransactions of a global transaction. To achieve failure atomicity, an aborted subtransaction is allowed to be resubmitted. To resolve the problems arising from cyclic dependency, certain restrictions are put on some subtransactions while no restriction on the others. In this way we can still use the commit order in the face of cyclic dependency while at the same time reduce the loss of local autonomy.A special class of value dependencies, namely access dependencies, is defined and the impact it has on the failure recovery schemes using resubmission is discussed. Possible solutions are provided to the problems arising from applying resubmission as a platform of failure recovery when access dependency is possible.

Some issues deserve future study. For example, how do we construct a formal model for the transaction execution based on resubmission? How do we develop efficient mechanisms to derive the dependency relation among the subtransactions

of a global transaction? What is the impact that the different choices of key sets have on the performance of the proposed protocol? The solutions to these problems will give deeper insight into the theoretical soundness and practical performance of resubmission-based protocol for failure recovery in MDBSs.

REFERENCES

1. R.Alonso, H.Garcia-Molina, and K.Salem. " Concurrency control and Recovery for Global Procedures in Federated Database Systems " In IEEE Data Engineering,pp 5-11,Sep 1987.

2. C.Beeri, P.Bernstein, and N. Goodman. " A Model for Concurrency in Nested Transactions Systems " Journal of the ACM, Vol.36,No.2,pp 230-269,1989.

3. Joel S. Berson. " Generic RDA Editor " Information Processing Systems-Open Systems Interconnection - Remote Database Access - Part 1: Generic Model, Service, and Protocol ISO/IEC JTC 1/SC 21 WG3 ISO,pp 20-35,1990.

4. Philip A. Bernstein, Vassos Hadzilacos and Nathan Goodman. " Concurrency Control and Recovery in Database Systems " Addison-Wesley Publishing Company, ISBN 0-201-107155, U.S.A,1987.

5. Ken Barker and M.Tamer Özsu. " Reliable Transaction Execution in Multidatabase System " In Proceedings of 1st Workshop on RIDE,pp 344-347,1991.

6. Yuri Breitbart and Avi Silberschatz. " Multidatabase Update Issues " In Proceedings of ACM SIGMOD Conference, Chicago,pp 135-142,May 1988.

7. Yuri Breitbart and Avi Silberschatz. " Multidatabase Systems with a Decen-

tralized Concurrency Control Scheme " IEEE Distributed Processing Technical Committee Newsletter, Vol.10,No.2,pp 35-41,1988.

8. Yuri Breitbart, A.Silberschatz and G.Thompson." Reliable Transaction Management in a Multidatabase System " Technique Report, University of Kentucky,pp 33-56,1990.

9. Yuri Breitbart, D. Georgakopoulos, M. Rusin Kiewicz and A. Silberschatz. " Rigorous Scheduling in Multidatabase Systems " In Workshop in Multidatabases and Semantic Interoperability,pp 79-101,Oct 1990.

10. Y.Breitbart,A.Silberschatz and G.Thompson." Transaction Management Issues in a Failure-Prone Multidatabase System Environment " The VLDB Journal, Vol.1,No.1,pp 121-154,July 1992.

11. Divyakant, Abbadi and Ambuj." Consistency and Orderability: Semantics-Based Correctness Criteria for Databases " ACM Transaction on Database System, Vol.18,No.3,pp 231-266,Sep 1993.

12. Weimin Du and Ahmed K.Elmagarmid." Quasi Serializability: A Correctness Criterion for Global Concurrency " Control in InterBase Proceedings of 15th VLDB Conference,Amsterdam,pp 347-355, August 1989.

13. A.K.Elmagarmid and W.Du." A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems" In Proceedings of IEEE International Conference on Data Engineering, Los Angeles,pp 254-278,Feb 1990.

14. A.K.Elmagarmid and A.A.Helal. " Supporting Updates in Heterogeneous Distributed Database Systems " In Proceedings of the International Conference on Data Engineering,pp 564-569,1988.

15. A.K.Elmagarmid and M.Rusinkiewicz." Critical Issues in Multidatabase Systems " Technique Report CSD-TR-966,pp 36-71,March 1990.

16. K.Eswaran, J.N.Gray, R.Lorie and I.Traiger." On the Notions of Consistency and Predicate Locks in a Database System " ACM Transactions on Computer Systems, Vol.19,No.11,pp 624-633,1990.

17. Abdel Aziz Farrag and M. Tamer Ozsu." Using Semantic Knowledge of transactions to Increase Concurrency " ACM Transactions on Database Systems, Vol.14,No.4,pp 162-187,Dec 1989.

18. D. Georgakopoulos." Multidatabase Recoverability and Recovery " In Proceedings of of 1rd Workshop on RIDE,pp 348-355,1991.

19. D. Georgakopoulos and M. Rusinkiewicz." Transaction Management in Multidatabase Systems " Technique Report UH-CS-89-20, Department of Computer Science, University of Houston,pp 29-46,Sep 1989.

20. H.Garcia-Molina and K.SALEM." Sagas " In Proceedings of the ACM SIGMOD 1987 Annual Conference, ACM Press,pp 249-259,May 1987.

21. J.Gray." The Transaction Concept: Virtues and Limitations " In Proceedings of the International Conference on Very Large Databases, Cannes,

France,pp 144-154,Sep 1981.

22. V.Gligor and G.L.Luckenbaugh." Interconnecting Heterogeneous Database Management Systems " IEEE Computer, Vol.17,No.1,pp 213-266,Jan 1984.

23. V. Gligor and R. Popescu-Zeletin." Transaction Management in Distributed Heterogeneous Database Management Systems " Information Systems, Vol.11,No.4,pp 287-297,1986.

24. H.K.Korth, E. Levy and A.Silberschatz." A formal Approach to Recovery by Compensating Transactions " In Proceedings of VLDB Conference,pp 95-106,1990.

25. Yahiko Kambayashi." Integration of Different Concurrency Control Mechanisms in Heterogeneous Distributed Databases " In Proceedings of Second International Symposium on Interoperable Information Systems,pp 313-320,1988.

26. E.Moss." Nested Transactions, an Approach to Reliable Distributed Computing " The MIT Press,pp 59-87,1985.

27. C.Pu." Superdatabases for Composition of Heterogeneous Databases " In Proceeding of the International Conference On Data Engineering,pp 548-555,Feb 1988.

28. K.Salem, H.Garcia-Molina, and R.Alonso." Altruistic Locking: A Strategy for Coping with Long-live Transaction " Technical Report CS-TR-087-87,pp

101-135,April 1987.

29. Kazuo Sugihara. " Concurrency Control Based on Distributed Cycle Detection " In Proceedings of IEEE 3rd International Conference in Data Engineering,pp 267-274,1987.

30. John Shillington and M.Tamer OZsu." Semipermeable Transactions and Semantics-Based Concurrency Control for Multidatabases " 0-8186-3710-2/93 IEEE,pp 213-267,1993.

31. J. Tang." Using Dummy Reads to Maintain Consistency in Heterogeneous Database Systems", In Proceedings of 3rd International Workshop on Future Trends of Distributed Computing Systems,pp 312-317,April 1992.

32. Antoni Wolski and Jari Veijalainen." The 2PC Agent Method and its Correctness " Tiedotteita Meddelanden Research Notes,pp 23-78,1992.

33. Antoni Wolski and Jari Veijalainen." 2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogenous Multidatabase " In Proceedings of PARBASE-90 Conference, pp 49-78,Feb 1990.

34. A. Zhang and J. Jing." On Structural Features of Global Transactions in Multidatabase Systems " In Proceedings of 3rd Workshop on RIDE,pp 199-206,1993.

35. Murhy Devarakonda, Bill Kish and Ajay Mohindra. " Recovery in the Calypso File System " ACM Transactions on Computer Systems, Vol.14,No.3,

pp 287-310,August 1996.

36. Flaviu Cristian, Bob Dancey, Jon Dehn. " Fault-Tolerance in Air Traffic Control Systems " ACM Transactions on Computer Systems, Vol.14,No.3, pp 265-287,August 1996.

37. Rohit Chandra, Ding-Kai Chen, Robert Cox, Dror E.Maydan, Nenad Nedeljkovic " Data Distribution Support on Distributed Shared Memory Multiprocessors" ACM SIGPLAN Vol.32,No.5,pp 334-346,May 1997.