TRADING CONTROL AUTONOMY FOR RELIABILITY IN MULTIDATABASE TRANSACTIONS

Nandit Soparkar, Henry F. Korth,
and Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-05 February 1991

Trading Control Autonomy for Reliability in Multidatabase Transactions *

Nandit Soparkar Henry F. Korth Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-1188 USA

Abstract

This paper deals with the problem of ensuring *correct* (i.e., atomic, serializable, and durable) transaction executions in a distributed multidatabase system where few changes are permissible in the constituent local database systems to accommodate the demands of the distributed environment. The degree of control over the transactions by the sites (execution autonomy) and by the multidatabase software (control autonomy) are examined to highlight the trade-offs necessary to obtain correct executions.

We propose techniques that infringe upon control autonomy in order to provide fault-tolerant transaction management without restricting the types of transactions allowed, and which need minimal changes to be made to the existing systems. These techniques allow a large number of concurrency control protocols to be handled, and local execution autonomy to be preserved. Our proposed protocols tolerate failures at a level comparable to traditional distributed database management systems. Moreover, our scheme exhibits the desirable properties of avoiding global deadlocks, and scalability.

1 Introduction

A multidatabase system (MDBS) is an integrated system consisting of several database management systems (DBMSs) that allow user transactions to access data located in the constituent heterogeneous hardware and software environments. An MDBS is expected to provide the integration of the heterogeneous environment into a new, unified system with minimal changes made, if any, to the underlying systems. The pressing practical importance of MDBSs has recently attracted the attention of the research community. There are several important problems that need to be addressed in the design of an MDBS, such as data translation for syntactic and semantic homogeneity, user interfaces, security, transaction management issues, etc. (e.g., see [7, 13, 14]). In this paper, we restrict our attention to transaction management issues.

1.1 Transaction Processing in an MDBS

The MDBS may be regarded as a distributed system — with one DBMS at each site, and any interaction between the sites being effected via message-passing. There are two types of transactions that execute in the system:

*Work partially supported by NSF grants IRI-8805215, IRI-9003341, and by grants from the IBM corporation.

- **Local Transactions**, those that access data at a local site only. The majority of these transactions are expected to arise from application programs that existed prior to the integration.
- **Global Transactions**, those that access data at several different sites. Global transactions execute by submitting subtransactions to some or all of the local DBMSs.

It is assumed that each local DBMS ensures serializability and handles local deadlocks (either by avoidance or detection). Each local DBMS is responsible also for recovery from failure of local transactions and failure of its own local site.

Desirable features that an MDBS should provide for transaction management include the following [14]:

1. Restrictions on data access by global transactions should be minimized. Likewise, the local transactions at a site should be permitted to access any data residing in the local database.
2. The MDBS should be designed to provide serializable executions over *both* local and global transactions.
3. The MDBS should provide a means to execute global transactions *atomically* — that is, all the subtransactions of a global transaction should commit, or all should abort. The MDBS must manage global concurrency control and deadlock detection or avoidance.
4. The MDBS should guard against the violation of *local autonomy* (e.g., see [7, 14] and below).
5. The MDBS must ensure that failures of sites, or failures in the distributed environment, do not affect the correctness of the executions.
6. It should be possible to *scale-up* the system to accommodate the new additions.

It may not be possible to achieve all aspects of the above desirable features. The literature contains numerous proposals for ensuring various subsets of these features or alternatives to them.

1.2 Autonomy

The local autonomy of a DBMS may be regarded loosely as the extent to which the execution of local transactions can proceed unaffected adversely due to its integration into an MDBS. Local autonomy impacts a number of concerns (e.g., see [7, 14]). However, we restrict our attention to those aspects that are crucial to the correct transaction management, which are the degrees to which the DBMS and the MDBS have control over the transaction executions as described below.

The *execution* autonomy of a DBMS refers to the degree of control that a local transaction manager has over the transactions or subtransactions executing at that site. Execution autonomy is preserved if the local transaction manager of a site can either delay (indefinitely, if necessary) the execution of a transaction (or subtransaction) operation, or abort a local transaction at any time.

The *control* autonomy of a DBMS refers to the degree that the MDBS does *not* control the local transactions executing at that site. Thus, control autonomy is preserved for a DBMS if the MDBS may neither abort the execution of a local transaction, nor delay its operations in any manner (except in terms of normal contention for resources — e.g., see [3, 4, 16]). Certain applications may permit the infringement of this aspect of autonomy (e.g., as implicitly suggested in [11, 15]), and we exploit the possibilities in this paper.

1.3 Tradeoffs in an MDBS

It is not possible to meet all the desirable characteristics mentioned above for an MDBS if we insist on preserving the correctness criteria of atomic, serializable, and durable executions in our design. First, consider the question of ensuring atomic executions of the (global) transactions. To achieve this, a *global atomic commit* (GAC) protocol is necessary, and such a protocol requires a *prepared* state (e.g., see [1]). There are difficulties encountered in guaranteeing the durability of the changes made by a subtransaction without actually committing it. These become clear when one considers the possibility of an *internal* abort which allows a DBMS to abort a transaction or subtransaction at any time during its execution. In particular, this may occur even *after* a commit operation has been submitted by the transaction or subtransaction in question (but before the DBMS responds with an indication that the operation was successfully executed) [6]. Even if all the changes to be effected by a subtransaction are maintained by the MDBS in stable storage in order to allow reinstating the changes by repeated retrials (e.g., see [4, 16]), the problem persists. This may be traced to the preservation of control autonomy requirement wherein a local DBMS decides the local serialization order independent of the MDBS, and hence, may permit other local transactions that were to have been serialized *after* the subtransaction in question to access states of the database not affected by that subtransaction. This is exemplified next.

Consider the situation depicted in Figure 1 for a DBMS where control autonomy is preserved. Assume that all the operations of subtransaction T, except for the commit operation, have been executed, and that the commit operation has been submitted to the underlying DBMS after a GAC protocol decided to commit the corresponding global transaction. Also, assume that the GAC executed with the expected serialization order that had T preceding L_2. However, an internal abort occurs for T, and since the MDBS has no control over the execution of L_2, the DBMS may execute and commit L_2. Hence, even if the changes to be effected by T are retrieved from stable storage and resubmitted to the DBMS, the position of T in the actual serialization order will be erroneous.

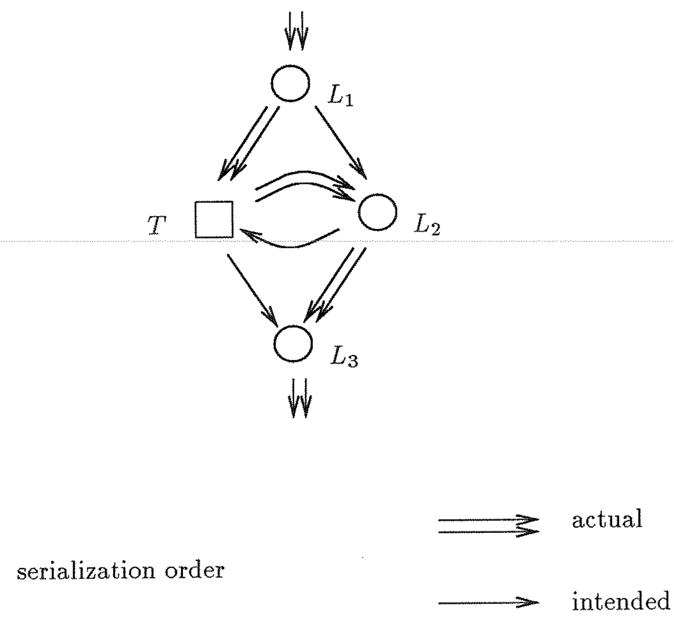

Figure 1: Example for Observation 1

To circumvent the above problem, the approaches used in [4, 16] are forced to place severe restrictions on the transactions regarding the data items that may be accessed. We thus make the following observation:

Observation 1. *Preserving execution and control autonomy, together with unrestricted types of transactions permissible in the system imply that a prepared state cannot be guaranteed.* □

Next, consider the question of ensuring the serializability of the transaction executions. The serialization orders of the transactions executing at a DBMS cannot be obtained explicitly by the MDBS, except under very specific conditions [12]. This leads to the potential for nonserializability among global transactions despite the enforcement of local serializability. For example, consider two global transactions, G_p and G_q, that access common data by means of subtransactions at two different sites, S_m and S_n. Suppose that G_p has the subtransactions T_{pm} and T_{pn} that execute at sites S_m and S_n, respectively, and similarly, G_q has the subtransactions T_{qm} and T_{qn} which execute at sites S_m and S_n, respectively. Since each site manages its transactions independently, the serialization orders imposed on the subtransactions may be (T_{pm}, T_{qm}) at site S_m, and (T_{qn}, T_{pn}) at site S_n. In such a situation, there is no serialization order possible for the global transactions G_p and G_q.

The MDBS cannot restrict attention to the conflicting operations (e.g., see [1]) of the global transactions alone if it is to maintain the serializability of the executions. Consider the situation depicted in Figure 2 for site S_m where we again assume that control autonomy is preserved. Let T_{pm} and T_{qm} be non-conflicting

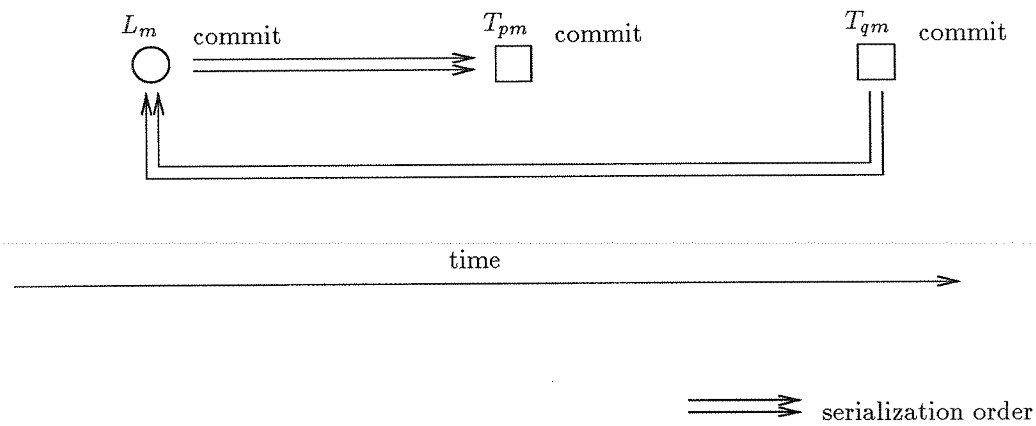

Figure 2: Example for Observation 2

subtransactions and assume T_{pm} commits first. The figure shows that this commit order may differ from the serialization order due to the presence of a local transaction L_m. As in the above example, a global serialization order may be impossible if the serialization order at site S_n differs with respect to that at site S_m. This illustrates the anomaly of a situation where two non-conflicting global transactions may be committed at different times, and yet they may be involved in non-serializable executions due to the presence of local transactions that the MDBS is not in a position to control. As pointed-out in [2], this is an important consideration because in such situations, ensuring the global atomic commitment of the transactions is useful in providing serializable executions only when a restricted class of *rigorous* schedules (usually not provided by typical centralized DBMSs) is generated by the DBMSs in question. In particular, it should be noted that the commonly used strict two-phase locking protocol, as implemented in many centralized DBMSs, is insufficient to guarantee serializability in these situations.

Therefore, we also make the following observation.

Observation 2. *Unrestricted types of transactions permissible in the system, and preservation of control autonomy, together imply that failure to enforce global serialization explicitly among the subtransactions may result in non-serializable executions.* □

1.4 Our Approach

The approach used in this paper is to infringe on one of the common denominators in the above observations, namely the control autonomy of the DBMSs.[1] This provides the MDBS designer some added control over the transaction executions. This approach has been implicitly considered by some other researchers as well, but

[1] The other common denominator regarding unrestricted transaction types was infringed upon in [4].

their approach to transaction management is different (e.g., the schemes in [11, 8] relies on compensations to achieve correct executions of transactions).

Our approach makes use of the existing means for concurrency control, recovery procedures, etc. that are provided by each of the underlying DBMSs, and is able to provide all the other desirable properties outlined above. We propose a simple transaction management scheme for the MDBS that makes minimal assumptions about the constituent DBMSs, uses serializability as the correctness criterion, and exhibits failure-resilience characteristics comparable to distributed DBMS technology. The salient features of our new scheme are:

1. All the desirable properties (with the exception of control autonomy) in an MDBS design as described above are provided.

2. The local DBMSs can use any concurrency control scheme that ensures the atomic, cascadeless, and serializable execution of the transactions submitted to it (including recovery in case of site failures [1]). Note that several commonly used concurrency control schemes, such as strict two-phase locking, provide these properties.

3. Our scheme is simple to implement since there is no need to access any control information from the transaction managers of the constituent DBMSs. Furthermore, our approach does not require changes to be made to the existing application programs. Thus, the costs associated with the development of our scheme are small.

4. Our scheme is resilient to local failures (e.g., transaction failures, system failures, etc. [6]). This is achieved by indirectly using the existing recovery mechanisms available at each site. Therefore, few new recovery techniques need to be developed. Our approach exhibits failure-resilience that is provided by a typical distributed DBMS.

5. Our scheme allows the integration of a new DBMS into an existing MDBS to be accomplished in a simple manner. The key to achieving these important features is that our approach allows the implementation of a distributed MDBS as opposed to a centralized one.

6. Our scheme permits the use of the original user interfaces for both the original DBMS as well as the MDBS at each site. This is advantageous since the user interfaces need not be redesigned for the integration.

2 System Structure

An MDBS consists of n sites, S_1, S_2, ..., S_n, interconnected by a computer network as shown in Figure 3. Each site S_i has a database system, $DBMS_i$, consisting of a local database, LDB_i, and a local transaction manager, LTM_i. We assume that $\bigcap_{i=1}^{n} LDB_i = \phi$ (i.e., no replicated data).

Each $DBMS_i$ supports the following common operations [1]:

1. **begin**: To indicate to LTM_i that a transaction has been initiated.
2. **insert()**: To insert a new data item in LDB_i.
3. **delete()**: To delete an existing data item from LDB_i.
4. **read()**: To read the value of an existing data item in LDB_i.
5. **write()**: To update the value of an existing data item in LDB_i.
6. **commit**: To commit a transaction.
7. **abort**: To abort a transaction.

Two operations that access a common data item are said to *conflict* if one of them is an **insert**, a **delete**, or a **write**.

We assume that the execution of the transactions submitted to LTM_i by the user programs is ensured to be *correct* in the following sense:

1. **Atomic.** Either all or none of the operations of a transaction are executed by LDB_i. In the LDB_i, all *committed* transactions have their effects reflected permanently, whereas none of the effects of *aborted* transactions appear.
2. **Serializable.** The *history* of any interleaved execution of operations of a set of committed transactions at $DBMS_i$ implies an acyclic local conflict *serialization graph* (SG) at site S_i (e.g., see [1]).
3. **Cascadeless.** The abort of a transaction does not require the aborting of other transactions. What this effectively means is that transactions do not read uncommitted values of data items [1].
4. **Durable.** For local failures [6], LTM_i ensures that all the transactions that were reported to the user programs as having been committed successfully have their effects permanently installed in LDB_i, whereas any active transactions are aborted.

5. **Deadlock Management.** Each LTM_i handles local deadlocks by either avoiding them, or by using some deadlock detection and recovery scheme.

The MDBS software is distributed among the n sites. The MDBS itself consists of n modules of software. Each module, $MDBS_i$, is located at site S_i running above $DBMS_i$. The $MDBS_i$ are interconnected by a communications network, but are otherwise independent of one another.

Once the MDBS is created, new data items may be added that are accessed by the global transactions. These are stored in the same manner as the original data items. Note that the actual data items placed in the various DBMSs may differ in their syntax and associated semantics. Since we are concerned more with the correctness of transaction management, we assume that the data conversion mechanisms are already available (e.g., see [7]), and hence, the MDBS may view all the data items to have homogeneous syntax and semantics. However, to keep the degree of data conversion low, a global transaction is executed as a set of subtransactions with at most one subtransaction per site — unlike the approach in [15] which uses single-site transactions. Each subtransaction is submitted to the local DBMS as a local transaction. That is, the DBMSs are not able to differentiate between the two types of transactions, which is a consequence of preserving local autonomy. Henceforth, we use the term *transaction* to refer to both the types, and we use the terms *local transaction* or *subtransaction* when we need to make the distinction.

All local and global transactions executed at a site S_i, are processed through $MDBS_i$. The $MDBS_i$ module is transparent to the user programs. That is, it provides the same interface to the user programs as LTM_i did prior to the integration. Similarly, $MDBS_i$ is transparent to LTM_i. Global transactions use the same interface as the original local transactions did. That is, the application programs need not distinguish between global and local transactions. The implementation issues regarding the transparency are further discussed in Section 8.

3 Transaction Management

A transaction is an ordered sequence of operations $op_1, op_2, \ldots, op_m$, where op_1 is **begin**, op_m is **commit** or **abort**, and op_i for $1 < i < m$ is a **read**, **write**, **insert**, or **delete** operation on some data item. We assume there are no *blind* **write** or **delete** operations. That is, each **write** or **delete** operation is preceded by a **read** operation that is generated by LTM_i on the corresponding data item.[2] The initiation of a transaction is effected by a **begin** operation, the completion is effected by one of **commit** or **abort** operations, and the active phase of the transaction constitutes the execution of the remaining operations. The execution of an

[2] This is a reasonable assumption since most systems first retrieve the data item which they subsequently update or delete. If this is not provided by LTM_i, the $MDBS_i$ can generate the necessary **read** operations instead.

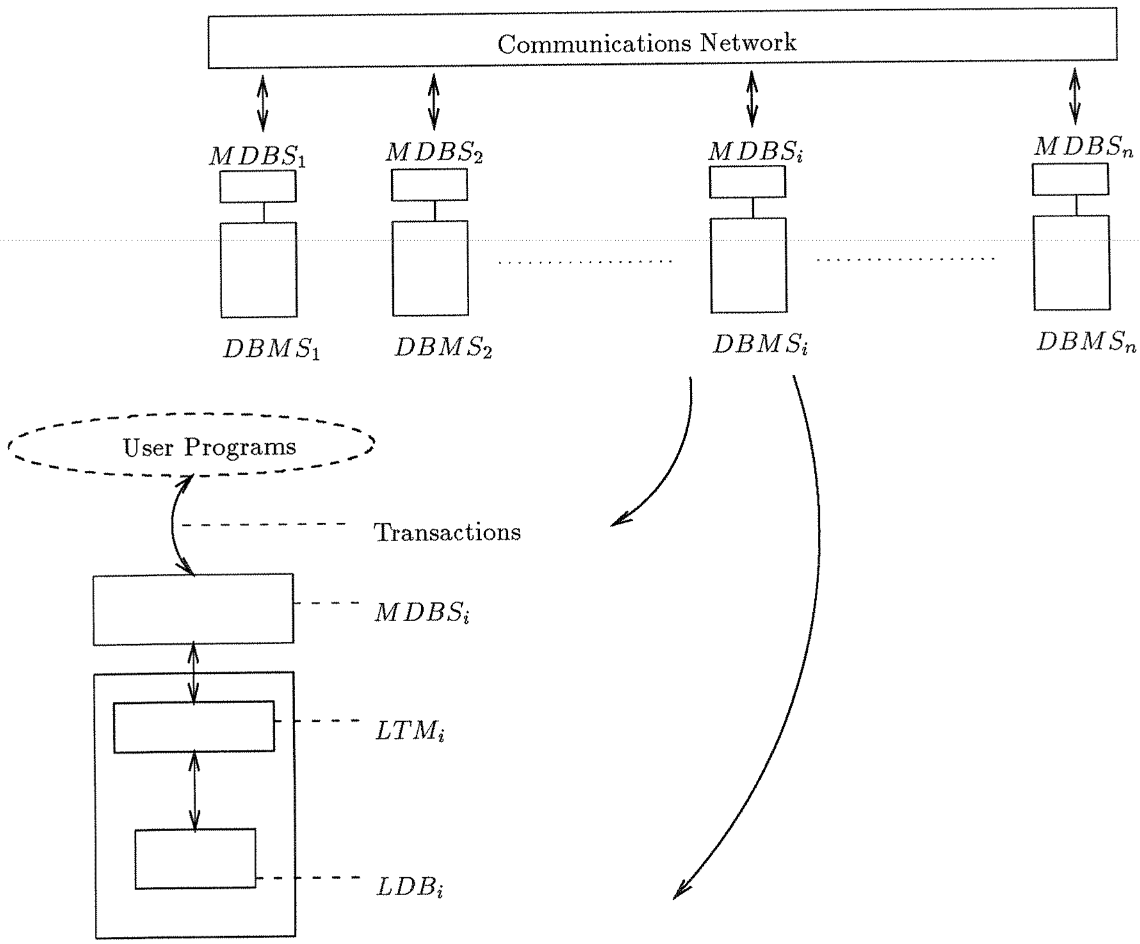

Figure 3: The MDBS System Structure

operation is initiated by its submission to $DBMS_i$ (or, equivalently, LTM_i), and the operation is said to be *fully* executed after LTM_i sends back a response for it. In particular, at each site S_i, we distinguish between an **abort** operation submitted by the user transactions (e.g., as a result of a logical error within the user transaction), and an abort that is enforced by $DBMS_i$ (e.g., to break a deadlock situation within $DBMS_i$). We call the former an *external* abort to distinguish it from the latter which is an internal abort. In order to ensure the correct execution of global transactions each $MDBS_i$ module maintains some control information as described below.

A user at a site S_i may submit any transaction, local or global, to $MDBS_i$. For a global transaction that is initiated at site S_i, site S_i is designated as the coordinator, and it initiates the requisite subtransactions at the other sites. The information regarding the various sites at which a global transaction executes is maintained in stable storage by the coordinator.

Under certain circumstances, an $MDBS_i$ may choose to abort an active transaction T_j forcibly by disregarding any operations of T_j as yet not submitted to $DBMS_i$, and by submitting an **abort** on the behalf of T_j to $DBMS_i$. The ensuing internal abort response by $DBMS_i$ is passed back to the user program which views the abnormal termination of T_j as an instance of the exercise of execution autonomy by $DBMS_i$. Alternatively, if no responses concerning the transaction under question have been passed back, the transaction may be restarted by $MDBS_i$. The circumstances under which $MDBS_i$ chooses to abort a transaction are determined by a pair of sets, $ch(T_j)$ and $pc(T_j)$ that are created for each subtransaction T_j executed at site S_i:

1. $ch(T_j)$: This set represents all the **ch**anges effected by T_j in LDB_i. Specifically, it contains all the **insert**, **delete**, and **write** operations of T_j together with the affected data items and values produced by the operations. This set is continually updated as the execution of T_j proceeds by including in it the relevant operations prior to submitting them to $DBMS_i$. This set is used to ensure that if subtransaction T_j *must* commit (for the reason that the associated global transaction is decided to be committed), it can do so as explained below.

2. $pc(T_j)$: This is a set of transactions executing at site S_i that **p**otentially **c**onflict with T_j *and* must be serialized after T_j. That is, taking $ch(T_j)$ into account, $pc(T_j)$ consists of transactions T_k executing at site S_i whose computation may be affected as a result of accessing the data items mentioned in $ch(T_j)$. As described below, the set $pc(T_j)$ is created toward the final stages of executing T_j, and until all the necessary steps have been taken to deal with T_j, $pc(T_j)$ is kept up to date. This set is used to choose the transactions to abort forcibly so as to ensure the requirements of GAC for global transactions. All these actions are made precise below.

The above sets are introduced to facilitate the description of how the transactions executing at a site S_i are affected by $MDBS_i$ to ensure that the overall execution of all the transactions remains correct. The key idea is to give the $MDBS_i$ control over when and if the various transactions executing at site S_i are allowed to commit. In all cases, the **commit** operation for a transaction is submitted by $MDBS_i$ to $DBMS_i$ only after the last but one operation of that transaction is fully executed.

3.1 Local Transaction Management

The **begin** operation of a local transaction T_k that is submitted to $MDBS_i$ at a site S_i initiates the execution of that transaction. Before the **begin** operation is submitted to $DBMS_i$, T_k is assigned a unique *id*, and the completion of any other book-keeping activities (as described in Section 8) for the purposes of maintaining the sets $ch(T_k)$ and $pc(T_k)$. Each subsequent operation is, likewise, submitted to $DBMS_i$ after completing any required book-keeping. The responses provided by $DBMS_i$ to the operations is passed back by $MDBS_i$ to the user program that submits T_k.

The final **commit** or **abort**, however, is handled differently. In case of an external abort, an **abort** operation is submitted to $DBMS_i$, and the response is returned back to the user program. Likewise, when $MDBS_i$ makes a decision to commit T_k, it submits the **commit** operation on behalf of T_k to $DBMS_i$, and passes back to the user program the response generated as for the other operations. To ensure the correctness of global transactions, $MDBS_i$ may delay the submission of the **commit** operation for T_k, or even abort T_k forcibly.

3.2 Global Transaction Management

A subtransaction T_j is executed at a site S_i by $MDBS_i$ in a manner similar to the local transactions up to the last but one operation. In addition, the $ch(T_j)$ set is created and updated during the execution of T_j. At the point where all the operations of T_j up to and including the last but one operation are fully executed, a *prepare-to-commit* message is expected by $MDBS_i$ from the coordinating site for the associated global transaction for T_j as part of a GAC protocol. On receipt of this message, $MDBS_i$ obeys the sequence of steps of the *MDBS protocol* specified below.

The MDBS protocol is run locally for only one active subtransaction T_j at a time,[3] and consists of the following five steps:

1. The set $pc(T_j)$ is created consisting of all transactions that have operations outstanding (i.e., operations that have been submitted to $DBMS_i$, but the response to which have not yet been received by $MDBS_i$) that conflict with $ch(T_j)$. Any subsequent operations conflicting with $ch(T_j)$ that are submitted to $DBMS_i$ on behalf of a transaction T_k result in the inclusion of T_k in $pc(T_j)$.

2. A unique data element ser_i that is maintained in LDB_i is accessed by a **write** operation submitted by $MDBS_i$ on behalf of T_j. This operation is not included in $ch(T_j)$, and the value written is immaterial. After obtaining a successful response to this operation, the MDBS protocol proceeds. Otherwise, if the response is an internal abort (or an unsuccessful **write**), then T_j is aborted, and the sets $ch(T_j)$ and $pc(T_j)$ are discarded.

3. The set $ch(T_j)$ is saved on stable storage, and a *ready-to-commit* message is sent to the coordinator for T_j. The saving of $ch(T_j)$ is the point when T_j (or equivalently, $MDBS_i$) is said to have entered

[3] Although this requirement can be relaxed, to keep the presentation simple, the MDBS protocol handles the subtransactions one at a time.

the *prepared* state. That is, $MDBS_i$ guarantees that if the coordinator should choose to commit the global transaction corresponding to T_j, then the effects of T_j on LDB_i will be permanently installed regardless of any failures.

4. Upon receipt of a final commit or abort decision from the coordinator for T_j, the corresponding operation for T_j is submitted to $DBMS_i$ by $MDBS_i$. In case the **commit** operation submitted to $DBMS_i$ results in an internal abort, all the transactions in $pc(T_j)$, at least, are forcibly aborted as described above. After this, a transaction that consists of the operations contained in $ch(T_j)$ is created by $MDBS_i$, and this transaction is repeatedly resubmitted to $DBMS_i$ until it is successfully committed.

5. Finally, $ch(T_j)$ is either removed from the stable storage, or an indication that it is no longer needed is also stored. Also, the sets $ch(T_j)$ and $pc(T_j)$ are discarded from the memory.

Note that in the MDBS protocol, the data item ser_i provides a conflict between each pair of subtransactions executing at site S_i in a manner similar to the approach in [5]. However, we do not use an optimistic approach to the concurrency control, and hence, we disregard the particular value written by the operation submitted to access ser_i. Also note that resorting to the repeated retrials of a transaction so as to ensure that its effects are installed in LDB_i is very similar to a set of *idempotent* redos for recovery purposes [9].

4 Global Atomic Commitment

The transaction management scheme described in Section 3 employs a GAC protocol to ensure the atomic commitment of the global transactions. A GAC protocol depends upon the availability of a prepared state [1]. By infringing upon the control autonomy of the sites as detailed in the MDBS protocol, we now demonstrate that a prepared state may be achieved. Once this is achieved, it is possible to use the traditional two-phase commit, or three-phase commit protocols (e.g., see [1]) as a GAC protocol.[4]

While the use of stable storage to save $ch(T_j)$ for a subtransaction T_j executing at a site S_i may appear to be sufficient to ensure a prepared state, this proves to be incorrect. As discussed in Section 1, the problems arise because T_j may be internally aborted at any time (which is associated with the preservation of execution autonomy as well), and that may lead to a situation where T_j may lose its intended position in the serialization order at site S_i. This is due to the fact that the commit decision for a global transaction reflects not only the atomicity of the global transaction, but also its commitment at a particular point in the serialization order at each site (see below). Thus, failures notwithstanding, it must be ensured that T_j can be committed if required at a particular position in the serialization order at site S_i by $DBMS_i$.

[4] Note that these protocols may entail *blocking* behavior which is discussed in Section 7.

By aborting all transactions that *read from* T_j (e.g., see [1]), it can be ensured that the position of T_j in the serialization order is not compromised. Aborting those transactions also ensures that there will be no contention for the data items in $ch(T_j)$ so that the repeated retrials of T_j will eventually succeed. Note that it is unnecessary to abort all the transactions in the transitive closure of the *read from* relation due to the restriction that the schedules are conflict serializable and cascadeless. Similarly, a transaction that accesses a data item that is only read by T_j need not be aborted. The question is whether $pc(T_j)$ indeed includes all the transactions that may read from T_j. To see that this is the case, consider all the transactions *excluded* from $pc(T_j)$. We claim that none of these transactions reads from T_j. Indeed, these transactions fall into the following three categories:

1. Committed transactions — since by the cascadelessness property of $DBMS_i$, they could not have read from the uncommitted T_j.

2. Transactions without conflicting operations — these could not have read from T_j for obvious reasons.

3. Transactions with no outstanding operations — since if it had a conflicting operation that had been fully executed already, it could not have read from the uncommitted T_j due to the cascadeless executions. Note that if a conflicting operation is submitted to $DBMS_i$ after $pc(T_j)$ is generated, then the corresponding transaction *is* included in $pc(T_j)$.

As the MDBS protocol progresses, notice that $pc(T_j)$ does not lose any elements. Furthermore, even if the transactions in $pc(T_j)$ fully execute all but their final operations (access to ser_i), they are not permitted to commit. This requirement may appear unusual, but a problem that is described below makes it easier to see why this is essential.

Consider the following subtle problem that may arise if a prepared state were declared for T_j *prior* to the full execution of the access on ser_i. In that case, it may happen that $DBMS_i$ decided to abort T_j internally after fully executing all the operations of T_j that preceded the access of ser_i. The notification of the abort may reach $MDBS_i$ after a short delay, and that could result in the execution of some conflicting operations of transactions that were intended to be serialized *after* T_j by $DBMS_i$. As far as the coordinator for T_j is concerned, it makes a GAC decision based on those transactions *not* having been executed before T_j. Such transactions may get committed, or may exhibit no outstanding conflicting operations at the time that $pc(T_j)$ is being generated. The full, and normal, execution of the access of ser_i serves as an indicator that T_j was not aborted internally prior to the complete generation of $pc(T_j)$. The same reasoning holds to justify the delay in completion of any transaction placed in $pc(T_j)$.

We are now in a position to state the following result that is closely related to Observation 1 of Section 1 concerning the feasibility of a prepared state (and hence, a GAC protocol).

Theorem 1. *The MDBS protocol provides a prepared state for a transaction, but it does not ensure full control autonomy.*

Proof Sketch: Only the situation that needs to be considered is when a GAC protocol concludes with a commit decision for the subtransaction T_j. It can be shown that the history generated using the MDBS protocol corresponds to one of the following two correct histories.

Firstly, in the case that there is no internal abort upon submission of the commit operation for T_j, the corresponding correct history is simply the one that includes both the committed T_j as well as the transactions that read from T_j. The second case for consideration is when an internal abort occurs upon submission of the commit operation for T_j. The corresponding correct history in that situation includes the committed subtransaction T_j, while the transactions in $pc(T_j)$ occur with an aborted termination (which could be ascribed to internal aborts as well). □

By using the prepared state provided by the MDBS protocol, a GAC protocol may be designed using any standard protocol used for distributed DBMSs (e.g., see [1]). Note that the MDBS protocol is described in terminology used for the standard two-phase commit protocol [1].

5 Serializability

Our transaction management scheme produces serializable executions over both global and local transactions. We restrict attention to transactions that are committed since serializability is a property dealing with committed transaction histories [1].

Consider *synchronization intervals* for a local history — a notion similar to synchronization events (e.g., see [12]) for a local history. An interval is a contiguous portion of a local history that may be associated with a transaction executing at the local site. Such an interval should not overlap with any other similar interval if the corresponding transactions have a path between them in the local SG. Also, the intervals should have the property that their order of occurrence in the schedule should be the same as the local serialization order of the transactions with which they are associated. Consider the synchronization intervals of subtransactions executing at each site. If a protocol such as 2PC is used, the intervals can be synchronized to yield serializable executions.

As an example, consider the use of 2PL and 2PC as depicted in Figure 4. The locked interval for a subtransaction that obeys 2PL (i.e., the duration for which all the locks are held) serves as the necessary

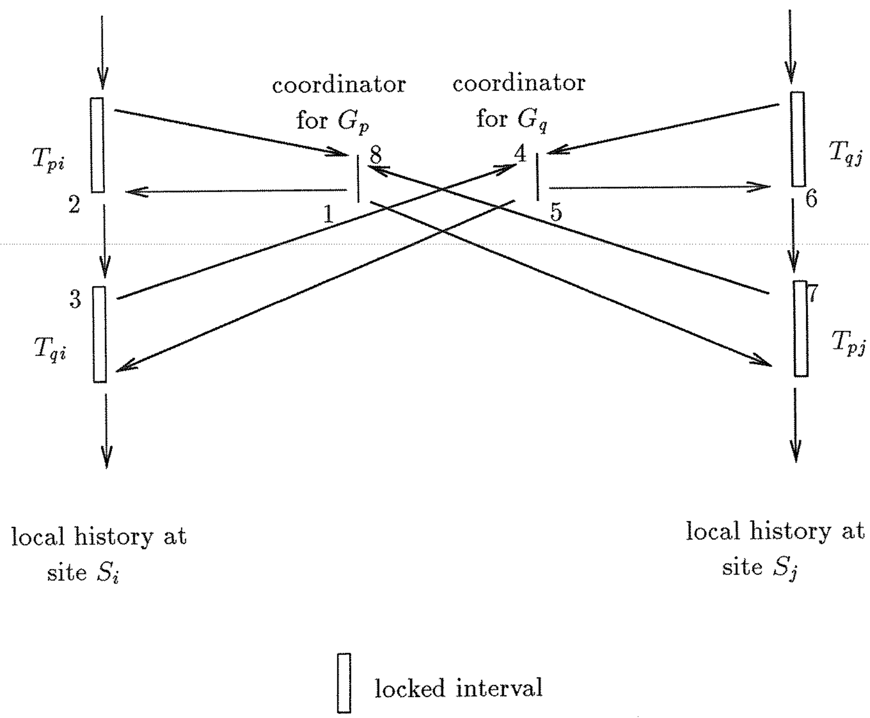

Figure 4: Correct Synchronization using 2PL and 2PC

synchronization interval. If the locks are not released prior to the completion of the 2PC protocol, a non-serializable execution cannot occur. This is because a cycle of events such as (1 2 ... 8 1) is impossible since the distributed history of events in a system must follow a partial order [10]. The same reasoning can be extended to more than two global transactions and several sites.

Theorem 2. *The MDBS protocol ensures serializability over all transactions, both global and local.*

Proof Sketch: Consider the history at a local site S_i. A synchronization interval consists of the portion of the history from the point immediately after the last operation of a subtransaction up to the commit operation. Due to the assumed cascadeless property of the local history, and the accesses to ser_i, this interval has the properties for a synchronization interval when the subtransactions are considered. Firstly, the accesses to ser_i ensure that there will exist a path in the local history between any two locally executing subtransactions. And since the executions are cascadeless, the complete execution of all the operations preceding the commit operation for any subtransaction T_{qi} must occur only after the commit operation of any subtransaction T_{pi} that precedes T_{qi} in the local serialization order. Hence, the intervals suggested do not overlap, and also, they occur in the same order as the serialization order for the local history.

Having identified the synchronization intervals, the MDBS protocol provides the means to engage in the necessary synchronization by means of a GAC protocol. Thus, from the preceding discussion and the results in the literature (e.g., see [5, 12]), serializability is ensured over all the transactions. □

Note the close correlation between the statement of Observation 2 of Section 2, and Theorem 2 which achieves serializability by the enforcement of local conflicts by accesses to the unique data items maintained at each site.[5]

6 Deadlocks

We now discuss the issues of global and local deadlocks. First, let us consider deadlocks within a site S_i. By our earlier assumption, any possible deadlocks within $DBMS_i$ are handled by LTM_i. The only other case of potential local deadlock is during a prepared state since the transactions in $pc(T_j)$ for a subtransaction T_j wait for the completion of T_j. However, since T_j does not directly or indirectly wait for the transactions in $pc(T_j)$ through a *local* sequence of waits, there can be no such deadlocks.

The situation changes when the potential for deadlocks that span more than one site is considered. These can occur when the transaction executions at the different sites give rise to non-compatible serialization orders for the global subtransactions. As an example, in the proof of Lemma 1, consider the situation where $T_{pm} \xrightarrow{+} T_{qm}$ at site S_m, and $T_{qn} \xrightarrow{+} T_{pn}$ at site S_n. In this case, the coordinator for G_p may execute all the operations pertaining to the subtransaction T_{pm}, and similarly, the coordinator for G_q for subtransaction T_{qn}. These subtransactions will await the initiation of the GAC protocol by their respective coordinators before actually committing. However, the coordinators will also be awaiting the completion of the operations by T_{pn} and T_{qm}, respectively, so as to be able to proceed with the GAC. But, the subtransactions T_{pn} and T_{qm} have to await the commitment of T_{qn} and T_{pm}, respectively, before proceeding. This is a classic deadlock situation involving the four subtransactions and their coordinators.

To deal with the above problem, timeouts are employed to ensure that a coordinator suffers only a finite or bounded amount of waiting periods. In effect, deadlock resolution is achieved by aborting the global transactions that are involved in possible distributed deadlocks. Our approach has the drawback of possible starvation which we do not address here.

[5]The statement of Observation 2 suggests that infringing upon control autonomy alone may suffice to provide serializability (i.e., without taking recourse to enforced conflicts between the subtransactions [5]). While that is possible, it may require an $MDBS_i$ module to carefully examine the local transactions, and effectively, manage the bulk of concurrency control — an inefficient alternative since the transactions would encounter delays at the LTM_i as well.

7 Failure Resilience

The MDBS needs to deal with a number of different possible failures in the system. These can be separated into two categories. The first category deals with failures that only affect a single site locally. These correspond to the failures that arise in a centralized DBMS. The second category deals with failures which arise as a result of the distributed nature of the MDBS, and these directly affect the global transactions, and indirectly, the local transactions.

7.1 Local Failures

Consider the various failures that may occur at site S_i, and the associated recovery activities provided by $MDBS_i$ [6]. Since subtransactions of global transactions execute only at the site at which they are initiated, our discussion applies to both the local and the global transactions executing at site S_i.

- **Action failure.** This is a failure that is anticipated by the user programs (e.g., missing data, resource limitations, etc.), and corresponds to an operation of a transaction that is not successfully completed. In such cases, no recovery activity need be done by $MDBS_i$, and the response provided by LTM_i is passed-on to the user program.

- **Transaction failure.** This failure occurs if a transaction must be aborted for some reason not anticipated by the user program. As discussed in Section 3, the **abort** may be submitted by the user program, or it may be an internal abort. There is no specific recovery activity that $MDBS_i$ is required to do — except in the case that the internal abort occurs after a prepared state for a subtransaction is reached. In such a situation, the MDBS protocol resubmits the transaction (see item 4 of the protocol).

- **System failure.** A failure of this type is characterized by the loss of all information stored in volatile memory. However, the information stored in non-volatile memory remains intact. The MDBS protocol ensures that at most one subtransaction T_j had reached the prepared state without having been dealt with completely. If such a T_j exists, the system, upon recovery, obtains $ch(T_j)$ from the stable storage. Since all transactions in $pc(t_j)$ must have been active at the time of the failure, they are aborted by the recovery mechanism of $DBMS_i$. Thus, $pc(T_j)$ is initialized to the empty set. Then $MDBS_i$ sends the *ready-to-commit* message again for T_j, and normal processing is resumed. These actions suffice to place the site in a prepared state for T_j, and the GAC proceeds from this point.

  In the context of volatile memory failures, there is the more subtle problem of what happens if such a failure occurs *after* $DBMS_i$ commits a transaction but before $MDBS_i$ records the response in stable storage. A similar problem is the occurrence of such a failure after $MDBS_i$ has recorded the completion of a transaction by $DBMS_i$ in stable storage but before it informs the user program of the completion. These problems are related to the implementation details of $MDBS_i$ as regards its failure-resilient transparency to the user programs and $DBMS_i$ by the use of requisite *handshaking* protocols etc., and these issues are discussed in Section 8. Suffice to note here that the problems are an extension of what happens in the case that such a failure occurs during the interaction of user programs and $DBMS_i$ before the integration of the system into an MDBS.

- **Non-volatile storage failures.** These failures result in the loss of non-volatile memory, and we do not address them further beyond noting that standard methods to deal with these (e.g., archives, checkpoints, etc.) need to be used.

Note that $MDBS_i$ has very few recovery activities to perform directly, and most of the recovery management is relegated to $DBMS_i$.

7.2 Failures in the Distributed Environment

As is the case for *any* protocol for GAC in distributed environments with arbitrary non-malicious failures possible, our approach also exhibits *blocking* behavior wherein some data items in a local DBMS become inaccessible for the duration of a failure [1]. This is manifested in our scheme during the prepared state for a subtransaction T_j that blocks the data items in $ch(T_j)$ by the delay in the submission of *commit* operations for the transactions in $pc(T_j)$.

The problem of choosing alternative coordinator sites in case of site failures is present in our approach just as in the case of typical distributed DBMSs. It is not difficult to see that the same techniques that are applicable in the traditional schemes also apply to our scheme. In essence, by the availability of a prepared state, an MDBS designed according to our specifications permits the use of techniques developed for distributed DBMSs in the environment of an MDBS.

8 Implementation and Performance Issues

The two major issues in this section are the implementation of the $MDBS_i$ modules, and the techniques used to handle the sets in the $MDBS_i$ associated with the creation of prepared states. We also briefly discuss the issues of scalability of our scheme.

8.1 The $MDBS_i$ Module

We require that each $MDBS_i$ be transparent to both the user program as well as LTM_i. This has several implications. It assumes that the interfacing details between the user programs and LTM_i are known, so that $MDBS_i$ can emulate this interface. The assumption is not unreasonable since the LTM_i itself need not be modified – only an interface module need be developed.

Note that failures may adversely affect the $MDBS_i$ functions. Consider the passing of a message from LTM_i to the user program through $MDBS_i$. The message must be recorded in stable storage by $MDBS_i$ before being passed-on to the user program, lest the message be lost in the event of a failure. The potential exists for a failure between the time LTM_i sends a message and $MDBS_i$ records it stably. This is analogous to the situation in which a failure occurs between the time LTM_i sends a message and the time the user program records or displays the message. It is important that the $MDBS_i$ module duplicate whatever "hand-shaking" protocol is used by user programs with LTM_i to ensure that no additional failure vulnerabilities are introduced.

In many systems, handshaking is accomplished by means of messages sent with sequence numbers, and corresponding acknowledgements for these messages [6]. If it is ensured that the messages are recorded in stable storage during their passage through the $MDBS_i$, it is not difficult to see that reliable handshaking can be duplicated easily in such environments.

To enforce failure-resilience, we have noted the need to be able to store information in stable storage. One way to do this is to have a module that performs the necessary information storage on a device that is accessed independent of the $DBMS_i$ stable storage mechanisms. A different option is to use the database itself as the information storage device [15]. The information is saved using a transaction running in $DBMS_i$ that inserts the requisite record into the relation. Retrieving the information involves reading the same relation. The method works correctly because committed transactions have their effects stored in the database permanently — just as in the case of records saved in stable storage. The advantage of this method is that it is simple, existing resources are made use of, and hence, it makes for the inexpensive development of an MDBS. The obvious disadvantage is in the inefficiency that may result since every time such information needs to be accessed, it is done using a separate transaction.

8.2 Managing the Sets $ch(T_j)$ and $pc(T_j)$

In examining the alternatives to managing a prepared state for a subtransaction T_j executing at a site S_i, a distinct trade-off becomes apparent. On one hand, it is clearly desirable to delay operations submitted by the user programs the least amount of time within $MDBS_i$ prior to their submission to $DBMS_i$. This can be achieved if the operations are not subjected to close scrutiny and analysis in the $MDBS_i$ module. However, doing so may incur heavier penalties in the number of transactions that may have to be forcibly aborted to ensure the atomicity of global transactions as explained below. The trade-off arises from the results in Section 4 that indicate that instead of considering $pc(T_j)$, we may use for the same purposes a set $pc'(T_j)$ as long as $pc(T_j) \subseteq pc'(T_j)$. An extreme example of this is to maintain no $pc(T_j)$ sets at all, and instead, only to keep track of the active transactions. That is, the active transactions are regarded as constituting the set $pc'(T_j) \bigcup T_j$. Thus, in the situation where the forcible aborts of the transactions in $pc(T_j)$ becomes necessary, all the active transactions except T_j are aborted. This is similar to emulating a system failure.

On the other hand, a careful interpretation of the operations (and perhaps, the values of the data items involved as well) would certainly minimize the size of $pc'(T_j)$ — and thus, the number of forcible aborts. However, that would clearly increase the delays in the passage of the operations and the associated responses between the user programs and the $DBMS_i$ at each site.

Notice that a nice feature of our design is that each site is permitted to have its own policy as regards maintaining the size of the $pc'(T_j)$ sets. And similarly, different policies may be used at different times at the *same* site. Thus depending on the circumstances, the policy used can change to suit the performance needs.

Below, we describe a few policies for the management of the $pc'(T_j)$ sets which demonstrate some of the possible alternatives in the trade-off. The description is enumerated in the order of increasing complexity of interpretation of the operations, and the consequent decrease in size of $pc'(T_j)$. In each case, the previous schemes are part of the scheme under consideration. Assume that the set $ch(T_j)$ is stored as a sequence of *operation(data item, value)* elements, and each such element is included in $ch(T_j)$ prior to submitting the corresponding operation to $DBMS_i$.

1. Maintain only the set of active transactions. The set $pc'(T_j)$ is taken to be this set without the entry for T_j. Thus, every **begin** operation creates an element for the corresponding transaction in the set, and the full execution of the final operation for a transaction results in the deletion of the corresponding element for the transaction. For this scheme, $ch(T_j)$ need not be accessed to determine $pc'(T_j)$.

   This approach provides a particularly simple scheme to integrate DBMSs that use the commonly used strict two-phase locking protocol. In contrast to the schemes designed in [3, 4, 16], no restrictions need be placed on the types or the access patterns of the transactions. Each $MDBS_i$ module needs to maintain the list of active local transactions that may need to be aborted, and the changes of only those uncommitted subtransactions in stable storage that are in the prepared state.

2. For each active transaction, maintain a count of the total number of operations that are outstanding. For example, this may be stored in a location accessed by a hash function on the transaction *id*. In such cases, $pc'(T_j)$ consists of the set of transactions with non-zero counts. As in the above scheme, $ch(T_j)$ is not needed to generate $pc'(T_j)$.

3. For each active transaction that has non-zero outstanding operations, check if the outstanding operations conflict with $ch(T_j)$. This can be done by maintaining for each active transaction with outstanding operations a set of data items being accessed by those operations. Thus, to generate and maintain $pc'(T_j)$, the newly introduced sets are compared with $ch(T_j)$ to determine whether the corresponding transaction needs to be included in $pc'(T_j)$.

4. Schemes that involve an in-depth examination of the values pertaining to the data items accessed by the operations together with the semantics of the operations may be considered to minimize the size of $pc'(T_j)$. However, this may clearly slow down the processing of the operations within the $MDBS_i$ modules by unacceptable amounts. Note that in such a scheme, it is possible that $pc'(T_j) \subseteq pc(T_j)$.

Note that if we may assume that only a few transactions execute concurrently, and given that quick retrieval of the information stored is desirable from the viewpoint of the expedient dispatch of the operations, the use of hashing techniques suggests itself in the above schemes.

8.3 Scalability

The question of scalability in our scheme is answered in a simple manner. To add a new DBMS to an existing MDBS created by our scheme, an $MDBS_i$ module must be created for the newly added $DBMS_i$. By providing access to the networking facilities for data communications, the new DBMS can be linked to the existing MDBS in a straightforward manner. Obviously, the ease of scaling-up relies on the facility in developing the $MDBS_i$ interface itself.

9 Conclusions

The multidatabase transaction management scheme outlined in this paper has the desired properties of simplicity, economy, failure-resilience, and generality. We provide a simple means to integrate different database management systems each of which possibly uses a different concurrency control protocol. Our scheme resorts to the aborting of certain transactions to guarantee global atomic commitment. Two major advantages of our approach is that no restrictions are placed on the types of local and global transactions that can be run, and that no major changes need be made to the DBMSs being integrated. Only an interface module needs to be written.

Our scheme enforces the standard correctness criterion of serializability for the interleaved execution of the transactions. The transaction management exhibits resilience to failures that may be encountered in practical environments and is comparable to the resilience of homogeneous distributed database management systems. The failures that affect local activities are handled indirectly using the local recovery mechanisms that are already an integral part of each local database system. Failures in the distributed environment are handled as is done in distributed databases.

The performance of our system can only be gauged with further analysis and evaluation. However, we have identified the potential sources for performance bottlenecks, exhibited performance trade-offs, and described several possible intermediate approaches for enhancing efficiency.

References

[1] Bernstein, P.A., Hadzilacos, V., and Goodman, N. 1987. Concurrency Control and Recovery in Database Systems. Addison-Wes. Series in Comp. Sci.

[2] Breitbart, Y., Georgakopolous, D., Rusinkiewicz, M. and Silberschatz, A. 1990. Rigorous Scheduling in Multidatabase Transactions. Position paper at the Workshop on Multidatabases and Semantic Interoperability. Tulsa, Oklahoma.

[3] Breitbart, Y. and Silberschatz, A. 1988. Multidatabase Update Issues. *Proc. 1988 ACM-SIGMOD International Conference on Management of Data* (Jun).

[4] Breitbart, Y., Silberschatz, A. and Thompson, G.R. 1990. Reliable Transaction Management in a Multidatabase System. *Proc. 1990 ACM-SIGMOD International Conference on Management of Data* (May).

[5] Georgakopolous, D. and Rusinkiewicz, M. 1990. On Serializability of Multidatabase Transactions through Forced Local Conflicts. Technical Report. Dept of Computer Science, University of Houston.

[6] Gray, J. 1978. Notes on Database Operating Systems. *Operating Systems — An Advanced Course.* Springer-Verlag Lecture Notes in Computer Science. V.60.

[7] Gupta, A. *Ed.* 1989. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press.

[8] Korth, H.F., Levy, E., and Silberschatz, A. 1990. Optimistic Commit in Multidatabase Systems. Position paper at the Workshop on Multidatabases and Semantic Interoperability. Tulsa, Oklahoma.

[9] Korth, H.F. and Silberschatz, A. 1991. Database System Concepts, second edition. McGraw Hill.

[10] Lamport, L. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM. Vol.21, No.7* (July), 558-565.

[11] Muth, P., Klas, W. and Neuhold, E. 1990. How to Handle Global Transactions in Heterogeneous Database Systems. Position paper in the Workshop on Multidatabases and Semantic Interoperability. Tulsa, Oklahoma.

[12] Pu, C. 1987. Superdatabases: Transactions across Database Boundaries. IEEE *Data Engineering.*

[13] Report on the Workshop on Heterogeneous Database Systems. 1989. Northwestern University, Evanston (Dec).

[14] Report on the Workshop on Multidatabases and Semantic Interoperability. Tulsa, Oklahoma.

[15] Soparkar, N.R. and Silberschatz, A. 1990. Transactions in Distributed Multidatabases. Position paper at the Workshop on Multidatabases and Semantic Interoperability. Tulsa, Oklahoma.

[16] Wolski, A. and Veijalainen, J. 1990. 2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase. *Proc. International Conference on Databases, Parallel Architectures, and their Applications.* Miami Beach, Florida.