Figure 7: The MDBS System Structure
Figure 6: Correct Synchronization using 2PL and 2PC
Figure 5: Example of Incorrect Synchronization
Figure 3: Example for Observation 1

Figure 4: Example for Observation 2
Figure 2: DBMSs Prior to Integration
Figure 1: Example of Non-serializability


software changes in the constituent DBMSs.

The research literature available that addresses the problems of transaction management in MDBS environments exhibits these trade-offs. While most of the literature is concerned with failure-free systems, it is the research in fault-tolerant systems that has practical applications. In such cases, it is incumbent upon the MDBS designers to make suitable policy decisions regarding the choices available in terms of correctness properties vis-a-vis the preservation of local autonomy.

Our techniques to provide fault-tolerant transaction management with minimal changes made to the existing systems by infringing upon control autonomy. These techniques allow a large variety 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, they exhibit the desirable properties of deadlock avoidance and scalability. In order to do so, our scheme resorts to the aborting of certain transactions to guarantee global atomic commitment.

References


approach is to enforce the correctness criteria in a uniform manner in all the DBMS at the level of the MDBS so as to remove the mismatch in the correctness notions. This is possible if the control autonomy of the sites is violated. However, such approaches could suffer from gross inefficiencies since the concurrency control delays occur at both the MDBS as well as the DBMS levels.

Infringement of control autonomy also raises the issue of maintaining security since all the transactions need to pass through the MDBS. In this regard, since security considerations would essentially restrict global transactions from accessing certain data in a DBMS, an alternative scheme could be the extension of schemes such as [4] since security restrictions in any case require the separation of the data. A related problem in violating control autonomy is that the DBMS interfaces may be unknown or difficult to duplicate for the MDBS designer, thereby making the inclusion of an MDBS module difficult to implement.

6.2 Future DBMSs

If it desired that a DBMS is designed to permit its subsequent inclusion into an MDBS, certain “hooks” must be provided to facilitate the integration. The preceding discussion indicates the following appropriate ideas where we restrict attention to an MDBS that would provide the ACID properties.

Firstly, in order to achieve atomicity, the DBMS should provide for two features. One is a user interface that permits the submission of transactions that have their operations separable in the manner described in Section 3. The second is a prepared state so as to facilitate the provision of a GAC protocol.

Secondly, in order to provide consistency, a DBMS should provide serializable executions using a concurrency control protocol that allows synchronization across the sites as described in Section 3. This also permits the development of distributed MDBS managers. In this regard, since 2PL is a well-used and understood mechanism, it may be the protocol of choice.

The above suggestions concur with the generally accepted wisdom of using 2PL in conjunction with 2PC in distributed DBMSs.

7 Conclusions

We have seen that it is not possible, in general, to satisfy the traditional correctness criteria for transactions while preserving the local autonomy of the sites. Hence, certain trade-offs become necessary, and the suggested techniques in the literature perforce sacrifice one or more of the desired features of an ideal MDBS. In view of the difficulties encountered, two general directions may be suggested to provide an MDBS environment for a system of DBMSs. The first is to sacrifice, directly or indirectly, some of the ACID properties of correctness while preserving local autonomy. This approach makes the task of the system designer difficult since the alternative notions of correctness must be carefully designed. The second approach is to infringe on some aspects of local autonomy while maintaining the traditional notions of correctness. This infringement requires more effort on part of the implementer since a large part of the burden is in actually making the
$MDBS_i$, and this provides us the means to ensure serializability. However, such an approach incurs a penalty in efficiency since concurrency control delays are incurred in the $MDBS_i$ as well as the $DBMS_i$ levels. Hence, we propose that the existing protocols in $DBMS_i$ be used except for the following difference. Since global transactions that have non-conflicting subtransactions at a common site may suffer non-serializable executions (see Section 3), we propose that the technique of forcing conflicts between subtransactions at each site [7] be adopted. Hence, a synchronizing protocol (e.g., a GAC protocol) may be used to ensure serializability.

5.3 Failure-Resilience

The primary problem for the $MDBS_i$ module is to provide the prepared states for the subtransactions executing at its site. Recovery from site failures in our scheme can be handled in a manner similar to the one for handling internal aborts in the GAC protocol. Thus, upon recovery from a site failure, $MDBS_i$ simply resubmits repeatedly the set $ch(T_k)$ for a subtransaction $T_k$ that was in a prepared state at the time of the failure (and if the final decision for $T_k$ was to commit). Note that $DBMS_i$ will have forcibly aborted the set $pc(T_k)$ as part of its own recovery procedure.

The problem of distributed deadlocks that may occur in our scheme is resolved through the use of time-outs. The provision of a distributed MDBS structure allows handling distributed failures in a manner similar to typical distributed DBMS technology.

6 Practical Issues for Integration

This discussion is limited in scope to the transaction management issues for an MDBS. We separate the discussion into two parts. The first deals with issues for integrating existing DBMSs, and the second part suggests criteria for future DBMSs so as to facilitate their eventual integration.

6.1 Integrating Existing Systems

While several techniques have been reviewed for DBMSs that provide the ACID transaction management properties, the limiting fact is that actual DBMSs seldom provide all these properties. In fact, different DBMSs may provide different degrees of the ACID properties. For example, some may only provide cursor stability in place of serializability [9], while still others may not provide durability. In such situations, it is not difficult to construct examples of global transactions in an MDBS that may corrupt the integrity of the data in one DBMS due to a relaxed notion of correctness in another.

One way to handle this is to restrict the types of global transactions permissible (e.g., in a manner similar to [6] wherein the degree of dependencies between the sub-transactions is limited). The problem in such approaches is that it is often difficult to find the necessary restrictions easily (e.g., see [14]). The other
(henceforth, we refer to either as transactions) with their operations separable. No concurrency control information is required to be passed to \( MDBS_i \) explicitly to ensure correct executions. Each DBMS is assumed to manage local recovery from failures and deadlocks, and to provide the ACID properties. In particular, for each local SG, if there is an edge from a transaction \( T_i \) to a transaction \( T_j \), then \( T_i \) must have committed before \( T_j \) — a property that is usually provided in practical DBMSs.

### 5.2 Transaction Management

Our scheme emulates a GAC protocol that is used to ensure the ACID properties. Thus, when all the operations (except the final commit) for a subtransaction \( T_k \) have been successfully executed, the concerned \( MDBS_i \) engages in a GAC protocol as described in the main text. Prior to declaring a prepared state to the coordinator, \( MDBS_i \) saves \( ch(T_k) \), the set of changes made by \( T_k \), in stable storage, and until the completion of the protocol, no further operation from any other transaction at site \( S_i \) is submitted to \( DBMS_i \). When the final decision on the fate of \( T_k \) is received, \( MDBS_i \) accordingly submits to \( DBMS \), the commit or abort operation on behalf of \( T_k \). If the decision is to commit, and it is successfully executed, then \( ch(T_k) \) may be discarded, and the site continues to process the other transactions.

However, if the commit request for \( T_k \) is discarded by \( DBMS_i \), and thus, an internal abort threatens to compromise atomicity, then the following actions are performed by \( MDBS_i \). First, all active transactions that may directly follow \( T_k \) in the local SG are forcibly aborted to enforce the isolation of \( T_k \). An easy way to accomplish this is to abort all the other active transactions at site \( S_i \) (i.e., to emulate a failure). Since this is likely to be too drastic, we advocate that \( pc(T_k) \), the set of active transactions that potentially conflict with \( T_k \), be constructed as described below, and only those transactions be forcibly aborted. Following this, the set \( ch(T_k) \) is converted into a separate write-only transaction that is repeatedly resubmitted to \( DBMS_i \) until successfully committed. Thus, the aim is to emulate an execution where no internal abort of the subtransaction \( T_k \) occurs, and instead, some other transactions suffer internal aborts.

The construction of the set \( pc(T_k) \) depends on the particular concurrency control protocol in use at \( DBMS_i \), and the following is a method for the case of strict 2PL. The transactions that are active at \( DBMS_i \) from the start of the prepared state for \( T_k \) up to the completion of the GAC protocol, need to be considered. Only those active transactions that have outstanding operations that conflict with the operations of \( T_k \) are included in \( pc(T_k) \). Note that some transactions may be added to \( pc(T_k) \) even after the GAC protocol completes its execution on behalf of \( T_k \) since the \( MDBS_i \) may receive operations that conflict with \( T_k \) only after the completion of the protocol.

Notice that in our approach, unlike others, \( MDBS_i \) need not delay local transactions except during the GAC protocol, and that forced aborts of transactions occur only if a commit operation for a subtransaction fails, which happens infrequently in practice.

The infringement of control autonomy allows concurrency control to be managed at the level of the
handled. However, the approach works only in failure-free environments and is not able to handle internal aborts either.

4.5 Correctness in an Environment with Failures

Very little work is available on reliable MDBSs, and with the exception of [4], none of the techniques described above addresses the question of failures. In fact, the problem of internal aborts that exists even if failures do not occur per se, is not handled in most cases.

That failure-resilience may prove to be a difficult problem to solve for an MDBS environment has been known [10]. Subsequently, the problem is addressed in some detail in [4]. The approach taken there is restricted to DBMSs obeying strict 2PL at each site (in fact, to those with the rigorous version of 2PL), and as described above, the data access patterns of the transactions are constrained to accommodate re-trials in a simple manner. The restrictions permit the method to preserve control autonomy. A very similar approach to the problem is also suggested in [15].

The approach suggested in [11] deals with the loss of atomicity that may accompany failures of different kinds in a distributed MDBS. This approach to transaction management has been suggested elsewhere as well (e.g., see [15]), and its salient ideas have been described above in the context of compensating subtransactions. Unfortunately, this approach does not provide the traditional ACID properties of correctness.

Finally, the discussion in Section 5 provides a solution that infringes on control autonomy. As one may expect from the discussions in Section 3, permitting unrestricted transaction types necessitates such infringements. Notice that unlike the other approaches, this scheme provides for a distributed MDBS which is important for failure-resilience so as to counter situations where the site with the MDBS software fails.

5 Trading Control Autonomy for Reliability

We describe the salient points of a new approach for transaction management in an MDBS environment that has reliability comparable to typical distributed DBMS technology. Our goal is to provide efficient mechanisms to guarantee the ACID properties for unrestricted types of transactions — which implies, as discussed in Section 3, that it is necessary to infringe on the control autonomy of the local DBMSs. However, we support full execution autonomy for each DBMS, make use of the existing mechanisms for concurrency control and recovery, and propose a simple technique that makes minimal assumptions regarding the DBMSs so as to ensure wide applicability. Several ideas that are suggested in the research literature are incorporated in our scheme, and we provide the justification for our particular approach.

5.1 System Structure

Our scheme assumes an MDBS structure consisting of \( n \) sites, \( S_1, S_2, \ldots, S_n \), interconnected by a communications network as shown in Figure 7. Each \( LTM_i \) is assumed to accept transactions or subtransactions
The approaches in [3, 4] rely on restricting the types of global transactions to ensure correctness. The approach in [3] disallows any pair of global transactions to have more than one site in common where data is accessed by both the transactions. Obviously, this places a severe constraint on the global transactions. The approach in [4] restricts the access patterns of the global and local transactions such that at each site, two different sets of data items must be maintained — global data and local data. Global data and local data may be unrestrictedly accessed by global transactions and local transactions, respectively. However, the access of data across different types needs to be very carefully managed. It is not difficult to see that this places severe restrictions on the types of the global transactions since pre-existing data essentially cannot be accessed by the global transactions.

An easy approach to managing serializable executions is suggested in [7]. The idea is to constrain each sub-transaction of a global transaction to obtain a “ticket” at the site where it executes. The tickets serve to serialize the sub-transactions executing at each site. This idea is adopted also in our approach as discussed further in Section 5. The difference in the approach outlined in our paper is that control autonomy is also infringed upon, but that becomes necessary to ensure failure-resilience as described in Section 3.

Now consider some approaches that do not ensure serializability, and instead, provide a different measure of consistency. In [6], the correctness criterion takes the projections of the entire schedule onto the global transactions, and onto the local transactions at each site, and hence, requires that each projection should represent serializable executions. While this does not provide serializability, by the restrictions placed on the transaction types (e.g., preventing dependencies between the sub-transactions of a global transaction), the resulting executions work correctly. In this scheme, the technique requires the serial submission of the global transactions so as to ensure a serializable projection. A similar idea that is more general is expressed in [14] wherein several conditions are formulated under which the restrictions on the transactions may be relaxed, and moreover, the global transaction management is made serializable as opposed to being serial. In doing so, however, the database integrity constraints have to be considered for correctness.

There exist several proposed schemes that exploit specific application-dependent properties of the DBMSs. These schemes do not need to preserve serializability since the applications do not require it. Since the approach in this paper is a study of techniques for more general environments, these are not discussed here. The interested reader will find these in the literature (e.g., see [10, 18]).

Trade-offs discussed in Section 3 should be apparent in the above discussion. In [16], it is not necessary to violate control autonomy — however, the assumption of availability of local serialization orders implicitly violates design autonomy. The techniques used in [3, 4] preserve control autonomy — however, the technique is applicable only to DBMSs using 2PL, and also, several stringent restrictions are placed on the types of transactions permissible. The work in [7] also preserves control autonomy, though it has the minor drawback that even non-conflicting transactions are not permitted to run in a fully concurrent manner. Furthermore, given certain properties of the schedules provided by the DBMS, any concurrency control protocol may be
environments, need to infringe on either the correctness properties, or on the local autonomy of the sites.

4.3 Trading Atomicity

To begin with, consider the problems when the DBMS interfaces prohibit the separation of the operations as discussed in Section 3. The loss of atomicity in such situations persists even if control autonomy is not preserved — assuming that grossly inefficient means are not of consequence. In such situations, to preserve a modicum of atomicity, it may be necessary to re-submit the aborted sub-transactions repeatedly (to commit the entire global transaction), or to execute compensating sub-transactions [11] for the committed sub-transactions (to abort the entire global transaction).

If control autonomy is infringed upon, it may be possible to ensure that the corrective sub-transaction, re-submitted or compensating, is executed directly after the untoward aborting or commitment, respectively, of the concerned sub-transaction. For example, by preventing the execution of other transactions or sub-transactions that may access the affected data before the corrective actions are taken. However, if control autonomy is preserved, there is always the possibility that local transactions may access data that is inappropriately affected by the untoward aborting or commitment of the sub-transaction in question. In such situations, the corrective sub-transactions need to be constructed with the semantics of the application programs, and perhaps also the database integrity constraints, accounted for appropriately. Thus, some of the advantages associated with the ACID properties as described in Section 2 are lost. The other alternative is to restrict the global transaction types as described in [4, 6] as discussed below.

In the above techniques, it should be noted that the global transactions cannot be of an unrestricted type since the non-atomic executions preclude this flexibility. In general, if there is no way to guarantee atomicity, or a GAC protocol is not executed for some reason, it becomes necessary to restrict the transaction types. Various techniques have been proposed to restrict inter-site dependencies, or to restrict how the global and local transactions access data, or both. Although restricting local transaction types violates certain notions of autonomy, in certain applications, such restrictions may be inherent.

4.4 Consistency in Failure-Free Environments

Consider the approaches suggested that ensure serializability. The basic idea is to ensure that the serialization orders at the various sites are compatible. In the approach of [16], the idea is to obtain explicit serialization orders from the underlying DBMSs. Unfortunately, this would necessitate substantial changes made to the DBMSs since they do not usually provide these orders — and without the difficult changes being effected in the software, it is not easy to realize this technique. However, if the underlying DBMSs use certain concurrency control mechanisms as described above, the serialization orders become available indirectly. Unfortunately, there is no obvious way in which to extend the idea to environments where internal aborts or failures may occur.
4.2 Serializability Using Synchronization

From the above discussion, note that similar mechanisms may be used to ensure serializability if it is possible to identify intervals of the sort mentioned above that serve as indicators to the local serialization order. In fact, the different sites could use different mechanisms and yet be synchronized by a protocol as described above. For example, as suggested in the literature, if it is known that the time-stamp in a time-stamp order concurrency control scheme is provided by a DBMS upon the receipt of the first operation of a transaction, a similar serialization interval can be created. A synchronization protocol can be initiated just after the first operation is submitted, and the interval between the execution of the first operation up to the time that the message indicating the end of the synchronization protocol is received becomes the required interval. The intervals will not overlap given that the first operation of another sub-transaction is not submitted prior to the completion of the synchronization interval.

An important case for identifying such intervals is the case of analogous executions where the order of commitment of transactions at a site is also the serialization order. Since atomicity requirements necessitate a GAC at the end of the execution of all the operations of a sub-transaction, the interval between the execution of the final operation of a sub-transaction, up to the commitment of that sub-transaction may be used as the serialization interval. These intervals will not overlap, as required, if each site participates in at most one GAC protocol at a time. Interestingly, in the case that such intervals cannot be identified easily, other properties of a local schedule that relate to aspects of recovery can be used in conjunction with explicit local synchronization mechanisms to create such intervals. An example of this is provided in the approach outlined in Section which is adapted from [7]. The idea is to use the features of strict or cascadeless executions, and to impose that each sub-transaction at a site must access a special data item unique to that site. The effect of these accesses is to force conflicts between the subtransactions, and to thereby create analogous executions which can then be synchronized as described above. Since most DBMSs provide strict or cascadeless executions, the idea is practicable.

It should also be noted that several systems may be better designed by providing servers for each global transaction that executes at a site. Since the servers are not expected to interact with one another, care must be taken with regard to the manner in which they submit the operations of the local subtransactions. The properties such as cascadelessness and recoverability may be exploited in conjunction with the technique of enforced conflicts described above to find the situations in which servers may be used as opposed to a single server for all the subtransactions executing at a site. While we do not elaborate further on this topic, it is not difficult to see that the more restrictive properties, such as strictness, are more amenable to providing the facility for designing individual servers for each subtransaction.

In light of the observations made in Section 3, it is clear that in general it is not possible to meet the ACID properties of correctness together with preserving control and execution autonomy. Thus, as we see next, the techniques suggested in the literature for achieving reliable transaction management for MDBS
4.1 Using 2PL and 2PC

The common approach to ensure serializability in distributed DBMSs which has also been suggested for MDBSs [1, 5, 18], is the combination of strict 2PL at each site along with the 2PC protocol. Let us consider why care must be exercised to ensure correctness in using this technique in the latter case, while it works easily in the former.

In a DBMS, consider the locked interval of a transaction that obeys 2PL to be defined as the interval between the procurement of the last lock and the first release of a lock. It is not difficult to show that if the SG of a history has a path between two transactions $T_1$ and $T_2$, then the their locked intervals cannot overlap, and moreover, the serialization order of the transactions is the same as the order in which the locked intervals occur in the history [1]. Thus, in an MDBS that has DBMSs that each use the 2PL protocol, ensuring that all the serialization orders are compatible is equivalent to showing that all the locked interval orders are compatible.

Consider the situation depicted in Figure 5 that concerns an MDBS. If it happens to be the case that the local DBMS realizes that all the operations corresponding to $T_{pi}$ have been executed, it may begin releasing the locks for it in accordance with 2PL obeyed locally. That is, the release of the locks could begin before the 2PC protocol is complete. This may happen because the local transaction manager is not under the control of the MDBS. This makes the lock interval orders, and hence, also the serialization orders to be incompatible for global serializability. Notice that this could occur even though each site may have executions in accordance with strict 2PL with the release of the read locks prior to the commit operation [1]. In this example, although a 2PC protocol may ensure atomicity, it may fail to ensure serializability due to incorrect synchronization.

On the other hand, as depicted in Figure 6, if the locks are not released until after the synchronization is achieved using a 2PC protocol, the above problem will not occur. This can be seen by following the cycle of events (1 2 3 8 1) as shown that cannot occur because the events in a distributed history of events must follow a partial order [13]. The same reasoning can be extended to show the impossibility of situations for a non-serializable execution involving more than two global transactions and several sites. Notice that the concurrent executions of the local sub-transactions have not been of concern here.

The reason that the above problem does not occur in a distributed DBMS is because the locks are not released prior to the completion of a GAC. The same effect can be achieved in an MDBS by ensuring that the local transaction managers do not have sufficient information to realize that all the operations of a particular sub-transaction have been executed until after the synchronization is complete. Thus, if a local transaction manager gets such information from the submission of a commit operation, or from an end-of-transaction operation, that operation should be submitted to the DBMS only after the synchronization is complete. Notice that we make the implicit assumption here that the final operation can be submitted to the DBMS separate from the other operations.
be circumvented at the cost of losing some degree of concurrency [7]. This is described in more detail in Sections 4 and 5.

Arising from the above discussions, we see that unrestricted types of transactions permissible in the system, and preservation of control autonomy, together imply that failure to enforce global serialization explicitly among the sub-transactions may result in non-serializable executions (Observation 2).

It is important to consider the forms in which the interface of a DBMS will accept the sub-transactions from an MDBS. For example, if the interface requires that an entire sub-transaction be submitted without separating it into its operations followed by the submission of the commit (or abort), several problems may occur:

- Atomicity of the global transactions may be compromised since there may be no guarantee provided by the underlying DBMS that a sub-transaction will be successfully committed.

- Even if atomicity is preserved, serializability may not be ensured since there is no way in which a synchronization of the sub-transactions may be achieved in the manner described in Section 4 — except, under certain circumstances, by the inefficient method of not interleaving the executions of sub-transactions at any site.

- The global transaction loses its control over the ability to abort the transaction as a whole. Hence, if there are conditions under which the transaction is not supposed to execute successfully (e.g., if the execution of a sub-transaction depends on the values read by another), these cannot be controlled. In fact, in such cases, besides the separation of the operations and the final commit or abort, it is not difficult to see that there needs to be separation among the other operations as well.

The above maladies arise from the inability of the MDBS to govern the decision to commit or abort the global transaction as a whole. Notice also that if it is permitted to infringe upon the control autonomy of the local sites, some of these problems may be alleviated at the cost of decreased efficiency. For example, it is possible to envisage schemes where the MDBS prevents any interleaved executions, and thus ensures atomicity of global transactions simply by re-trying or undoing the sub-transactions of each global transaction as necessary (see Section 4).

Henceforth, unless otherwise specified, we shall assume that the separation of the nature mentioned above is available.

4 Relevant Techniques

We now review the techniques suggested in the literature in the context of the above discussions. It should be noted that the question of failures in an MDBS has been examined only of late. Hence, most of the available results apply only to failure-free systems.
that all the operations of sub-transaction $T$, 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.

Notice that an internal abort may also occur due to concurrency control reasons. For example, in an optimistic concurrency control scheme, the fate of a particular transaction may not be known until after the commit operation for it is submitted. To complicate matters further, notice that if failures are possible in the system, essentially the same problem occurs. The uncommitted sub-transaction $T$, for which a global decision to commit has been made, may be aborted by the recovery management schemes after a failure occurs. Hence, before the changes made by the sub-transaction are retrieved from stable storage to be retried by the MDBS, new local transactions over which the MDBS has no control may be executed as depicted in Figure 3. Thus we find that preserving execution and control autonomy, together with unrestricted types of transactions permissible in the system imply that a prepared state cannot be guaranteed (Observation 1).

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 for very special cases [16]. And often, this may involve substantial changes made to the underlying mechanisms. For example, Figure 1 provided an instance of the potential for non-serializable executions among global transactions despite the enforcement of local serializability.

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 4 for site $S_j$ where we again assume that control autonomy is preserved. Let $T_{pi}$ and $T_{qi}$ be two non-conflicting sub-transactions arising from global sub-transactions $G_p$ and $G_q$, and assume $T_{pi}$ commits first. The figure shows that this commit order may differ from the serialization order due to the presence of a local transaction $L_i$. As in the above example, a global serialization order may be impossible if the serialization order at site $S_j$ differs with respect to that at site $S_i$. This illustrates the subtlety 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 regarding which the MDBS is not informed or not in a position to control. This is important if the GAC is used for synchronization to ensure serializability, and hence, this point needs to be carefully considered. Also, as pointed out in [2], ensuring that the GAC of the transactions provides serializable executions is easy for a restricted class of rigorous schedules which are more restrictive than strict executions, and are usually not provided by typical centralized DBMSs. However, by adding explicit serialization mechanisms between sub-transactions executing at each site, this problem may
has a database management system, $DBMS_i$, consisting of a local database, $LDB_i$, and a local transaction manager, $LTM_i$. Since there could not have been transactions accessing data at more than one site prior to the integration, we may assume that the data is not replicated. The data items are regarded to conform to a common data model, and we shall also assume that the data manipulation methods are homogeneous.

The MDBS is designed to be 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:

- **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 sub-transactions to some or all of the local DBMSs.

We shall assume that the transactions have no restrictions placed on them as regards their data access patterns except that local transactions may access only locally stored data.

It is assumed that each local DBMS, besides ensuring the ACID properties, also handles local deadlocks either by avoidance or detection. Each local DBMS is responsible for recovery from the aborts of the local transactions, and also the failure of the local site. In particular, if a local site fails, upon recovery, it is expected to first undo all uncommitted transactions that had been active at the time of the failure, and to ensure that the committed transactions have their changes recorded in the database prior to permitting further executions of transactions.

Let us now consider the question of ensuring the atomic execution of the global transactions using a GAC in the context of an MDBS. There are difficulties encountered in guaranteeing the durability of the changes made by a sub-transaction without actually committing it. This problem arises mainly because the underlying DBMS may exercise its execution autonomy by aborting sub-transactions at any time. As an illustration, the possibility of an *internal* abort may be considered which allows a DBMS to abort a transaction or sub-transaction at any time during its execution. In particular, this may occur even after a commit operation has been submitted by the transaction or sub-transaction in question (but before the DBMS responds with an indication that the operation was successfully executed) [8]. Even if all the changes to be effected by a sub-transaction are maintained by the MDBS in stable storage in order to allow reinstating the changes by repeated retrials (e.g., see [4, 19]), 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 sub-transaction in question to access states of the database not affected by that sub-transaction. This is exemplified next.

Consider the situation depicted in Figure 3 for a DBMS where control autonomy is preserved. Assume
In typical distributed DBMSs, this may be achieved in several ways. For example, the transactions may be coerced to access the data at the various sites in a compatible order (e.g., by assigning global time-stamp [1]). Or, the same effect may be achieved by synchronizing the sub-transactions so that the orders are compatible (e.g., the common technique is to use two-phase locking (2PL) in conjunction with the two-phase commit (2PC) protocol [1]). The latter technique provides a way to integrate different concurrency control mechanisms as explained in Section 4. As described next, since a global atomic commitment (GAC) protocol (e.g., see [1]) is necessary in practice, it can be conveniently used to effect the desired synchronization.

Since it is not guaranteed that all the sub-transactions of a particular transaction will commit successfully, distributed DBMSs execute a GAC protocol to guarantee the atomicity of a transaction at the end of the executions of all its sub-transactions. Furthermore, to ensure the durability of the transaction, the changes to be effected by each sub-transaction need to be first stored in a stable manner prior to completion of a GAC protocol. Such a state for any sub-transaction is known as a prepared state to indicate the preparedness of the site in question to install all changes in spite of failures.

It is important to note that any atomic commitment protocol also plays a part in maintaining serializable executions if it is used to synchronize sub-transactions as mentioned above. When a transaction that accesses data at several sites reaches the stage of committing the actions, it is expected to have seen the database state as affected by a certain set of transactions (those that it reads from [1]), and if committed, it affects the database state permanently for transactions that are serialized after it. In particular, the synchronization means that if the decision is taken to commit the transaction, in spite of failures, it must be ensured that the transactions serialized after it do not see a database state unaffected by the transaction in question.

As seen below, the above points emerge as important considerations for ensuring, likewise, the requirements of atomicity and serializability for concurrently executing transactions in an MDBS environment especially given that it is desirable to preserve local autonomy.

3 Transaction Management in an MDBS

The fact that DBMSs are designed to function autonomously is the main cause for the problems encountered in the design of a reliable and correct transaction management scheme in an MDBS environment. The first step in designing an MDBS scheme that ensures the ACID properties for all the transactions is to make the assumption that the DBMSs also, in fact, provide these properties.\(^3\) In spite of doing so, as we exhibit below, it is not possible to provide the desired properties while preserving local autonomy.

We examine the problem of managing MDBS transactions with the traditional criteria for correctness as described in Section 2. To restrict attention to transaction management issues, we shall henceforth regard the original system to be as shown in Figure 2. The system consists of \(n\) sites, \(S_1, S_2, \ldots, S_n\). Each site \(S_i\)

\(^3\)Many actual systems do not conform to the traditional notions of correctness for application-specific or efficiency reasons. Thus, integrating such systems is more difficult, and this issue is discussed in more detail in Section 6.
guarantee that the actions being performed by the transaction are correct until such time that the commit operation is submitted to the system. However, once the commit operation is processed for a transaction by the system, it is guaranteed that other transactions following in the serialization order will observe database states that reflect the execution of that transaction.

The above notion has the following related implications. Firstly, ensuring that unexpected executions do not occur in case a transaction aborts means that a transaction must always maintain its isolation properties. This is usually achieved by further restricting the set of serializable executions increasingly as in being recoverable, cascadeless, or strict [1]. The reasons for these restrictions vary from ensuring isolation alone to the added practical matter of making the techniques to achieve isolation facile and efficient.

Secondly, system failures may result in the loss of data stored in volatile memory. Hence, it is necessary that the processing of the commit operation must include the saving of the effects of the transaction in question onto a stable storage medium. This is to ensure that the changes may be installed in the database itself in case a system failure occurs. These actions are necessary to ensure the property of durability of transaction executions.

The four properties sketched above are regarded as the defining paradigms for the traditional correctness of transaction executions.

2.3 Distributed DBMSs

Preserving the ACID properties in a distributed DBMS requires techniques in addition to those used in centralized environments. A transaction that accesses data at several sites has a designated coordinator site that is usually the site initiating the transaction. This site maintains the book-keeping information for the transaction and coordinates the actions of the constituent sub-transactions. At the end of the execution of each sub-transaction, the coordinator may choose to commit or abort the entire transaction by doing the same for each sub-transaction.

Now let us consider the correctness criteria in distributed DBMSs. Firstly, note that ensuring the serializability of the transaction executions is equivalent to ensuring that the serialization orders of all the sites are compatible in the following sense. Consider an SG for the entire distributed system (e.g., see [5]). It is constructed by a union of the local SGs which has one node each for committed local and global transactions. The edges between the local transactions are the same as the ones in the local SGs. However, any edges to and from a sub-transaction in the local SGs is added to and from, respectively, the node for the corresponding global transaction in the new SG. Hence, if there are no cycles in the new SG, the serialization orders of the different sites are said to be compatible. Note that if it can be ensured that each local SG is acyclic, then the only cycles that can be present in the new SG must include nodes from several local SGs, and an example is shown in Figure 1. The depiction shows three local SGs, and the new SG will have a cycle consisting of $G_p$, $G_q$, and $G_r$. 
properties. These properties are expected to hold in both centralized and distributed environments, and we examine them briefly. First, we review correctness in the context of a failure-free system. This is followed by the considerations for recovery from failures. Finally, we review these notions in the specific context of distributed DBMSs since an MDBS is most closely related to a distributed DBMS.

2.1 Consistency

Correctness for concurrent transaction management is defined by conflict serializability [1]. This is defined for a history of actions of the transactions that are committed. The acyclicity of the serializability graph (SG) that is defined for such a history is equivalent to the notion of a conflict serializable history [1] which is the defining measure for consistency of the executions.

Several considerations have prompted the adoption of this highly successful paradigm for correctness. For example, it can be easily implemented by efficient mechanisms. Furthermore, since this paradigm is based on the consistency of a transaction that executes alone in a system, the design of a transaction may concern itself with only the correctness of the constituent actions of that transaction — i.e., without regard to the other transactions that may actually execute concurrently with it.

Thus, no constraints are placed on the design of a transaction to restrict its data access patterns due to concurrency control reasons.\footnote{Though there may be restrictions due to other reasons such as security.} The paradigm of consistency permits the concurrency control mechanism to remain oblivious of both, the semantics of the programs that submit the transactions, as well as the implicit database integrity constraints.\footnote{Typically, integrity constraints are difficult to state explicitly for most practical systems [1, 12].}

We make this assumption in the discussions to follow so as to permit the techniques developed to be widely applicable, and especially so for the case of existing systems.

2.2 Presence of Failures

In the absence of failures, serializability usually suffices to guarantee the correctness of transaction executions. However, practical considerations preclude that possibility. Besides unexpected failures in the systems, concurrency control mechanisms may themselves be implemented in a manner that not all transactions execute all their constituent operations — i.e., the mechanisms may require to abort certain transactions. This brings-up the question of the atomicity of transaction executions wherein a transaction is expected to either execute fully, or not at all. This effectively means that the transaction management scheme must ensure that despite the incomplete execution of a transaction, the other transactions either see all the changes made it, or none at all. Thus, if a transaction is able to execute only partially, and hence has to abort, then the changes to data items that are made by it must be undone prior to allowing other transactions any access to those data items. This has the implication that until the commitment of a transaction, the actions of a transaction are not guaranteed by the underlying system. Similarly, if the system is such that it permits the user to submit a commit operation separate from the other actions, the user program does not make a
if necessary, the execution of a transaction or sub-transaction operation, or even abort an execution. 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, 19]). Local transactions in such situations continue to be submitted to the DBMS directly.

The work in managing transactions for MDBS environments that are failure-free has been relatively well-researched, and many techniques have been developed. These vary in the degree to which they violate local autonomy, and also, in the degree of flexibility that they provide to the users (e.g., see [10]). The basic problem addressed in the research has been the management of the heterogeneous concurrency control mechanisms that are used by the underlying DBMSs, and the aspect examined is one of providing the means to coordinate and synchronize the distributed executions.

However, in an environment where failures may occur (which is the norm rather than the exception), little research has been done previously. Some initial results were reported in [4], but overall, the area is relatively ill-understood. In this paper, we show that the issues of autonomy as described above are important considerations, and that they force the designer of an MDBS to trade-off certain desirable properties in order to achieve reliability for transaction management. In this light, we compare and contrast several available techniques in the research literature. We also describe our approach to solving the problem, and this approach infringes on certain aspects of autonomy.

The remainder of this paper is organized as follows. Section 2 describes the traditional correctness criteria for database transactions, and the associated implications these have for the purposes of implementation. Section 3 examines the problems of transaction management for an MDBS, and an analysis is presented to show why the desirable properties of maintaining correct executions along with the preservation of local autonomy cannot be simultaneously satisfied. Section 4 examines some of the approaches suggested in the literature in the context of the discussions in the preceding sections. In Section 5, we describe our technique to address the issues of correct transaction management for an MDBS. The issues raised in the integration of practical existing and future DBMSs is discussed in Section 6. Finally, Section 7 constitutes the conclusions.

2 Correctness Criteria

In this section, we describe the traditional correctness criteria for transactions that execute concurrently in a DBMS environment. We expect that the readers have a basic understanding of the ideas at a level comparable to the development in texts such as [12]. The intent here is to highlight some aspects of the theory that have a bearing on the design of an MDBS transaction management scheme.

Database transactions executing correctly in the conventional sense are characterized by their atomicity, consistency, isolation, and durability properties (e.g., see [1, 5]) — commonly referred to as the ACID
• It is desirable to allow users at a particular site to access the MDBS facilities using the same interface as is used to access the local DBMS — i.e., to make the use of the MDBS transparent to the users. This may involve the knowledge of the user interface implementation on the part of the MDBS designer since the interface may have to be duplicated to provide such a facility. Since such information regarding a user interface may not be available, this may not be an easy task to accomplish.

• While the first steps in designing an MDBS involve achieving the integration in any manner, care must be taken to avoid grossly inefficient designs even if they provide a correct integration. Thus, the problems of query optimization, access strategies, etc., should be borne in mind for a suitable design. These problems are different as compared to a distributed DBMS environment since the techniques used at each site may not be available to the MDBS designer.

• There are several other concerns that affect the design of an MDBS. For example, there is a need to make the system scalable so that, as the system evolves in time, it is possible to add new DBMSs to the system easily — without requiring a major reorganization. As another example, the problem of heterogeneous protocols used for inter-site communications suggest that the help of other research in the areas of network communications and distributed operating systems need to be utilized in order to achieve an integrated system.

In this paper, we restrict our attention to transaction management issues in the design of an MDBS. Since an MDBS is a distributed system, at first sight it may appear that we could use typical distributed DBMS techniques to manage the transactions (e.g., see [5]). However, that is a simplistic assumption since the requirement of having underlying DBMSs that function autonomously is not a concern in distributed DBMSs. That is, the distributed transactions that arise as a consequence of the integration in an MDBS require that their sub-transactions execute under the control of the underlying DBMSs that are un-coordinated among themselves. Moreover, other transactions may also be present that execute autonomously at each DBMS. It is the task of the MDBS to provide the necessary coordination between the sites so as to manage all the transactions. These issues are examined here in detail with the emphasis on concurrency control and fault tolerance.

The autonomous execution of the underlying DBMSs essentially captures the idea that they must continue to function as they did prior to the integration. This refers to the question of preserving local autonomy — an important design goal for an MDBS. Of the several connotations given to local autonomy (e.g., see [10]), we restrict attention here to those that are closely related to transaction management issues. Two important facets of transaction management considerations refer to the degrees to which the DBMSs and the MDBS have control over the transaction executions. The execution autonomy of a DBMS refers to the degree of control that a local transaction manager has over the transactions or sub-transactions executing at that site. Execution autonomy is preserved if the local transaction manager of a site can either delay, indefinitely
1 Introduction

A multidatabase system (MDBS) is a distributed database system in which the sites are not required to run the same underlying database management system (DBMS). The database systems that run locally at each site may be existing commercial DBMSs with large collections of applications already developed. The goal of an MDBS is the integration of this heterogeneous collection of DBMSs into a distributed database, and at the same time preserving the economic investment in the DBMSs and the related applications. This necessitates an MDBS design that requires at most minimal changes to be made to the underlying systems and allows the sites a substantial degree of control over the local data and locally executing transactions. Thus, an MDBS is expected to provide the additional functionality of access to several DBMSs by providing a software package on top of the existing systems. This is not an easy task, but its pressing practical importance (e.g., see [10]) has attracted the attention of the research community.

As may be expected, most of the problems arise because the existing database systems were not designed with a view that they will be eventually integrated into an MDBS. There are several important aspects that need to be addressed in the design of an MDBS, such as data translation for syntactic and semantic homogeneity, user interfaces, security, and transaction management issues (e.g., see [10, 17, 18]). Below, we briefly outline some of these issues.

- Differences in data manipulation languages and data models pose the problems of conversion of the queries expressed in one language to the other as well as the conversion of the resulting data from one form to the other. The ensuing research into these issues (e.g., see [17, 18]) indicate that these conversions are likely to be computationally expensive, and hence, concerns of efficiency dictate that transactions accessing data at several sites need to be effected via sub-transactions at each site that use some pre-defined set of data manipulation routines [18]. Thus, the dynamic information exchange between the different DBMSs should be kept to a minimum.

- Each DBMS may use a different concurrency control mechanism for the transactions executing at that site. Therefore, for a transaction that accesses data at several sites, each sub-transaction may be controlled by a different concurrency control mechanism. Moreover, back-up and recovery mechanisms that are an integral part of any practical system, may also be different. Providing an efficient, integrated concurrency control and recovery mechanism for an MDBS is the difficult problem that is addressed in this paper.

- Maintaining the security and the integrity of the data at the different DBMSs may have been achieved in different ways. Since the DBMS designer may not be privy to the techniques used, creating transactions that access data at several sites while continuing to maintain the accepted levels of security and integrity at each is another issue that needs research. The problem is compounded by the fact that individual DBMSs may impose additional restrictions dictated by the policy of the governing organizations.
Techniques for Failure-Resilient Transaction Management in Multidatabases *

Nandit Soparkar        Henry F. Korth        Abraham Silberschatz

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

Fax (512) 471-9028
{soapy,hfk,avi}@cs.utexas.edu
(512) 471-{9718,9716,9706}

Abstract
The ensuring of correct transaction executions in practical multidatabase systems is a difficult problem, since few changes are permissible in the constituent local database systems to accommodate the demands of the distributed environment. In this paper, we show that the preservation of two aspects of local autonomy — the degrees of control over the transactions by the sites (execution autonomy), and the multidatabase system (control autonomy) — has implications with regard to the correct executions of transactions in a multidatabase system that is susceptible to failures. The relationship between the local autonomy and achieving correct executions sheds some light on the reasons behind some of the difficulties that have been encountered in previous research.

We show that for transaction management in multidatabases, certain trade-offs must be made in order to achieve correct executions despite the possibility of failures. We examine several of the previously proposed approaches in this regard by comparing and contrasting them in the way they handle these difficulties by trading-off some of the desirable properties for multidatabases.

We also describe our new technique that infringes upon control autonomy in order to provide fault-tolerant transaction management with minimal changes made to the existing systems. This technique allows a large variety of concurrency control protocols to be handled, and local execution autonomy to be preserved. Our proposed protocol tolerates failures at a level comparable to traditional distributed database management systems. Moreover, it exhibits the desirable property of scalability.

Keywords
Multidatabase; Transaction management; Autonomy; Concurrency control; Fault-tolerance

*Work partially supported by NSF grants IRI-8805215, IRI-9003341, and by a grant from the IBM Corporation. This work was completed while the second author was at the University of Texas at Austin.
TECHNIQUES FOR FAILURE-RESILIENT
TRANSACTION MANAGEMENT
IN MULTIDATABASES

Nandit Soparkar
Henry F. Korth
Abraham Silberschatz

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
TR-91-10 April 1991
(Revised October 1991)