Using Tickets to Enforce the Serializability of Multidatabase Transactions

Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Amit Sheth

Abstract— **To enforce global serializability in a multidatabase environment the multidatabase transaction manager must take into account the indirect (transitive) conflicts between multidatabase transactions caused by local transactions. Such conflicts are difficult to resolve because the behavior or even the existence of local transactions is not known to the multidatabase system. To overcome these difficulties, we propose to incorporate additional data manipulation operations in the subtransactions of each multidatabase transaction. We show that if these operations create direct conflicts between subtransactions at each participating local database system, indirect conflicts can be resolved even if the multidatabase system is not aware of their existence. Based on this approach, we introduce optimistic and conservative multidatabase transaction management methods that require the local database systems to assure only local serializability. The proposed methods do not violate the autonomy of the local database systems and guarantee global serializability by preventing multidatabase transactions from being serialized in different ways at the participating database systems. Refinements of these methods are also proposed for multidatabase environments where the participating database systems allow schedules that are cascadeless or transactions have analogous execution and serialization orders. In particular, we show that forced local conflicts can be eliminated in rigorous local systems, local cascadelessness simplifies the design of a global scheduler and that local strictness offers no significant advantages over cascadelessness.**

Keywords— **multidatabase transactions, serializability, indirect conflicts, tickets, analogous execution and serialization orders, rigorous scheduling**

I. Introduction

Multidatabase system (MDBS) [1], [2] is a facility that supports global applications accessing data stored in multiple databases. It is assumed that the access to these databases is controlled by autonomous and possibly heterogeneous *Local Database Systems* (LDBSs). The MDBS architecture (Figure 1) allows *local transactions* and *global transactions* to coexist. Local transactions are submitted directly to a single LDBS, while the multidatabase (global) transactions are channeled through the MDBS interface. The objectives of a multidatabase transaction management are to avoid inconsistent retrievals and to preserve the global consistency in the presence of multidatabase updates. These objectives are more difficult to achieve in MDBSs than in homogeneous distributed database systems because, in addition to the problems caused by *data distribution* that all distributed database systems have to solve, transaction management mechanisms in MDBSs must also cope with *heterogeneity* and *autonomy* of the participating LDBSs.

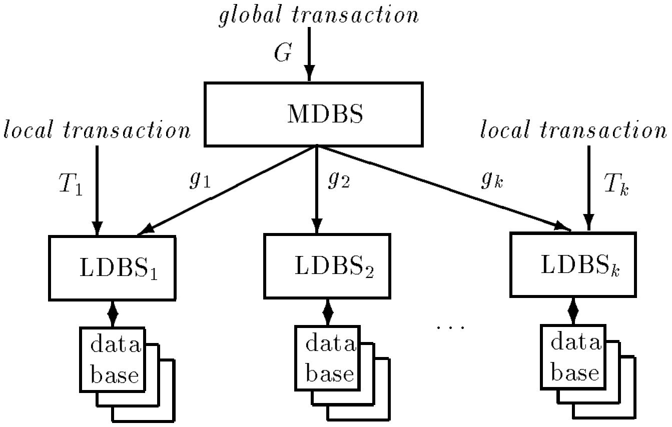


Fig. 1. Multidatabase system architecture.

The most important heterogeneities from the perspective of transaction management are dissimilarities in (i) the transaction management primitives and related error detection facilities available through the LDBS interfaces, and (ii) the concurrency control, commitment, and recovery schemes used by the LDBSs.

Local autonomy is the most fundamental assumption of the MDBS concept. Autonomy specifies the degree of independence and control the LDBSs have over their data. Since total autonomy means lack of cooperation and communication, and hence total isolation, some less extreme notions of LDBS autonomy have been proposed in the literature [3], [4], [2], [5]. Garcia-Molina and Kogan [4] explored the concept of node (site) autonomy in the context of a distributed system. Veijalainen [3] classifies the LDBS autonomy requirement into design autonomy, execution autonomy, and communication autonomy. In addition to these notions of autonomy, Sheth and Larson [2] identify additional LDBS properties that preserve association autonomy. In this paper, we consider that LDBS autonomy is not violated if the following two conditions are satisfied:

1. The LDBS is not modified in any way.
2. The local transactions submitted to the LDBS need not to be modified in any way (e.g., to take into account that the LDBS participates in a MDBS).

In a multidatabase environment the serializability of local schedules is, by itself, not sufficient to maintain multidatabase consistency. To ensure that global serializability

D. Georgakopoulos is with the Distributed Object Computing Department, GTE Laboratories, Incorporated, 40 Sylvan Road, MS-62, Waltham, MA 02254.

M. Rusinkiewicz is with the Department of Computer Science, University of Houston, Houston, TX 77204-3475.

A. Sheth is with Bellcore, 444 Hoes Lane, Piscataway, NJ 08854.

is not violated, local schedules must be validated by the MDBS. However, the local serialization orders are neither reported by the local database systems, nor can they be determined by controlling the submission of global subtransactions or observing their execution order. To determine the serialization order of the global transactions at each LDBS, the MDBS must deal not only with *direct conflicts* that may exist between the subtransactions of multidatabase transactions, but also with the *indirect conflicts* that may be caused by local transactions. Since the MDBS has no information about the existence and behavior of local transactions, determining if an execution of global and local transactions is globally serializable is difficult. An example illustrating this problem is presented in the next section.

Several solutions have been proposed in the literature to deal with this problem, however, most of them are not satisfactory. The main problem with the majority of the proposed solutions is that they do not provide a way of assuring that the operation execution order of global transactions, which can be controlled by the MDBS, is reflected in the local serialization order of the global transactions produced by the LDBSs. For example, it is possible that a global transaction G_i is executed and committed at some LDBS before another global transaction G_j, but their local serialization order is reversed. In this paper, we address this problem by introducing a technique that disallows such local schedules, and enables the MDBS to determine the serialization order of global transactions in each participating LDBS. Our method does not violate the local autonomy and is applicable to all LDBSs that ensure local serializability. Unlike other solutions that have been proposed in the literature, our technique can be applied to LDBSs that provide interfaces at the level of set-oriented queries and updates (e.g., SQL or QUEL).

Having established a method to determine the local serialization order of global transactions in LDBSs, we introduce optimistic and conservative methods that enforce global serializability. In addition, we propose efficient refinements of these methods for multidatabase environments where the participating database systems use cascadeless or rigorous schedulers [6], [7]. We show that multidatabase scheduling is simplified in multidatabase environments where all local systems are cascadeless. Further simplifications are possible if LDBSs use one of the many common schedulers that assure that transaction serialization orders are analogous to their commitment order. We show that in such multidatabase environments the local serialization order of global transactions can be determined by controlling their commitment order at the participating LDBSs. Although we address the problem of enforcing global serializability in the context of a multidatabase system, the solutions described in this paper can be applied to a Distributed Object Management System [8].

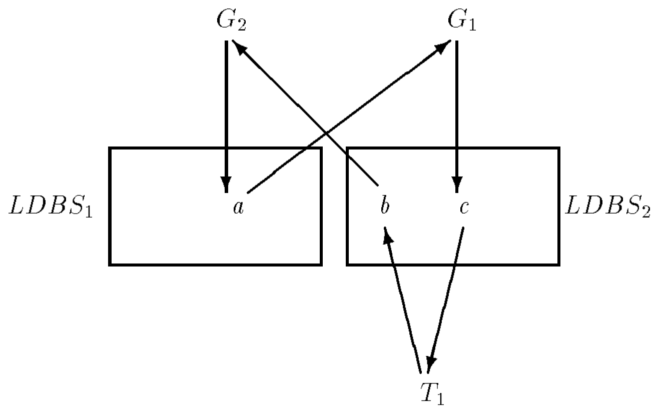
This paper is organized as follows. In Section II, we identify the difficulties in maintaining global serializability in MDBSs and review related work. The multidatabase model and our assumptions and requirements towards local database management systems are discussed in Section III. In Section IV, we introduce the concept of a ticket and propose the *Optimistic Ticket Method* (OTM) for multidatabase transaction management. To guarantee global serializability, OTM requires that the LDBSs ensure local serializability. In Section V, we introduce the *Conservative Ticket Method* (CTM) that also requires global transactions to take tickets but is free from global restarts. Variations of OTM and CTM that use simpler global schedulers but work only in multidatabase systems in which all local systems are cascadeless are presented in Section VI. In Section VII we introduce the concept of implicit tickets and propose the *Implicit Ticket Method* (ITM) which does not require subtransaction tickets but works only in multidatabase environments where the participating LDBSs are rigorous. Integrating the methods above in mixed multidatabase schedulers is discussed in Section VIII. Finally, in Section IX, we summarize our results.

II. Problems in maintaining global serializability and related work

Many algorithms that have been proposed for transaction management in distributed systems are not directly applicable in MDBSs because of the possibility of indirect conflicts caused by the local transactions. To illustrate this point consider Figure 2 which depicts the execution of two multidatabase transactions G_1 and G_2, and a local transaction T_1. If a transaction G_i reads a data item a, we draw an arc from a to G_i. An arc from G_i to a denotes that G_i writes a. In our example, the global transactions have subtransactions in both LDBSs. In LDBS$_1$, G_1 reads a and G_2 later writes it. Therefore, G_1 and G_2 directly conflict in LDBS$_1$ and the serialization order of the transactions is $G_1 \rightarrow G_2$. In LDBS$_2$, G_1 and G_2 access different data items: G_1 writes c and later G_2 reads b. Hence, there is no direct conflict between G_1 and G_2 in LDBS$_2$. However, since the local transaction T_1 writes b and and reads c, G_1 and G_2 conflict indirectly in LDBS$_2$. This indirect conflict is caused by the presence of the local transaction T_1. In this case, the serialization order of the transactions in LDBS$_2$ becomes $G_2 \rightarrow T_1 \rightarrow G_1$.

In a multidatabase environment, the MDBS has control over the execution of global transactions and the operations they issue. Therefore, the MDBS can detect direct conflicts involving global transactions, such as the conflict between G_1 and G_2 at LDBS$_1$ in Figure 2. However, the MDBS has no information about local transactions and the indirect conflicts they may cause. For example, since the MDBS has no information about the local transaction T_1, it cannot detect the indirect conflict between G_1 and G_2 at LDBS$_2$. Although both local schedules are serializable, the global schedule is non-serializable, i.e. there is no global order involving G_1, G_2 and T_1 that is compatible with both local schedules.

In the early work in this area, the problems caused by indirect conflicts were not fully recognized. In [9], Gligor and Popescu-Zeletin stated that a schedule of multidatabase transactions is correct if multidatabase transactions have



LDBS$_1$: $r_{G_1}(a)w_{G_2}(a), i.e., G_1 \rightarrow G_2$
LDBS$_2$: $r_{T_1}(c)w_{G_1}(c)r_{G_2}(b)w_{T_1}(b), i.e., G_2 \rightarrow T_1 \rightarrow G_1$

Fig. 2. Serial execution of multidatabase transactions may violate serializability.

the same relative serialization order at each LDBS they (directly) conflict. Breitbart and Silberschatz have shown [10] that the correctness criterion above is insufficient to guarantee global serializability in the presence of local transactions. They proved that the sufficient condition for global consistency requires multidatabase transactions to have the same relative serialization order at all sites they execute. The solutions to the problem of concurrency control in MDBSs proposed in the literature can be divided into several groups:

Observing the execution of the global transactions at each LDBS [11]. The execution order of global transactions does not determine their relative serialization order at each LDBS. For example, at LDBS$_2$ in Figure 2, the global transaction G_1 is executed before G_2, but G_2 precedes G_1 in the local serialization order there. To determine local conflicts between transactions, Logar and Sheth [12] proposed using the commands of the local operating system and DBMS to "snoop" on the LDBS. Such an approach may not always be possible without violating the autonomy of the LDBS.

Controlling the submission and execution order of global transactions. Alonso et al. proposed to use site locking in the *altruistic locking* protocol [13] to prevent undesirable conflicts between multidatabase transactions. Given a pair of multidatabase transactions G_1 and G_2, the simplest altruistic locking protocol allows the concurrent execution of G_1 and G_2 if they access different LDBSs. If there is a LDBS that both G_1 and G_2 need to access, G_2 cannot access it before G_1 has finished its execution there. Du et al. [14] have shown that global serializability may be violated even when multidatabase transactions are submitted serially, i.e., one after the completion of the other, to their corresponding LDBS. The scenario in Figure 2 illustrates the above problem. G_1 is submitted to both sites, executed completely and committed. Only then is G_2 submitted for execution; nevertheless the global consistency may be violated.

Limiting multidatabase membership to the LDBSs that use strict schedulers. By disallowing local executions that are serializable but not strict, this approach places additional restrictions on the execution of both global and local transactions at each participating LDBS. A solution in this category, called the 2PC Agent Method, was proposed in [15]. The 2PC Agent Method assumes that the participating LDBSs use two-phase locking (2PL) [16] schedulers and produce only strict [17] schedules. The basic idea in this method is that strict LDBSs will not permit local executions that violate global serializability. However, even local strictness is not sufficient. To illustrate this problem, consider the LDBSs in Figure 2 and the following local schedules:

LDBS$_1$:$r_{G_1}(a)commit_{G_1}w_{G_2}(a)commit_{G_2}$, $G_1 \rightarrow G_2$
LDBS$_2$:$r_{G_1}(b)r_{G_2}(b)w_{G_1}(b)commit_{G_1}commit_{G_2}$, $G_2 \rightarrow G_1$

The schedule at LDBS$_1$ is serial. In LDBS$_2$, G_1 and G_2 are both able to obtain read-locks and read b. Next, G_2 releases its read lock on b and does not acquire any more locks. G_1 is able to obtain a write lock and update b before G_2 commits. This execution is allowed by 2PL. Strictness in 2PL is satisfied if each transaction holds only its *write-locks* until its end. Therefore, both schedules above are strict and are allowed by 2PL. However, global serializability is violated.

Assume conflicts among global transactions whenever they execute at the same site. This idea has been used by Logar and Sheth [12] in the context of distributed deadlocks in MDBSs and by Breitbart et al. [18] for concurrency control in the Amoco Distributed Database System (ADDS). Both approaches are based on the notion of the *site graph*. In the ADDS method, when a global transaction issues a subtransaction to a LDBS, *undirected edges* are added to connect the nodes of the LDBSs that participate in the execution of the global transaction. If the addition of the edges for a global transaction does not create a cycle in the graph, multidatabase consistency is preserved and the global transaction is allowed to proceed. Otherwise, inconsistencies are possible and the global transaction is aborted.

The site graph method does not violate the local autonomy and correctly detects possible conflicts between multidatabase transactions. However, when used for concurrency control, it has significant drawbacks. First, the degree of concurrency allowed is rather low because multidatabase transactions cannot be executed at the same LDBS concurrently. Second, since site locking uses an undirected graph to represent conflicts, not all cycles in the graph correspond to globally non-serializable schedules. Third, and more importantly, the MDBS using site graphs has no way to determine when it is safe to remove the edges of a committed global transaction. The edge removal policy used in the Serialization Graph Testing algorithm [17] is not applicable in this case, since the site graph is undirected. To illustrate this problem consider the LDBSs in Figure 2 and the following local schedules:

LDBS$_1$:$r_{G_1}(a)commit_{G_1}w_{G_2}(a)$
LDBS$_2$:$r_{T_1}(c)w_{G_1}(c)commit_{G_1}\ r_{G_2}(b)$

Since G_1 and G_2 perform operations in both LDBSs the site graph that corresponds to the schedules above contains a cycle between G_1 and G_2. To resolve the cycle, the site graph method aborts G_2. Suppose that the edges corresponding to G_1 are removed from the site graph immediately following the commitment of G_1. If G_2 is restarted after the commitment of G_1, it will be allowed to commit, since there is no cycle in the site graph. Now suppose that after G_2 commits, a local transaction T_1 issues $w_{T_1}(b)$ and commits. The execution of these operations results in the schedules shown in Figure 2 that locally serializable, but globally non-serializable. Therefore, if the edges corresponding to a global transaction are removed from the site graph immediately following its commitment, global serializability may be violated.

The site graph method may work correctly if the removal of the edges corresponding to a committing transaction is delayed. However, concurrency will be sacrificed. In the scenario represented by Figure 2, the edge corresponding to G_1 can be removed after the commitment of the local transaction T_1. However the MDBS has no way of determining the time of commitment or even the existence of the local transaction T_1. This problem has been recognized in [6], [7].

Modifying the local database systems and/or applications. Pu [19] has shown that global serializability can be ensured if LDBSs present their local serialization orders to the MDBS. Since traditional DBMSs usually do not provide their serialization order, Pu suggests modifying the LDBSs to provide it. Pons and Vilarem [20] proposed modifying existing applications so that all transactions (including local ones) are channeled through multidatabase interfaces. Both methods mentioned here preserve multidatabase consistency, but at the expense of partially violating the local autonomy.

Rejecting serializability as the correctness criterion. The concept of *sagas* [21], [22] has been proposed to deal with long-lived transactions by relaxing transaction atomicity and isolation. *Quasi-serializability* [23] assumes that no value dependencies exist among databases so indirect conflicts can be ignored. *S-transactions* [24] and *flexible transactions* [25] use transaction semantics to allow non-serializable executions of global transactions. These solutions do not violate the LDBS autonomy and can be used whenever the correctness guarantees they offer are applicable. In this paper, we assume that the global schedules must be serializable.

III. The multidatabase system model

Global transactions consist of a transaction *begin* operation, a partially ordered collection of read and write operations, and a *commit* or *abort (rollback)* operation. In the following discussion, we refer to the collection of the read and write operations performed by a transaction T as the *database operations* of T. We use the term *transaction management operations* to refer to the non-database operation performed by T

The MDBS processes each global transaction G as follows. First, the MDBS decomposes G to subtransactions g_1, g_2, ... g_n. The decomposition of G is based on the location of the data objects G accesses. For example, if G accesses data objects on LDBS$_i$, the MDBS issues a subtransaction g_i to carry out the operations of G at LDBS$_i$. We assume that subtransactions generated by the MDBS satisfy the following requirements:

1. There is at most one subtransaction per LDBS for each global transaction.
2. Like global transactions, subtransactions consists of database operations and transaction management operations. All subtransaction operations can be executed locally by the LDBS. A subtransaction may perform a *prepare-to-commit* operation before issuing *Commit*, if the LDBS provides this operation in its interface.
3. Subtransactions have a *visible prepared-to-commit* state.

We say that a transaction enters its prepared-to-commit state [26] when it completes the execution of its database operations and leaves this state when it is committed or aborted. During this time, all updates reside in its private workspace and become permanent in the database when the transaction is committed. The prepared-to-commit state is *visible* if the application program (in this case the MDBS) can decide whether the transaction should commit or abort. To process G, the MDBS submits the subtransactions of G to their corresponding LDBSs. To ensure that the logically indivisible action to commit or abort G is consistently carried out in the participating sites, the MDBS uses the *two-phase commit* (2PC) [26] protocol. Since LDBSs may reside at remote sites, an MDBS *agent* process is associated with each LDBS to submit G's operations to the LDBS and handle the exchange and synchronization of all messages to and from the MDBS.

A. Local database management systems assumptions

We assume that a LDBS provides the following features without requiring any modification:

1. Permits only serializable and recoverable [17] schedules.
2. Ensures failure atomicity and durability of transactions. If a subtransaction fails or is aborted, the DBMS automatically restores the database to the state produced by the last (locally) committed transaction.
3. Supports the *begin*, *commit* and *abort* (*rollback*) transaction management operations. Each subtransaction can either issue a *commit* and install its updates in the database or issue an *abort* to roll back its effects.
4. Notifies the transaction programs of any action it takes unilaterally. In particular, it is assumed that a DBMS interface is provided to inform subtransaction programs when they are unilaterally aborted by the LDBS. For example, to resolve a deadlock, a DBMS may roll back one (e.g., the youngest) of the transactions involved and notify the killed transaction about the rollback (e.g., by setting a flag in the program

communication area).

These features are supported by the majority of commercial DBMSs, including[1] DB2, INGRES, ORACLE, and SYBASE. Furthermore, all the features described above comply with the SQL [27] and RDA [28] standards.

Most DBMSs use high level languages (e.g. SQL) to support set-oriented queries and updates. In our discussion we model global transactions, their subtransactions and local transactions as collections of read and write operations. We have chosen the read/write transaction model to simplify the discussion of problems in enforcing global serializability in a multidatabase environment, and we use this model to describe corresponding solutions. However, the use of the read/write model neither limits the generality of the solution proposed in this paper, nor makes it more difficult to apply them in a LDBS that supports interfaces at the level of set-oriented queries and updates. To illustrate this, we have included an Appendix that discusses implementation-related issues for LDBS using SQL interfaces.

B. *The prepared-to-commit state in a multidatabase environment*

Earlier in Section III, we listed the assumption that subtransactions have a visible prepared-to-commit state. Many database management systems, designed using the client-server architecture (e.g., SYBASE) provide a visible prepared-to-commit state and can directly participate in a multidatabase system. On the other hand, if the LDBS does not explicitly provide such a state, the MDBS can simulate it [29], [30].

To simulate the prepared-to-commit state of a subtransaction, the MDBS must determine whether all database operations issued by the subtransaction have been successfully completed. One way to accomplish this is to force a handshake after each operation, i.e., the MDBS must submit the operations of each subtransaction one at a time and wait for the completion of the previous database operation before submitting the next one. Alternatively, the RDA standard [28] allows asynchronous submission of several database operations and provides a mechanism to inquire about the status of each of them.

Consider the state of a subtransaction that has successfully finished all its operations but is neither committed nor aborted. To distinguish such a state from a prepared-to-commit state, we refer to it as the *simulated prepared-to-commit* state. The basic difference between the prepared-to-commit state and the simulated prepared-to-commit state is that a transaction in the simulated state has no firm assurance from the DBMS that it will not be unilaterally aborted. However, database management systems do not unilaterally abort any transaction after it has entered its simulated prepared-to-commit state.[2] Transactions in this state cannot be involved in deadlocks because they have successfully performed all their operations and have acquired all their locks. The same is true for LDBSs that use aborts and restarts to resolve conflicts. For example, timestamp ordering [17] aborts a transaction when it issues an operation that conflicts with some operation performed earlier by a younger transaction. Therefore, timestamp ordering schedulers never abort transactions after they have successfully issued all their operations and entered their simulated prepared-to-commit state. The behavior of optimistic concurrency control protocols [32] is similar. No transaction is ever aborted after it passes validation.

While DBMSs do not abort transactions in this state for concurrency control and recovery reasons, it is possible to argue that DBMSs must set timeouts to avoid having "idle" transactions holding resources forever. However, due to the difficulties in determining whether a subtransaction is "idle" and for how long, the only timeouts set by most DBMSs are on outstanding operations (e.g., in SYBASE and ORACLE). Therefore, when the last read or write operation of a subtransaction is completed, the MDBS can be certain that the subtransaction has entered a state which in practice is no different from the prepared-to-commit state required by 2PC. In the rest of this paper, we do not distinguish whether a visible prepared-to-commit state is simulated or is provided by local systems. Additional issues related to the problem of effectively providing a prepared-to-commit state are discussed in [33].

IV. The Optimistic Ticket Method (OTM)

In this section, we describe a method for multidatabase transaction management, called OTM, that does not violate LDBS autonomy and guarantees global serializability if the participating LDBSs ensure local serializability. The proposed method addresses two complementary issues:

1. How MDBS can obtain information about the relative serialization order of subtransactions of global transactions at each LDBS?
2. How MDBS can guarantee that the subtransactions of each multidatabase transaction have the same relative serialization order in all participating LDBSs?

In the following discussion, we do not consider site failures (commitment and recovery of multidatabase transactions are discussed, among others, in [34], [35], [30], [33]).

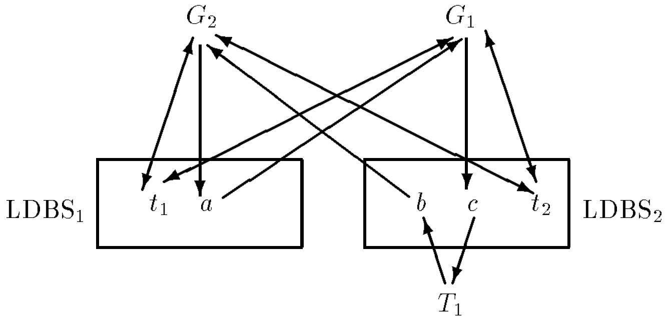
A. *Determining the local serialization order*

OTM uses *tickets* to determine the relative serialization order of the subtransactions of global transactions at each LDBS. A ticket is a (logical) timestamp whose value is

[1] Any mention of product or vendors in this paper is done for background information, or to provide an example of a technology for illustrative purposes and should not be construed as either a positive or negative commentary on that product or vendor. Neither inclusion of a product or a vendor in this paper nor omission of a product or a vendor should be interpreted as indicating a position or opinion of that product or vendor on the part of the authors or of Bellcore. Each reader is encouraged to make an independent determination of what products are in the marketplace and whether particular features meet their individual needs.

[2] Wound-Wait deadlock avoidance technique [31] may abort a transaction holding a lock because some other transaction requests the same lock. This is the only policy we are aware of that may abort a transaction in its simulated prepared-to-commit state Since its use is limited in commercial DBMSs, we do not consider it in this paper and assume that a transaction in the simulated prepared-to-commit state is not aborted by its LDBS.

stored as a regular data item in each LDBS. Each subtransaction of a global transaction is required to issue the *Take-A-Ticket* operation which consist of reading the value of the ticket (i.e., $r(ticket)$) and incrementing it (i.e., $w(ticket + 1)$) through regular data manipulation operations. The value of a ticket and all operations on tickets issued at each LDBS are subject to local concurrency control and other database constraints. Only a single ticket value per LDBS is needed. The Take-A-Ticket operation does not violate local autonomy because no modification of the local systems is required. Only the subtransactions of global transactions have to take tickets [3]; local transactions are not affected.



LDBS$_1$: $r_{G_1}(t_1)w_{G_1}(t_1+1)r_{G_1}(a)r_{G_2}(t_1)w_{G_2}(t_1+1)$ $w_{G_2}(a), i.e.,\ G_1 \rightarrow G_2$

LDBS$_2$: $r_{T_1}(c)r_{G_1}(t_2)w_{G_1}(t_2+1)w_{G_1}(c)r_{G_2}(t_2)$ $w_{G_2}(t_2+1)r_{G_2}(b)w_{T_1}(b), i.e.,$

$$\begin{tikzcd} G_2 \arrow[r] & T_1 \arrow[r] & G_1 \arrow[ll, bend left] \end{tikzcd}$$

Fig. 3. The effects of the Take-A-Ticket approach.

Figure 3 illustrates the effects of the Take-A-Ticket process on the example in Figure 2. The ticket data items at LDBS$_1$ and LDBS$_2$ are denoted by t_1 and t_2, respectively. In LDBS$_1$, the t_1 values obtained by the subtransactions of G_1 and G_2 reflect their relative serialization order. This schedule will be permitted by the local concurrency controller at LDBS$_1$. In LDBS$_2$ the local transaction T_1 causes an indirect conflict such that $G_2 \rightarrow T_1 \rightarrow G_1$. However, by requiring the subtransactions to take tickets we force an additional conflict $G_1 \rightarrow G_2$. This additional ticket conflict causes the execution at LDBS$_2$ to become locally non-serializable. Therefore, the local schedule:

$$r_{T_1}(c)r_{G_1}(t_2)w_{G_1}(t_2+1)w_{G_1}(c)\ r_{G_2}(t_2)\ w_{G_2}(t_2+1)\ r_{G_2}(b)\ w_{T_1}(b)$$

will be not allowed by the local concurrency control (i.e., the subtransaction of G_1 or the subtransaction of G_2 or T_1 will be blocked or aborted).

On the other hand, if the local schedule in LDBS$_2$ were for example:

$$r_{G_1}(t_2)w_{G_1}(t_2+1)w_{G_1}(c)\ r_{G_2}(t_2)\ w_{G_2}(t_2+1)\ r_{T_1}(c)\ r_{G_2}(b)\ w_{T_1}(b)$$

the tickets obtained by G_1 and G_2 would reflect their relative serialization order there and the local schedule would be permitted by the local concurrency control at LDBS$_2$. Although the transactions in our example take their tickets at the beginning of their execution, transactions may take their tickets at any time during their lifetime without affecting the correctness of the Take-A-Ticket approach. Theorem 1 formally proves that the tickets obtained by the subtransactions at each LDBS are guaranteed to reflect their relative serialization order.

Theorem 1: The tickets obtained by the subtransactions of multidatabase transactions determine their relative serialization order.

Proof: Let g_i and g_j be the subtransactions of global transactions G_i and G_j, respectively, at some LDBS. Without loss of generality we can assume that g_i takes its ticket before g_j, i.e., $r_{g_i}(ticket)$ precedes $r_{g_j}(ticket)$ in the local execution order. Since a subtransaction takes its ticket first and then increments the ticket value, only the following execution orders are possible:

E_1:$r_{g_i}(ticket)r_{g_j}(ticket)w_{g_i}(ticket+1)w_{g_j}(ticket+1)$
E_2:$r_{g_i}(ticket)r_{g_j}(ticket)w_{g_j}(ticket+1)w_{g_i}(ticket+1)$
E_3:$r_{g_i}(ticket)w_{g_i}(ticket+1)r_{g_j}(ticket)w_{g_j}(ticket+1)$

However, among these executions only E_3 is serializable and can be allowed by the LDBS concurrency control. Therefore, g_i increments the ticket value before g_j reads it and g_j obtains a larger ticket than g_i.

To show now that g_i can only be serialized before g_j, it is sufficient to point out that the operations to take and increment the ticket issued first by g_i and then by g_j create a direct conflict $g_i \rightarrow g_j$. This direct conflict forces g_i and g_j to be serialized according to the order in which they take their tickets. More specifically, if there is another direct conflict between g_i and g_j, such that $g_i \rightarrow g_j$ (Figure 4 (a)) or indirect conflict caused by local transactions, such that $g_i \rightarrow T_1 \rightarrow T_2 \ldots \rightarrow T_n \rightarrow g_j (n \geq 1)$ (Figure 4 (c)), the resulting schedule is serializable and both g_i and g_j are allowed to commit. In this case, g_i is serialized before g_j and this is reflected by the order of their tickets. However, if there is a direct conflict $g_j \rightarrow g_i$ (Figure 4 (b)), or an indirect conflict $g_j \rightarrow T_1 \rightarrow T_2 \ldots \rightarrow T_n \rightarrow g_i (n \geq 1)$ (Figure 4 (d)), the ticket conflict $g_i \rightarrow g_j$ creates a cycle in the local serialization graph. Hence, this execution becomes non-serializable and is not allowed by the LDBS concurrency control. Therefore, indirect conflicts can be resolved through the use of tickets by the local concurrency control even if the MDBS cannot detect their existence. □

An implementation of tickets and the Take-A-Ticket operation in LDBSs using SQL is described in Appendix I.

B. Enforcing global serializability

To maintain global consistency, OTM must ensure that the subtransactions of each global transaction have the same relative serialization order in their corresponding LDBSs [10]. Since, the relative serialization order of the subtransactions at each LDBS is reflected in the values of

[3] This may create a "hot spot" in the LDBSs. However, since only subtransactions of multidatabase transactions and not local LDBS transactions have to compete for tickets, we do not consider this to be a major problem affecting the performance of our method.

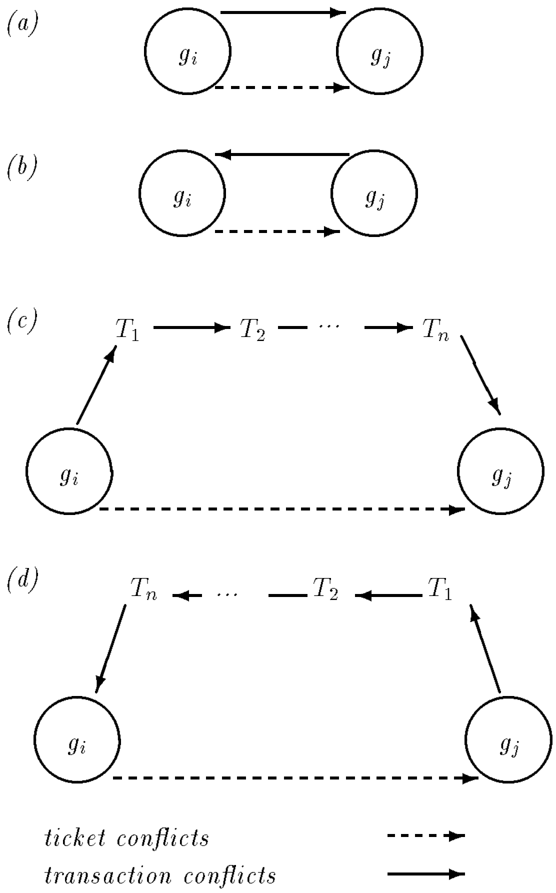


Fig. 4. The effects of ticket conflicts in OTM.

their tickets, the basic idea in OTM is to allow the subtransactions of each global transaction to proceed but commit them only if their ticket values have the same relative order in all participating LDBSs. This requires that all subtransactions of global transactions have a visible prepared-to-commit state.

OTM processes a multidatabase transaction G as follows. Initially, it sets a timeout for G and submits its subtransactions to their corresponding LDBSs. All subtransactions are allowed to interleave under the control of the LDBSs until they enter their prepared-to-commit state. If they all enter their prepared-to-commit states, they wait for the OTM to validate G. The validation can be performed using a *Global Serialization Graph* (GSG) test.[4] The nodes in GSG correspond to "recently" committed global transactions. For any pair of recently committed global transactions G_i^c and G_j^c, GSG contains a directed edge $G_i^c \rightarrow G_j^c$ if at least one subtransaction of G_i^c was serialized before (obtained a smaller ticket than) the subtransaction of G_j^c in the same LDBS. A strategy for node removal from the GSG is presented in Lemma 1 below.

Initially, GSG contains no cycles. During the validation of a global transaction G, OTM first creates a node for G in GSG. Then, it attempts to insert edges between G's node and nodes corresponding to every recently committed multidatabase transaction G^c. If the ticket obtained by a subtransaction of G at some LDBS is smaller (larger) than the ticket of the subtransaction of G^c there, an edge $G \rightarrow G^c$ ($G \leftarrow G^c$) is added to GSG. If all such edges can be added without creating a cycle in GSG, G is validated. Otherwise, G does not pass validation, its node together with all incident edges is removed from the graph, and G is restarted. This validation test is enclosed in a single critical section.[5]

G is also restarted, if at least one LDBS forces a subtransaction of G to abort for local concurrency control reasons (e.g., local deadlock), or its timeout expires (e.g., global deadlock). If more than one of the participating LDBSs uses a blocking mechanism for concurrency control, the timeouts mentioned above are necessary to resolve global deadlocks.

The timeout assigned to a global transaction G is based on a conservative estimate of the expected execution time of G. If it is difficult to estimate the expected duration of a global transaction G, an alternative solution is to set a different timeout for each subtransaction of G. The latter timeout strategy can be combined with a *wait-for graph* (WFG). The WFG is maintained by the MDBS and has LDBSs as nodes. If a cycle is found in the WFG, and the cycle involves LDBSs that use a blocking scheme to synchronize conflicting transactions, a deadlock is possible. MDBSs that maintain a WFG can resolve global deadlocks by setting timeouts only for operations issued at LDBSs that are involved in a WFG cycle and, in addition, use blocking to enforce local serializability and recoverability. In this paper, we do not discuss timeout strategies further, because the choice of the timeout strategy does not effect the correctness of OTM. A decentralized deadlock-free refinement of the Optimistic Ticket Method is described in [38].

As we mentioned, the serialization graph must contain only the nodes corresponding to recently committed global transactions. Below we provide a condition for safe removal of transaction nodes from the serialization graph.

Lemma 1: A node corresponding to a committed transaction G^c can be safely removed from the serialization graph if it has no incoming edges and all transactions that were active at the time G^c was committed are either committed or aborted. When a node is removed from the graph, all edges incident to the node can be also removed.

Proof: For a transaction node to participate in a serialization cycle it must have at least one incoming edge. No transaction started after the commitment of G^c can take its tickets before G^c, so it cannot add incoming edges to the node of G^c. Since we assume that G^c has no incoming

[4] Other validation tests such as the certification scheme proposed in [19] can be also used to validate global transactions.

[5] Including the validation test in a critical section has been originally proposed by Kung and Robinson in [32]. Several schemes have been proposed in the literature (e.g., the parallel validation schemes in [32], [36]) to deal with the possibility of bottlenecks caused by such critical sections. Although we could have adopted any of these schemes, there is no evidence that they allow more throughput than performing transaction validation serially, i.e., within a critical section as in OTM. Most commercial implementations of optimistic concurrency control protocols have chosen serial validation over parallel validation for similar reasons (e.g., Datacycle [37]).

edges and all transactions that were active at the time G^c was committed are finished, the node corresponding to G^c will be never involved in a serialization cycle. Therefore, is can be safely removed from the serialization graph. □

The following theorem proves the correctness of OTM.

Theorem 2: OTM guarantees global serializability if the following conditions hold:

1. the concurrency control mechanisms of the LDBSs ensure local serializability;
2. each multidatabase transaction has at most one subtransaction at each LDBS; and
3. each subtransaction has a visible prepared-to-commit state.

Proof: We have already shown that the order in which subtransactions take their tickets reflects their relative serialization order (Theorem 1). After the tickets are obtained by a global transaction at all sites it executes, OTM performs the global serialization test described earlier in this section. Global transactions pass validation and are allowed commit only if their relative serialization order is the same at all participating LDBSs. Lemma 1 shows that the the serialization test involving only the recently committed transactions is sufficient to guarantee global serializability. □

C. *Effect of the ticketing time on the performance of OTM*

OTM can process any number of multidatabase transactions concurrently, even if they conflict at multiple LDBS. However, since OTM forces the subtransactions of multidatabase transactions to directly conflict on the ticket, it may cause some subtransactions to get aborted or blocked because of ticket conflicts (Figure 4 (b)). Since subtransactions may take their tickets at any time during their lifetime without affecting the correctness of OTM, optimization based on the characteristics of each subtransaction (e.g., number, time and type of the data manipulation operations issued or their semantics) is possible. For example, if all global transactions conflict directly at some LDBS, there is no need for them to take tickets. To determine their relative serialization order there, it is sufficient to observe the order in which they issue their conflicting operations.

Choosing the right time to to take a ticket during the lifetime of a subtransaction can minimize the synchronization conflicts among subtransactions. For example, if a LDBS uses 2PL it is more appropriate to take the ticket immediately before a subtransaction enters its prepared-to-commit state. To show the effect of this convention consider a LDBS that uses 2PL for local concurrency control (Figure 5 (*a*)). 2PL requires that each subtransaction sets a write lock on the ticket before it increments its value. Given four concurrent subtransactions g_1, g_2, g_3 and g_4, g_1 does not interfere with g_2 which can take its ticket and commit before g_1 takes its ticket. Similarly, g_1 does not interfere with g_3, so g_1 can take its ticket and commit before g_3 takes its ticket. However, when g_4 attempts to take its ticket after g_1 has taken its ticket but before g_1 commits and releases its ticket lock, it gets blocked until g_1 is committed. The ticket values always reflect the serialization order of the subtransactions of multidatabase transactions but ticket conflicts are minimized if g_1 takes its ticket as close as possible to its commitment time.

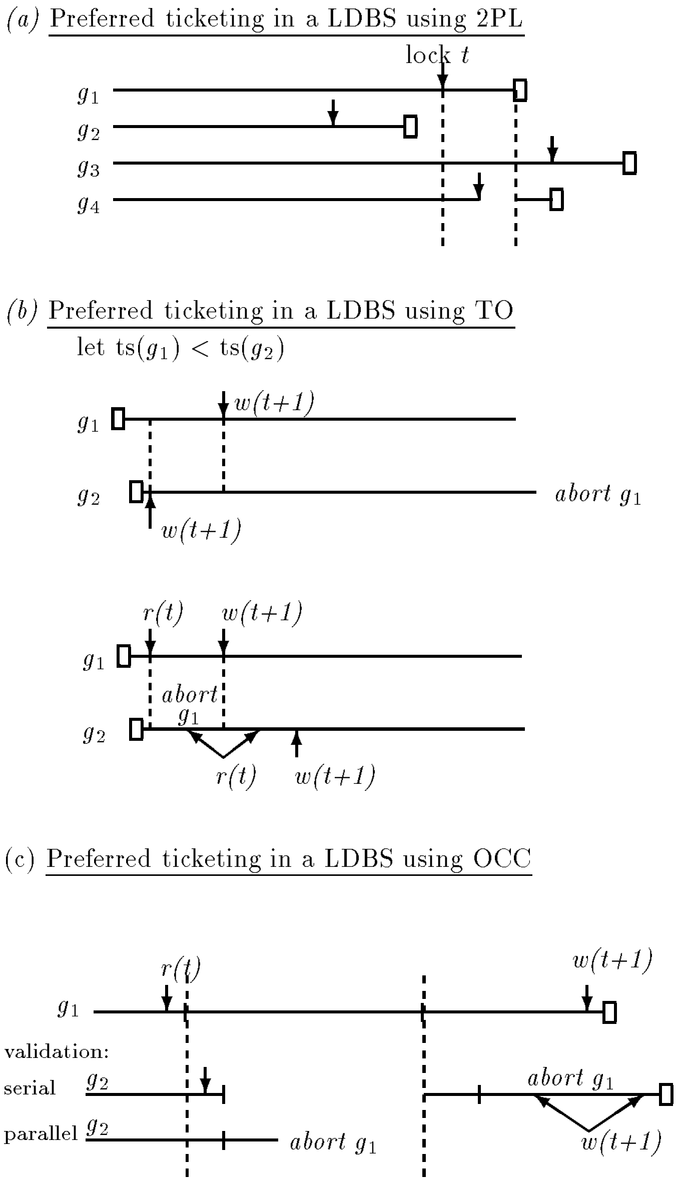


Fig. 5. Preferred ticketing in LDBSs.

If a LDBS uses timestamp ordering (TO) [17] (Figure 5 (*b*)), it is better to obtain the ticket when the subtransaction begins its execution. TO assigns a timestamp $ts(g_1)$ to a subtransaction g_1 when it begins its execution. Let g_2 be another subtransaction such that $ts(g_1) < ts(g_2)$. If the ticket obtained by g_1 has a larger value than the ticket of g_2 then g_1 is aborted. Clearly, if g_2 increments the ticket value before g_1 then, since g_2 is younger than g_1, either $r_{g_1}(ticket)$ or $w_{g_1}(ticket)$ conflicts with the $w_{g_2}(ticket)$ and g_1 is aborted. Hence, only g_1 is allowed to increment the ticket value before g_2. Similarly, if g_2 reads the ticket before g_1 increments it, then when g_1 issues $w_{g_1}(ticket)$ it conflicts with the $r_{g_2}(ticket)$ operation issued before and g_1 is aborted. Therefore, given that $ts(g_1) < ts(g_2)$, either g_1 takes its ticket before g_2 or g_1 is aborted. Hence, it is better for subtransactions to take their tickets as close as possible to the point they are assigned their timestamps under TO, i.e., at the beginning of their execution.

Another significant optimization can be used to completely eliminate tickets in LDBSs that use TO schedulers. Let g_1 and g_2 be a pair of subtransactions that do not take tickets. Since transactions under the control of a TO scheduler are assigned their timestamp some time between their submission and the time they complete their first database operation, the global scheduler can ensure that g_1 obtains a local timestamp smaller than the timestamp of g_2 by delaying the submission of g_2 until g_1 completes its first database operation. By using this technique, the global scheduler can ensure that the submission order of the subtransactions determines their local serialization order and that g_1 is serialized before g_2 in the local system.

Finally, if a LDBS uses an optimistic concurrency control (OCC) [32] protocol there is no best time for the subtransactions to take their tickets (Figure 5 (c)). Transactions under the control of OCC have a read phase that is followed by a validation phase. OCC uses transaction readsets and writesets to validate transactions. Only transactions that pass validation enter a write phase. Thus, each subtransaction g_1 reads the ticket value before it starts its (serial or parallel) validation but increments it at the end of its write phase. If another transaction g_2 is able to increment the ticket in the meantime, g_1 does not pass validation and is restarted.

The basic advantages of OTM are that it requires the local systems to ensure only local serializability and that the optimistic global scheduler imposes no restrictions on the local execution of global transactions. Its main disadvantages are the following:

- under optimistic scheduling global restarts are possible,
- the global scheduler must maintain a GSG, and
- tickets introduce additional conflicts between global transactions which may not conflict otherwise.

In the following three sections we describe solutions that address these issues, respectively.

V. The Conservative Ticket Method (CTM)

OTM does not affect the way in which the LDBSs handle the execution of global transactions up to the point in which their subtransactions enter their prepared-to-commit state. Optimistic global schedulers based on uncontrolled local execution of the global subtransactions, such as OTM, are easier to implement and in some cases allow more concurrency than conservative schedulers. However, since optimistic global schedulers allow global transactions to take their tickets in any order, they suffer from *global restarts* caused by out-of-order ticket operations. To explain the problem of global restarts consider a situation in which a global transaction G_i obtains its ticket before another global transaction G_j at some LDBS. If in another LDBS G_j is able to obtain its ticket before G_i, the MDBS scheduler aborts and restarts either G_i or G_j to disallow the globally non-serializable execution of their ticket operations. In multidatabase systems in which the participating LDBSs use blocking for local concurrency control, the incompatible orders in which G_i and G_j take their tickets in different LDBSs causes a *global deadlock*. To resolve such a global deadlock the OTM scheduler aborts and restarts the global transaction whose timeout expires first. If the LDBSs do not use blocking for local concurrency control, then incompatible execution orders of ticket operations cause a cycle in the GSG. In this case, the global transaction that enters global validation last is rejected, and the OTM scheduler aborts it.

In this section we describe CTM, a method for multidatabase transaction management that eliminates global restarts. Like OTM, CTM requires subtransactions of global transactions to take tickets at their corresponding LDBSs. However, unlike OTM, CTM controls the order in which the subtransactions take their tickets. To avoid global restarts, CTM ensures that the relative order in which global transaction take their tickets is the same in all participating LDBS.

CTM requires that all subtransactions of global transactions have a visible *prepared-to-Take-A-Ticket* state in addition to a visible prepared-to-commit state. A subtransaction enters its prepared-to-Take-A-Ticket state when it successfully completes the execution of all its database operations that precede the Take-A-Ticket operations and leaves this state when it reads the ticket value. The visible prepared-to-Take-A-Ticket state can be provided by the multidatabase system by employing the same techniques that simulate the prepared-to-commit state. For example, one way to make the prepared-to-Take-A-Ticket state of a subtransaction visible, is to force a handshake after each database operation that precedes the Take-A-Ticket operations. That is, if all operations that precede the Take-A-Ticket operations are completed successfully, the MDBS can be certain that the subtransaction has entered its prepared-to-Take-A-Ticket state. We say that a global transaction becomes prepared to take its tickets when *all* its subtransactions enter their prepared-to-Take-A-Ticket state.

CTM processes a set $\mathcal{G}$ of global transactions as follows. Initially, the CTM sets a timeout for each global transaction in $\mathcal{G}$, and then submits its subtransactions to the corresponding LDBSs. The subtransactions of all global transactions are allowed to interleave under the control of the LDBSs until they enter their prepared-to-Take-A-Ticket state. Without loss of generality, suppose that the subtransactions of global transactions G_1, G_2, …, G_k in $\mathcal{G}$ become prepared to take their tickets before their timeout expires. Furthermore, suppose that a subtransaction of G_2 enters its prepared-to-Take-A-Ticket state after all subtransactions of G_1 become prepared to take their tickets (i.e., G_1 becomes prepared to take its tickets before

G_2); a subtransaction of G_3 becomes prepared to take its ticket after all subtransactions of G_2 enter their prepared-to-Take-A-Ticket state (i.e., G_2 becomes prepared to take its tickets before G_3); ...; and a subtransaction of G_k enters its prepared-to-Take-A-Ticket state after all subtransactions of G_{k-1} become prepared to take tickets (i.e., G_{k-1} becomes prepared to take its tickets before G_k). The CTM allows the subtransactions of such global transactions G_1, G_2, ..., G_k to take their tickets in the following order: the subtransactions of G_1 take their tickets before the subtransactions of G_2, the subtransactions of G_2 take tickets before the subtransactions of G_3, ..., the subtransactions of G_{k-1} take their tickets before the subtransactions of G_k.

Global transactions are allowed to commit only if all their subtransactions successfully take their tickets and report their prepared-to-commit state. On the other hand, the MDBSs abort and restart any multidatabase transaction that has a subtransaction that did not report its prepared-to-commit state before its timeout expired. Local optimizations discussed in Section IV-C can also be applied on CTM.

Theorem 3: CTM guarantees global serializability and it is free of global restarts if the following conditions are satisfied:

1. the concurrency control mechanisms of the LDBSs ensure local serializability;
2. each multidatabase transaction has at most one subtransaction at each LDBS; and
3. each subtransaction has a visible prepared-to-Take-A-Ticket and a visible prepared-to-commit state.

Proof: Without loss of generality, suppose that global transactions in a set $\mathcal{G}$ become prepared to take their tickets in the following order: G_1, G_2, ..., G_k. Under the control of CTM, G_1 takes all its tickets before G_2 takes its tickets, G_2 takes tickets before G_3, ..., G_{k-1} takes its tickets before G_k. Since CTM ensures that the relative order in which the subtransactions of each global transaction take their tickets is the same in all participating LDBS and we have proven that the order in which the subtransactions take their tickets reflects their relative serialization order (Theorem 1), CTM guarantees global serializability and avoids global restarts due to ticket conflicts. □

Another important property of CTM is that it does not require a GSG. Hence, the global CTM scheduler is simpler than the global OTM scheduler. An optimistic scheduler that does not require a GSG is described next.

VI. Cascadeless Tickets Methods

To ensure correctness in the presence of failures and to simplify recovery and concurrency control, transaction management mechanisms used in database management systems often ensure not only serializability and recoverability [17] but also one of the properties defined below:

- A transaction management mechanism is *cascadeless* [17] if each transaction may read only data objects written by committed transactions.
- A transaction management mechanism is *strict* [17] if no data object may be read or written until the transactions that previously wrote it commit or abort.

Many commercial DBMSs allow only strict schedules to eliminate cascading aborts and also to be able to ensure database consistency when before images are used for database recovery.

¿From the perspective of the multidatabase scheduler, the cascadelessness of the LDBSs is important because it can be used to eliminate the GSG (Global Serialization Graph) test required by OTM. To take advantage of cascadeless LDBSs, we introduce a refinement of OTM, called the *Cascadeless* OTM. Like OTM, the Cascadeless OTM ensures global serializability by preventing the subtransactions of each multidatabase transaction from being serialized in different ways at their corresponding LDBSs. Unlike OTM, Cascadeless OTM takes advantage of the fact that if all LDBSs produce cascadeless schedules then global transactions cannot take tickets and commit, unless their tickets have the same relative order at all LDBSs.

Cascadeless OTM processes each global transaction G as follows. Initially, the MDBS sets a timeout for G and submits its subtransactions to the appropriate LDBSs. All subtransactions are allowed to interleave under the control of the LDBSs until they enter their prepared-to-commit state. If all subtransactions of G take their tickets and report their prepared-to-commit state, the Cascadeless OTM allows G to commit. Otherwise, the MDBSs abort and restart any global transaction that has a subtransaction that did not report its prepared-to-commit state before the timeout of G expired. Local optimizations mentioned in Section IV-C can be also applied on Cascadeless OTM.

Theorem 4: Cascadeless OTM guarantees global serializability if the following conditions are satisfied:

1. the concurrency control mechanisms of the LDBSs ensure local serializability and cascadelessness;
2. each multidatabase transaction has at most one subtransaction at each LDBS; and
3. each subtransaction has a visible prepared-to-commit state.

Proof: We have already shown that the order in which the subtransactions take their tickets reflects their relative serialization order (Theorem 1). To prove that global serializability is enforced without a GSG test, consider *any* pair of global transactions G_i and G_j in a set $\mathcal{G}$ having subtransactions in multiple LDBSs, including LDBS$_k$ and LDBS$_l$. Without loss of generality assume that at LDBS$_k$ the subtransaction of G_i takes its ticket before the subtransaction of G_j, but at LDBS$_l$ the subtransaction of G_j takes its ticket before the subtransaction of G_i. Since the LDBSs are cascadeless, G_j cannot write its ticket value at LDBS$_k$ before G_i commits, and G_i cannot write its ticket at LDBS$_l$ before G_j commits. Therefore, there are two possible outcomes for the execution of a global transaction under Cascadeless OTM. Either the tickets of its subtransactions have the same relative order at all LDBSs and global serializability is ensured, or it has at least one subtransaction that cannot commit. □

Like the OTM, the Cascadeless OTM is not free of global restarts. A *Cascadeless* CTM which is similar to CTM can be used to deal with global restarts.

While local cascadelessness can be used to simplify the global optimistic scheduler (i.e., there is no need to maintain a GSG), strictness offers no additional advantages over cascadelessness. In the following section we show that if the schedulers of local systems meet additional conditions, ticket conflicts can be eliminated.

VII. Implicit Tickets and the Implicit Ticket Method (ITM)

We have argued that the basic problem in multidatabase concurrency control is that the local serialization orders do not necessarily reflect the order in which global transactions are submitted, perform their operations or commit in the LDBSs. To deal with this problem we have introduced the concept of the ticket and proposed several methods that must take tickets to ensure global serializability. However, tickets introduce additional conflicts between global transactions that may not conflict otherwise. Thus, it is desirable to eliminate tickets whenever possible. In the following sections we identify classes of schedules that include events that can be used to determine the local serialization order of transactions without forcing conflicts between global transactions. We refer to such events as *implicit tickets*.

A. Determining the local serialization order

In Section IV-C, we have discussed how to eliminate tickets in LDBSs that use TO for local concurrency control. This approach can be applied to all LDBSs that allow transactions to commit only if their respective local serialization order reflect their local submission order. That is, in the subclass of LDBSs that allow schedules in which the transaction submission order determines their serialization order, the order transactions issue their *begin* operations constitutes their implicit tickets.

Another important class of local systems in which global transactions do not have to take tickets includes LDBSs that allow only schedules in which the local commitment order of transactions determines their local serialization order, i.e., the order transactions perform their *commit* operations constitutes their implicit tickets. In [6], [7], we have defined the class of schedules that transactions have *analogous execution (commitment) and serialization order* as follows:

Definition 1: Let S be a serializable schedule. We say that the transactions in S have *analogous execution and serialization order* if for any pair of transactions T_i and T_j such that T_i is committed before T_j in S, T_i is also serialized before T_j in S.

The property of analogous execution and serialization orders applies to both view serializable and conflict serializable schedules and is difficult to enforce directly. The subclass of schedules that are conflict serializable and have analogous executions and serialization order is characterized in terms of *strong recoverability* [7] defined below.

Definition 2: Let S be a schedule. We say that S is *strongly recoverable* if for any pair of committed transactions T_i and T_j, whenever an operation $op_{T_i}(x)$ of T_i precedes an operation $op_{T_j}(x)$ of T_j in S and these operations conflict (at least one of these operations is a write), then $commit_{T_i}$ precedes $commit_{T_j}$ in S.

A transaction management mechanism is strongly recoverable if its produces only strongly recoverable schedules. In [7], we have shown that if a transaction management mechanism is strongly recoverable, it produces conflict serializable schedules in which transaction execution and serialization orders are analogous. The significance of strong recoverability in simplifying the enforcement of global serializability in multidatabase systems has been recognized in the literature. For example, the notion of *commitment ordering* proposed in [39], [40] as a solution to enforce global serializability without taking tickets is identical to strong recoverability.

Although strongly recoverable schedulers can be realized in real DBMSs, most real transaction management mechanisms produce schedules that satisfy stronger properties that are easier to enforce.

The notion of *rigorous* schedules [6], [7] defined next effectively eliminates conflicts between uncommitted transactions. Thus, it provides an even simpler way to ensure that transaction execution and serialization orders are analogous.

Definition 3: A schedule is *rigorous* if the following two conditions hold: (i) it is strict, and (ii) no data item is written until the transactions that previously read it commit or abort.

We say that a transaction management mechanism is rigorous if it produces rigorous schedules, and we use the term *rigorous LDBS* to refer to a LDBS that uses a rigorous scheduler. In [6] we have shown that if a transaction management mechanism ensures rigorousness, it produces (conflict) serializable schedules in which transaction execution and serialization orders are analogous. In [7] we proved that the class of rigorous schedules is a subclass of strongly recoverable schedules.

The class of rigorous transaction management mechanisms includes several common conservative schedulers [6], [7], such as conservative TO [17] and *rigorous* two-phase locking (2PL) (i.e., the variant of strict 2PL under which a transaction must hold its read and write locks until it terminates). Rigorous variations of TO and optimistic concurrency control [32] protocols have been introduced in [6], [7]. However, while many conservative schedulers are rigorous, enforcing rigorousness is too restrictive for optimistic schedulers, i.e., rigorous optimistic schedulers behave like conservative schedulers.

The following class of schedules permits optimistic synchronization of operations.

Definition 4: A schedule is *semi-rigorous* if its committed projection is rigorous.

Semi-rigorousness permits validation of transactions after they have finished all their operations. Therefore, it simplifies the design of optimistic schedulers. Most real optimistic schedulers, including the schedulers described in [32], allow only semi-rigorous schedules. While semi-rigorousness simplifies optimistic concurrency control, it

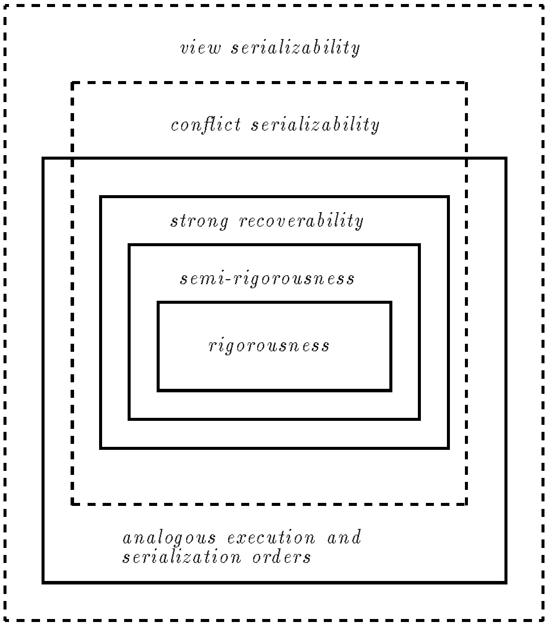


Fig. 6. Relationship among analogous execution and serialization orders, strong recoverability, semi-rigorousness and rigorousness.

does not ensure recoverability as it is defined in [17]. Therefore, most optimistic schedulers ensure cascadelessness or strictness in addition to semi-rigorousness. For example, schedulers that use the optimistic protocol with serial validation [32] permit schedules that in addition to being semi-rigorous that are also strict.

The class of semi-rigorous schedules includes the superclass of rigorous schedules and is a subclass of strongly recoverable schedules. The relationship among analogous execution and serialization orders, strong recoverability, semi-rigorousness, and rigorousness is depicted in Figure 6.

Finally, note that strictness is not sufficient to ensure that the transaction execution order is analogous to the transaction serialization order. For example, if we assume that transactions commit immediately after they complete their last operation, the schedule at $LDBS_2$ in Figure 2 is strict, but the the execution order of the transactions is not analogous to their serialization order.

B. Enforcing global serializability

To take advantage of LDBSs that allow only analogous execution and serialization orders, we introduce the *Implicit Ticket Method* (ITM). Like OTM, ITM ensures global serializability by preventing the subtransactions of each multidatabase transaction from being serialized in different ways at their corresponding LDBSs. Unlike OTM, ITM does not need to maintain tickets and the subtransactions of global transactions do not need to take and increment tickets explicitly. In LDBSs that allow only analogous execution and serialization orders, the implicit ticket of each subtransaction executed there is determined by its commitment order. That is, the order in which we commit subtransactions at each LDBS determines the relative values of their implicit tickets. To achieve global serializability, ITM controls the commitment order and thus the serialization order of multidatabase subtransactions as follows.

Assuming rigorous LDBSs, ITM guarantees that for any pair of multidatabase transactions G_i and G_j, either the subtransactions of G_i are committed before the subtransactions of G_j, or the subtransactions of G_j are committed prior to the subtransactions of G_i. This can be easily enforced by a distributed agreement protocol such as the 2PC protocol.

ITM processes a set $\mathcal{G}$ of global transactions as follows. Initially, the ITM sets a timeout for each global transaction in $\mathcal{G}$, and then submits its subtransactions to the corresponding LDBSs. The subtransactions of all global transactions are allowed to interleave under the control of the LDBSs until they enter their prepared-to-commit state. Without loss of generality, suppose that the subtransactions of global transactions G_1, G_2, …, G_k in $\mathcal{G}$ become prepared to commit before their timeout expires. Furthermore, suppose that a subtransaction of G_2 enters its prepared-to-commit state after all subtransactions of G_1 become prepared to commit, a subtransaction of G_3 becomes prepared to commit after all subtransactions of G_2 enter their prepared-to-commit state, …, and a subtransaction of G_k enters its prepared-to-commit state after all subtransactions of G_{k-1} become prepared to commit. The ITM allows the subtransactions of such global transactions to commit in the following order: the subtransactions of G_1 before the subtransactions of G_2, the subtransactions of G_2 before the subtransactions of G_3, …, the subtransactions of G_{k-1} before the subtransactions of G_k. Global transactions that have one or more subtransactions that do not report their prepared-to-commit state before their timeout expires are aborted and restarted by the MDBS.

Theorem 5: ITM ensures global serializability if the following conditions hold:

1. the concurrency control mechanisms of the LDBSs ensure analogous executions and serialization orders;
2. each multidatabase transaction has at most one subtransaction at each LDBS; and
3. each subtransaction has a visible prepared-to-commit state.

Proof: Without loss of generality, suppose that global transactions in a set $\mathcal{G}$ enter their prepared to commit state in the following order: G_1, G_2, …, G_k. Under the control of ITM, the subtransaction of G_1 commit before the subtransactions of G_2, the subtransaction G_2 commit before the subtransaction of G_3, …, and the subtransactions of G_{k-1} commit before the subtransactions of G_k. Since ITM ensures that the relative order in which the subtransactions of each global transaction commit is the same in all participating LDBSs and the LDBSs ensure that the subtransaction commitment order reflects their relative serialization order, ITM guarantees global serializability. □

VIII. Mixed Methods

In a multidatabase environment where rigorous, cascadeless, and non-cascadeless LDBSs participate, mixed ticket methods that combine two or more of the methods described in the previous sections of this paper can be used to ensure global serializability. In this section we describe a mixed ticket method that combines OTM, CTM, and their cascadeless variations with ITM.

A mixed method processes a multidatabase transaction G as follows:

1. Sets a timeout for G and submits its subtransactions to the corresponding LDBSs.
2. Subtransactions that are controlled by ITM, OTM, and the cascadeless variation of OTM are allowed to interleave until they enter their prepared-to-commit state. Subtransactions that are controlled by CTM and the cascadeless CTM are allowed to proceed until they enter their prepared-to-Take-A-Ticket state.
3. If all subtransactions under the control of OTM, and the cascadeless OTM take tickets and report their prepared-to-commit state, global validation is applied to make sure that these subtransactions are serialized the same way. If G does not pass global validation, it is aborted.
4. Subtransactions under the control of CTM and the cascadeless CTM are allowed to take their tickets according to the serialization order of G determined earlier by the validation process. To ensure this, the mixed method delays the Take-A-Ticket operations of the subtransactions of G that execute under the control of CTM and the cascadeless CTM until there is no uncommitted global transaction G' such that:
   - G' has subtransactions that have not taken their tickets, and
   - there is at least one LDBS in which the subtransaction of G' has taken its ticket before the subtransaction of G.

   If there is no global transaction that satisfies these conditions, the mixed method allows the the subtransactions of G to take their tickets under the control of CTM.
5. If all subtransactions of G enter their prepared-to-commit states, the mixed method commits G. Other global transactions are allowed to commit either before the first subtransaction of G commits, or after the commitment of all subtransactions of G.
6. If the timeout expires in any of these steps, the MDBSs aborts and restarts G.

Simpler mixed methods, e.g., combining only optimistic or only conservative ticket methods, can be developed similarly.

IX. Summary and Conclusion

Enforcing the serializability of global transactions in a MDBS environment is much harder than in distributed databases systems. The additional difficulties in this environment are caused by the autonomy and the heterogeneity of the participating LDBSs.

To enforce global serializability we introduced OTM, an optimistic multidatabase transaction management mechanism that permits the commitment of multidatabase transactions only if their relative serialization order is the same in all participating LDBSs. OTM requires LDBSs to guarantee only local serializability. The basic idea in OTM is to create direct conflicts between multidatabase transactions at each LDBS that allow us to determine the relative serialization order of their subtransactions.

We have also introduced a Conservative Ticket Method, CTM. Under CTM, global transactions must take tickets, but CTM does not require global serialization testing and eliminates global restarts due to failed validation. Refinements of OTM and CTM for multidatabase environments where all participating LDBSs are cascadeless, may use simpler global schedulers. Unless the subtransactions of multidatabase transactions take their tickets at approximately the same time (e.g., the subtransactions of each global transaction take their tickets at the end of their execution and their duration is approximately the same), conservative ticket methods may allow a higher throughput than the corresponding optimistic ticket methods.

To take advantage of additional properties of LDBSs we proposed the Implicit Ticket Method. ITM eliminates ticket conflicts, but works only if the participating LDBSs disallow schedules in which transaction execution and serialization orders are not analogous. ITM uses the local commitment order of each subtransaction to determine its implicit ticket value. It achieves global serializability by controlling the commitment (execution) order and thus the serialization order of multidatabase transactions. Compared to the the ADDS approach and Altruistic Locking, ITM can process any number of multidatabase transactions concurrently, even if they have concurrent and conflicting subtransactions at multiple sites. Both OTM and ITM do not violate the autonomy of the LDBSs and can be combined in a single comprehensive mechanism.

Analogous transaction execution and serialization orders is a very useful property in a MDBS. For example, it can be shown that the ADDS scheme [10], [18], Altruistic Locking [13], and 2PC Agent Method [15] produce globally serializable schedules if the participating LDBSs disallow schedules in which transaction execution and serialization orders are not analogous. Similarly, quasi-serializable schedules [23] become serializable if all LDBSs permit only analogous transaction execution and serialization orders. On the other hand, if the local systems allow schedules in which transaction execution and serialization orders are not analogous, these methods may lead to schedules that are not globally serializable.

Another important finding is that local strictness in a multidatabase environment offers no advantage over cascadelessness in simplifying the enforcement of global serializability.

Further research and prototyping are currently performed at GTE Laboratories, Bellcore, and the University of Houston. These activities include performance evaluation of the proposed ticket methods, and benchmarking of a prototype implementation. Current research conducted at GTE Laboratories, includes adaptation of ticket methods to provide consistency in a Distributed Object Management System (DOMS) [8] in which global transactions access homogeneous objects that encapsulate autonomous concurrency control mechanisms, and/or attached objects that represent data and functionality of autonomous and heterogeneous LDBSs.

The Take-A-Ticket operation can be viewed as a function that returns the serialization order of a transaction in a LDBS. If such a function is provided by the interfaces of future DBMSs, multidatabase transaction management methods that use tickets to enforce global serializability can substitute the ticket operations by calls to DBMS-provided serialization order functions and continue to enforce global serializability without any modification.

Acknowledgments

The idea to use tickets in multidatabase transaction management had emerged during a discussion with Gomer Thomas. We thank Yuri Breitbart for pointing out an error in one of our definitions in an earlier version of this paper. Piotr Krychniak has implemented some of the ticket methods in real DBMSs and contributed to the discussion of implementation issues in Appendix I. We also thank Mark Hornick and Ole Anfindsen for their useful comments.

References

[1] W. Litwin, "From database systems to multidatabase systems: Why and how", in *British National Conference on Databases*. Cambridge Press, 1988.

[2] A. Sheth and J. Larson, "Federated databases: Architectures and integration", *ACM Computer Surveys*, September 1990, To be published.

[3] J. Veijalainen, *Transaction Concepts in Autonomous Database Environments*, R. OLDENBOURG VERLAG, 1990.

[4] H. Garcia-Molina and B. Kogan, "Node autonomy in distributed systems", in *Proceedings of International Symposium on Databases in Parallel and Distributed Systems*, December 1988.

[5] A. Elmagarmid W. Du and W. Kim, "Effects of local autonomy on heterogeneous distributed database systems", Tech. Rep. TR ACT_OODS_EI_059-90, Microelectronics and Computer Corporation, Austin TX, February 1990.

[6] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz, "Rigorous scheduling in multidatabase systems", in *Workshop in Multidatabases and Semantic Interoperability*, October 1990.

[7] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz, "On rigorous transaction scheduling", *IEEE Transactions on Software Engineering*, vol. 17, no. 9, 1991.

[8] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie, "Distributed object management", *International Journal of Intelligent and Cooperative Information Systems*, vol. 1, no. 1, March 1992.

[9] V.D. Gligor and R. Popescu-Zeletin, "Concurrency control issues in distributed heterogeneous database management systems", in *Distributed Data Sharing Systems*, F.A. Schreber and W. Litwin, Eds. North-Holland, 1985.

[10] Y. Breitbart and A. Silberschatz, "Multidatabase update issues", in *Proceedings of ACM SIGMOD International Conference on Management of Data*, June 1988.

[11] A.K. Elmagarmid and A.A. Helal, "Supporting updates in heterogeneous distributed database systems", in *IEEE Proceedings of the 4th International Conference on Data Engineering*, 1988.

[12] T. Logan and A. Sheth, "Concurrency control issues in heterogeneous distributed database management systems", Unpublished, July 1986.

[13] R. Alonso, H. Garcia-Molina, and K. Salem, "Concurrency control and recovery for global procedures in federated database systems", *A quarterly bulletin of the Computer Society of the IEEE technical committee on Data Engineering*, vol. 10, no. 3, September 1987.

[14] W. Du, A. Elmagarmid, Y. Leu, and S. Osterman, "Effects of autonomy on maintaining global serializability in heterogeneous distributed database systems", in *Proceedings of the second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, October 1989.

[15] A. Wolski and J. Veijalainen, "2PC Agent method: Achieving serializability in presence of failures in a heterogeneous multidatabase", in *Proceedings of PARBASE-90 Conference*, February 1990.

[16] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The notions of consistency and predicate locks in a database system", *Communications of ACM*, vol. 19, no. 11, November 1976.

[17] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[18] Y. Breitbart, A. Silberschatz, and G. Thompson, "An update mechanism for multidatabase systems", *A quarterly bulletin of the Computer Society of the IEEE technical committee on Data Engineering*, vol. 10, no. 3, September 1987.

[19] C. Pu, "Superdatabases for composition of heterogeneous databases", in *IEEE Proceedings of the 4th International Conference on Data Engineering*, 1988.

[20] J. Pons and J. Vilarem, "Mixed concurrency control: Dealing with heterogeneity in distributed database systems", in *Proceedings of the Fourteenth International VLDB Conference*, August 1988, Los Angeles.

[21] J.N. Gray, "The transaction concept: Virtues and limitations", in *Proceedings of the 7th International Conference on VLDB*, September 1981.

[22] H. Garcia-Molina and K. Salem, "SAGAS", in *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.

[23] W. Du and A. Elmagarmid, "QSR: A correctness criterion for global concurrency control in InterBase", in *Proceedings of the 15th International Conference on Very Large Databases*, 1989.

[24] J. Veijalainen, F. Eliassen, and B. Holtkamp, "The S-Transaction model", in *Advanced Transaction Models for New Applications*, A. Elmagarmid, Ed. Morgan-Kaufmann, 1992.

[25] M. Rusinkiewicz, A. Elmagarmid, Y. Leu, and W. Litwin, "Extending the transaction model to capture more meaning", *SIGMOD record*, vol. 19, no. 1, March 1990.

[26] J. N. Gray, *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, Springer-Verlag, 1978.

[27] C.J. Date, *A Guide to The SQL Standard*, Addison-Wesley Publishing Company, 1987.

[28] Generic RDA Editor (Joel S. Berson), *Information Processing Systems - Open Systems Interconnection - Remote Database Access - Part 1: Generic Model, Service, and Protocol (ISO/IEC JTC 1/SC 21 WG3)*, ISO, 1990.

[29] D. Georgakopoulos, *Transaction Management in Multidatabase Systems*, PhD thesis, University of Houston, Department of Computer Science, 1990.

[30] D. Georgakopoulos, "Multidatabase recoverability and recovery", in *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, 1991.

[31] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, "System level concurrency control for distributed database systems", *ACM Transactions on Database Systems*, vol. 3, no. 2, June 1978.

[32] H.T. Kung and J.T. Robinson, "On optimistic methods for concurrency control", *ACM TODS*, vol. 6, no. 2, June 1981.

[33] A. Wolski and J. Veijalainen, "Prepare and commit certification for decentralized transaction management in rigorous heterogeneous multidatabases", in *Proceedings of the 8th International Conference on Data Engineering*, February 1992.

[34] D. Georgakopoulos and M. Rusinkiewicz, "Transaction managemement in multidatabase systems", Tech. Rep. UH-CS-89-20, Department of Computer Science, University of Houston, September 1989.

[35] Y. Breitbart, A. Silberschatz, and G. Thompson, "Reliable transaction management in a multidatabase system", Tech. Rep. 157-90, University of Kentucky, 1990.

[36] C.U. Orji, L. Lillien, and J. Hyziak, "A performance analysis of an optimistic and a basic timestamp-ordering concurrency control algorithms for centralized database systems", in *IEEE Proceedings of the 4th International Conference on Data Engineering*, 1988.

[37] T. F. Bowen, G. Gopal, G. Herman, T. Hickey, K. C. Lee, W.H. Mansfield, J. Raitz, and A. Weinrib, "The datacycle(tm) archi-

tecture", *Communications of ACM*, vol. 35, no. 12, December 1992.

[38] R. Batra, M. Rusinkiewicz, and D. Georgakopoulos, "A decentralized deadlock-free concurrency control method for multidatabase transactions", in *Proceedings of 12th International Conference on Distributed Computing Systems*, June 1992.

[39] Yoav Raz, "Extended commitment ordering, or guaranteeing global serializability by applying commitment order selectively to global transactions", Tech. Rep. DEC-TR-842, Digital Equipment Corporation, November 1991.

[40] Yoav Raz, "The commitment order coordinator (coco) of a resource manager, or architecture for distributed commitment ordering based concurrency control", Tech. Rep. DEC-TR-843, Digital Equipment Corporation, December 1991.

Appendix

I. Implementation Issues

System interfaces of many real DBMSs are at the level of set-oriented queries and updates (e.g., SQL, QUEL). Transactions are implemented in a high-level programming language that includes DBMS calls embedded in the transaction program. Such calls are supported by an embedded language interface provided by the DBMS. In this paper, we have modeled global transactions, their subtransactions, and the local transactions as collections of read and write operations. We have chosen the read/write transaction model to simplify the discussion of problems and corresponding solutions in enforcing global serializability in a multidatabase environment. The use of the read/write model to describe transaction management issues neither limits the generality of the proposed solutions, nor makes it more difficult to apply them in a LDBS that supports interfaces at the level of set-oriented queries and updates. To support this claim, we illustrate the implementation of the ticket data object in a relational DBMS that support only an SQL interface.

To a DBMS the MDBS appears as a regular user. To create the ticket data object, the MDBS creates a relation that has only one row and a single integer column. We refer to this relation as the *ticket_relation*, and we refer to the integer value stored in this relation as the *ticket_value*. To create the ticket data object the MDBS performs the following commands:

```
CREATE TABLE OWNER=mdbs
TABLE_NAME=ticket_table
(COLUMN_NAME=ticket_value DATATYPE=integer);

REVOKE INSERT, DELETE, UPDATE, SELECT
ON (TABLE_NAME=ticket_table)
FROM ALL_USERS;
```

The last statement is required to prevent local transactions from accessing the ticket.

To take tickets, the MDBS augments each subtransaction of a global transaction with the following statements that read and increment the ticket value.

```
UPDATE ticket_table
SET ticket_value = ticket_value + 1;
```

Dimitrios Georgakopoulos received the B.S. degree in pure mathematics from Aristotelio University of Thessaloniki, Greece, in 1982, and the M.S. and Ph.D. degrees from the University of Houston, Houston, Texas, in 1986 and 1990, respectively. He is currently a Senior Member of Technical Staff in the Distributed Object Computing Department at GTE Labs, where he specializes in the areas of transaction and workflow processing. His research interests include distributed object computing systems, multidatabase systems, interoperable systems. Dr. Georgakopoulos has received the IEEE Computer Society Outstanding Paper Award. He is a member of the ACM and the IEEE Computer Society.

Marek Rusinkiewicz Marek Rusinkiewicz is Professor of Computer Science at the University of Houston. His research interests include heterogeneous database systems, distributed computing systems, query languages, and transaction processing. He has published numerous journal and conference papers and has consulted for industry and government organizations in these areas. Rusinkiewicz is the program chairman for the 1994 IEEE-CS International Conference on Data Engineering.

Amit Sheth Amit P. Sheth has led projects on developing a heterogeneous distributed database system, integration of AI systems with database systems, and tools for schema integration and view update. Currently Dr. Sheth is working on management of transactional workflows and interdependent data, as well as a developing a corporate heterogeneous information management environment (CHIME). He was the general chair of the 1st Intl. Conf. on Parallel and Distributed Systems (PDIS) and a program co-chair of RIDE-IMS'93. Currently he is an ACM lecturer and the program co-chair of the 3rd PDIS.

Dr. Sheth is a member of IEEE-CS and ACM.