1988

Effects of Autonomy on Maintaining Global Serializability in Heterogeneous Database Systems

W. Du

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Y. Leu

S. F. Ostermann

Report Number:
88-835

EFFECTS OF AUTONOMY ON MAINTAINING GLOBAL SERIALIZABILITY IN HETEROGENEOUS DATABASE SYSTEMS

W. Du
A.K. Elmagarmid
Y. Leu
S.F. Ostermann

CSD-TR-835
December 1988
Revised April 1989

Effects of Local Autonomy on Global Concurrency Control in Heterogeneous Database Systems [1]

W. Du, A. K. Elmagarmid, Y. Leu and S. D. Ostermann
Department of Computer Science
Purdue University
West Lafayette, IN 47907
(317)-494-1998
ahmed@cs.purdue.edu

[1]This work is supported by a PYI Award from NSF and grants from AT&T Foundation and Tektronix.

Abstract

A heterogeneous database system (HDBS) is a system which integrates pre-existing database systems to support global applications accessing more than one database. In this paper, we study the difficulties and general approaches of global concurrency control in HDBSs. In particular, we discuss the difficulties of maintaining global serializability in HDBSs. This paper was originally motivated by the difficulties encountered when designing a global concurrency control strategy for a distributed HDBS. The need for new strategies is presented in this paper, as well as counter examples to existing algorithms. An assumption usually made in addressing multidatabase systems is that the element databases are autonomous. The meaning of autonomy and its effects on designing global concurrency control algorithms are addressed. It is the goal of this paper to motivate other researchers to realize the need for new correctness criteria by which concurrency control algorithms can be validated.

1 Introduction

A heterogeneous database system (HDBS) is a system which interconnects pre-existing database systems to support global applications that access data items in more than one database. The heterogeneities among various database systems include differences in data models (network, hierarchical and relational) and transaction management strategies (concurrency control and recovery) [GP85]. In order to provide a correct environment for the execution of global updates, a global concurrency controller must be provided to control the concurrent execution of global transactions.

Recently, several concurrency control protocols have been proposed [GL84] [GP85] [Pu88] [BST87] [EH88] [BS88b]. In our opinion, these algorithms have underestimated the difficulty of the problem and its related dimensions. In this paper, we discuss the effects of local autonomy on global concurrency control. In particular, we discuss the difficulties of maintaining serializability of global executions within an HDBS. These difficulties result from the difference between serialization orders and execution orders, and the autonomies of the local databases. We also discuss the limitations of the algorithms proposed in the literature and illustrate how they violate serializability and autonomy.

The rest of this paper is organized as follows. In section 2, we present a transaction processing model for HDBSs. The difficulties in designing global concurrency control algorithms and the unsuitability of serializability as the correctness criterion for global concurrency control are discussed in sections 3 and 4. In section 5, a discussion of some of the proposed global concurrency control algorithms is presented in terms of serializability, local autonomy and degree of concurrency. Some concluding remarks are given in section 6.

2 A Transaction Processing Model

The transaction processing model used in this paper is shown in figure 1. It consists of a Global Data Manager (GDM), a Global Transaction Manager (GTM), a collection of local database systems (LDBSs), a set of global and local transac-

tions, and a set of server processes. A server process runs at each site participating in a global transaction and represents the interface between the GTM and the LDBSs.

When a global transaction is submitted to the HDBS, it is decomposed by the GDM into a set of global subtransactions that run at the various sites where the referenced data items reside. These subtransactions are then submitted to the GTM. The GTM, in turn, submits these subtransactions to the LDBSs in such a way that the global database consistency is maintained. By maintaining the global database consistency we mean that global transactions transform the collection of underlying local databases from one consistent state to another consistent state. It is assumed that, for every global transaction, there is at most one subtransaction per site [GL84]. The Global Concurrency Controller (GCC) is a functional unit in the GTM which controls the execution of the global transactions. We assume that every LDBS has a Local Concurrency Controller (LCC) which controls the execution of local transactions and global subtransactions at that site.

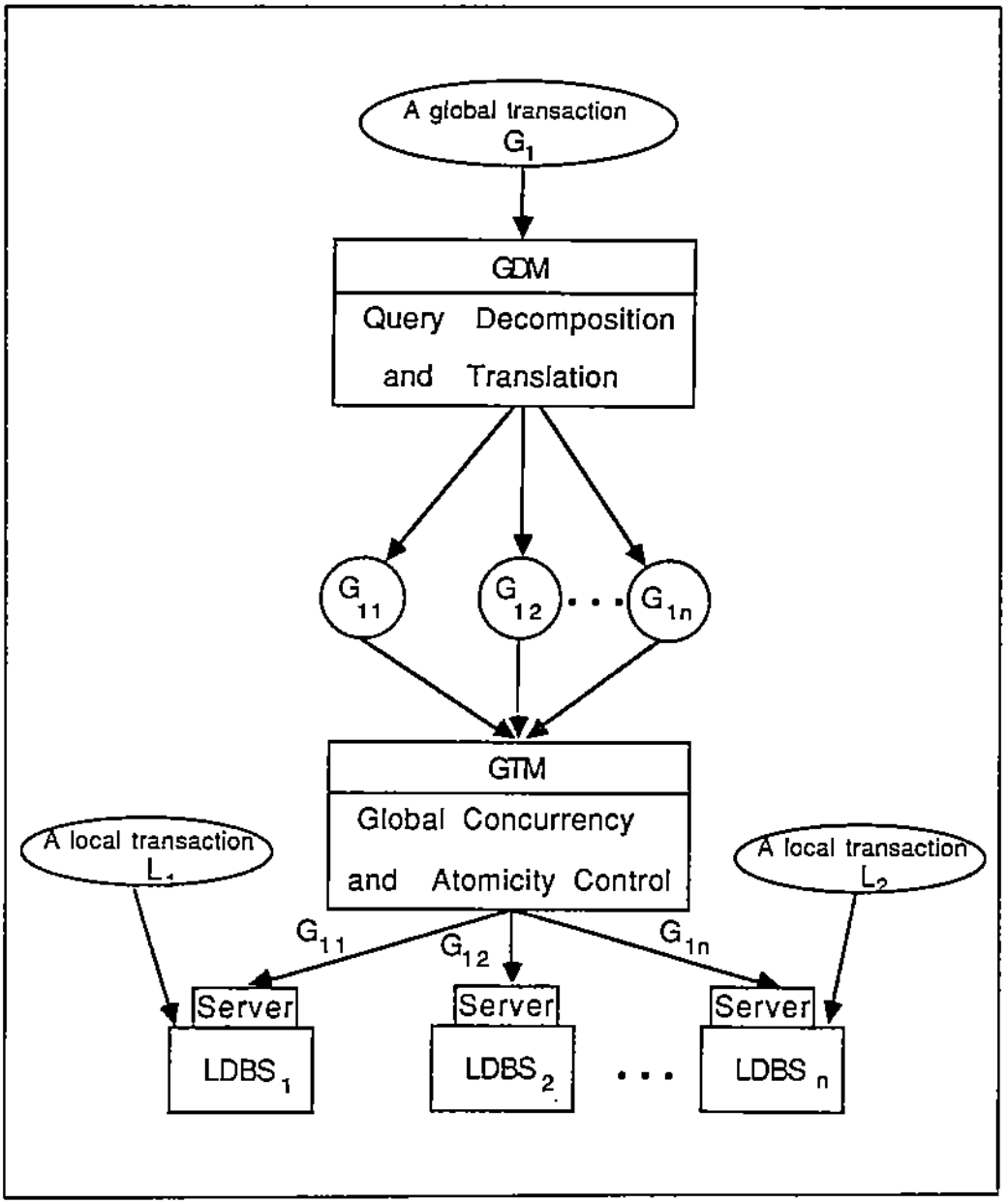

Figure 1 A transaction processing model for HDBSs

3 Global Concurrency Control

One of the main differences between homogeneous and heterogeneous database environments is the existence of local autonomies. In this section, we discuss the effect of these local autonomies on the design of global concurrency control algorithms for HDBSs. We also discuss difficulties of doing global concurrency control under the restrictions of these autonomies, and possible ways of remedying the problem.

3.1 Local Autonomies

Designing a concurrency control strategy for a heterogeneous database environment is different from designing a concurrency control algorithm for a homogeneous (distributed) database system and is much more difficult. The difficulties are primarily because a GCC must deal with heterogeneity and autonomy of underlying databases, in addition to the possibility that data may be distributed among various sites. In a homogeneous environment, there is only one concurrency controller to certify and produce the schedules. The concurrency controller has access to all internal information it needs to produce and/or certify the schedules. In addition, it normally has control over all transactions running in the system. The fact that concurrency control algorithms in traditional systems do not have to deal with the questions of autonomy and heterogeneity makes the problem sufficiently different that new algorithms must be designed for the heterogeneous environment. These difficulties manifest themselves even more gravely when addressing commitment protocols. In this paper, however, discussion will be limited to synchronization atomicity.

The design of a Global Concurrency Controller is complicated by the autonomy of the local databases. The autonomies are defined in terms of design, communication, and execution [EV87], and can be explained in the following manner.

First, local concurrency controllers are designed in such a way that they are totally unaware of other LDBSs or of the integration process. This type of autonomy is defined as design autonomy. It generally indicates that each of the LDBSs

is free to use whatever algorithms it wishes. When this LDBS is integrated into a federation, design autonomy specifies that we cannot retrofit its algorithms. The only way to remedy this problem is to use a hierarchical approach to concurrency control.

Second, the GCC needs information regarding local executions in order to maintain global database consistency. However, the GCC has no direct access to this information, and is unable to force the LCCs to supply it. This type of autonomy is defined as communication autonomy, which means that an LCC is allowed to make independent decisions as to what information to provide.

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

3.2 The Difficulties of Global Concurrency Control

In an HDBS, local and global concurrency control must be addressed separately because of local autonomies. LCCs guarantee the correctness (usually using serializability) of the executions of local transactions and global subtransactions at each local sites. The GCC, on the other hand, is responsible for retaining the consistency of the global database. Before the discussion of the difficulties of retaining the global database consistency, let us first introduce direct and indirect conflicts between operations.

In a concurrent execution, one operation might conflict with another operation in the sense that the effect of one influences the other. The influences can either be on the values read by an operation or on the current database state. Two conflicting operations are said to directly conflict if they both operate on the same data item, and indirectly conflict otherwise. Similarly, we say that two global

transactions directly conflict if they contain directly conflicting operations. If two global transactions are not directly conflicting but contain indirectly conflicting operations, then they indirectly conflict with each other.

Example 3.1: Consider an HDBS consisting of two LDBSs D_1 and D_2, where data item a is at D_1 and b and c are at D_2. The following global transactions are submitted to the HDBS:

$G_1 : w_{g_1}(a) r_{g_1}(c)$

$G_2 : r_{g_2}(a) w_{g_2}(b)$

Let L be the local transaction submitted at D_1:

L: $r_l(b) w_l(c)$

Let E_1 and E_2 be local executions at D_1 and D_2, respectively:

$E_1 : w_{g_1}(a) r_{g_2}(a)$

$E_2 : w_{g_2}(b) r_l(b) w_l(c) r_{g_1}(c)$

In E_1, $r_{g_2}(a)$ read the value written by $w_{g_1}(a)$. They are directly conflicting. In E_2, $w_{g_2}(b)$ and $r_{g_1}(c)$ operate on different data items. However, the effect of $w_{g_2}(b)$ might be propagated, through operations of L, to $r_{g_1}(c)$. So they are indirectly conflicting. □

Direct conflicts among global operations involve global operations only, and therefore can be easily predicted by a GCC. Indirect conflicts among global operations, however, might be introduced by local operations. This is illustrated in the previous example. To maintain global database consistency, a GCC has to be able to detect and resolve both direct and indirect conflicts among global operations properly. To do this, the GCC needs to access some information about local executions. However, this is usually very difficult (if not impossible) for the following reasons.

1. Some LDBSs may not have the information the GCC wants. For example, a GCC using a two phase locking protocol would like to coordinate the lock points of all global transactions with conflicting operations. An LCC

which uses a timestamp ordering protocol, however, does not usually have any information about lock points. This problem is caused by the design autonomy of LDBSs.

2. Even if all LDBSs have the right information, communication autonomy may still prevent the GCC from accessing it.

3. Assuming that a GCC has gotten all the information it wants, and has detected some conflicts, how can it resolve these conflicts? It has to block or restart some transactions. This, unfortunately, will violate the execution autonomy of the LDBSs

HDBSs have various requirements of local autonomy. It is likely that in many special environments, including most practical environments, global concurrency control is much easier than we have described above. Generally speaking, however, global concurrency control in heterogeneous database environments is very difficult.

3.3 General Approaches to Global Concurrency Control

To maintain global database consistency, a GCC can take one of the following approaches:

Approach 1 *It can assume that there are indirect conflicts among global operations wherever such conflicts could possibly exist.* This approach is based on the assumption that not every pair of global operations could conflict arbitrarily. Since LCCs guarantee the consistency of LDBSs, global operations can only interleave in a restricted way. In addition, the type of global operation can be used by a GCC when assuming conflicts. For example, it might be reasonable to assume that two read operations have a lesser chance of conflicting than two write operations in certain environments. The basic idea of this approach is to do concurrency control posteriorly, not in advance. After global transactions have been executed, the GCC tries to find, hopefully with some help from the LCCs, the possible conflicts among

global operations. If an undesirable situation arises, then the GCC tries to resolve it (usually by aborting global transactions). One advantage of this approach is the potentially high degree of concurrency. Another advantage is that it is possible for a GCC to make use of local information whenever it is available. The main disadvantage of this approach, however, is the potentially poor performance due to its having to abort global transactions. This can be especially bothersome when the rate of correct conflict prediction is low.

Approach 2 *It can impose restrictions on global transactions such that undesirable indirect conflicts cannot occur.* This approach, unlike the first one, does global concurrency control before submitting global transactions, therefore preventing undesirable situations from occurring. One way to do this is for a GCC to use a restricted transaction model for global transactions so that most undesirable situations are impossible. The other way is to control the submission of global transactions so that they can only be properly interleaved. For example, two global transactions with possibly conflicting operations should not be allowed to executed concurrently. GCCs using this approach will not abort global transactions due to improper concurrent execution. This will generally result in better performance than approache 1 because less global transactions will be aborted. On the other hand, this approach allows a lower degree of concurrency. This problem, however, may be alleviated if some knowledge of the LCCs is used. Another problem of this approach is that GCCs are usually unable to make use of information about local executions because it is impossible to predict in advance whether this information will be available.

Some global concurrency control protocols using the above approaches will be given in section 5.

We did not mention how well these approaches work. Actually, it is not even clear to us whether the goal of designing a global concurrency controller that provides a high concurrency degree while maintaining local autonomy is attainable. The solution to this problem depends on the correctness criterion used for con-

currency control. In the next section, we discuss this problem for serializability.

4 Maintaining Serializability in HDBSs

In this section, we discuss the difficulties of maintaining global serializability in HDBSs. We start by distinguishing serialization order from execution order among operations. It is this difference that makes the problem of maintaining global serializability much more difficult. We will then discuss the problems with the two approaches mentioned previously.

Serializability has been introduced as a (sufficient) correctness condition for concurrency control in homogeneous (centralized) database environments. A concurrent execution is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions.

Serializability has been extended to the distributed environment [BG81]. A global execution is serializable if there exists a globally serial execution of the same transactions such that they produce the same output and have the same effect on the global database. It has been proven that a global execution is serializable if and only if its global serialization graph is acyclic [CO82]. The same idea has also been applied to heterogeneous database environments [BS88b].

4.1 Serialization Order

A serialization order may exist between two global operations. If so, it indicates that, in any serial executions which are equivalent to the original execution, the transactions containing the two global operation are in the given serialization order. Therefore, a global execution is serializable if there exists a total order of global transactions which is compatible with all serialization orders among global operations.

Given a local execution, let o_1 and o_2 be two operations of the execution. The execution order is, by its name, the order that each operation is executed. Specifically, we say that $o_1 \rightarrow_e o_2$ if o_1 is executed before o_2. Clearly, the $\rightarrow_e$

relation holds between any two operations, making it a total order.

Serialization order, on the other hand, is a partial order of all operations in the executions. Basically, there are two kinds of serialization orders: direct and indirect serialization orders, corresponding to direct and indirect conflicts. We denote a direct serialization order from o_1 to o_2 as $o_1 \rightarrow_d o_2$ and an indirect serialization order as $o_1 \rightarrow_i o_2$. A direct serialization order holds between o_1 and o_2 if they both operate on the same data item, at least one of them is a write operation, and o_1 is executed before o_2. An indirect serialization order from o_1 to o_2 holds if there exist two operations o_3 and o_4 of another transaction such that $o_1 \rightarrow_d o_3$ and $o_4 \rightarrow_d o_2$. The serialization order is simply the transitive closure of direct and indirect serialization order.

It is worth noting that the serialization order between two operations might be different than their execution order. In example 3.1, the indirect serialization order of $r_{g_1}(c)$ and $w_{g_2}(b)$ in E_2 would be unchanged even if their execution order had been changed from:

$$w_{g_2}(b) r_l(b) w_l(c) r_{g_1}(c)$$

to:

$$w_l(c) r_{g_1}(c) w_{g_2}(b) r_l(b).$$

It is this difference that makes global concurrency control very difficult.

4.2 Assuming Indirect Serialization Orders

To maintain global serializability, a GCC needs to coordinate the serialization orders among global operations at each local site. Although direct serialization orders which are the same as their execution orders are predictable at global level, indirect serialization orders are usually not predictable at the global level.

One way to anticipate is to be pessimistic. In other words, we assume that there exists an indirect serialization order between each pair of global operations that operate on data items at the same site (but are not directly conflicting with each other) [BS88b]. Since all possibilities of serialization orders among

global operations (both direct and indirect) have been considered, the global serializability is therefore guaranteed. The price is, however, low concurrency degree.

On the other hand, it seems very unlikely that indirect serialization orders can be anticipated more precisely at the global level. The reasons for this are as follows. First, it is impossible to anticipate according to the types of two global operations. For any two global operations, there can always exist local executions which introduce indirect serialization order between them. Second, it is also impossible to anticipate indirect serialization order according to the execution order of two global operations because the serialization order might be different from the execution order.

Example 3.1 is an example of the first reason. Global operations $r_{g_1}(c)$ does not directly conflict with $w_{g_2}(b)$ in E_2. They both operate on data items at D_2. However, there is an indirect serialization order between them because of the presence of local transaction L. If, on the other hand, the local transaction L is absent, there would be no serialization order between them at that site.

The fact that indirect serialization orders cannot be anticipated makes it very hard for a GCC to maintain, in general, the serializability of global executions.

4.3 Imposing Restrictions on Global Transactions

Another approach that a GCC can adopt to enforce global serializability is to impose restrictions on global transactions. Restrictions on operation types of global transactions (e.g. read only) and restrictions on the ways that global transactions are submitted (e.g. serially) are common ones.

The following example shows that even if a GCC submits global transactions serially at global level, the global serializability is still not guaranteed.

Example 4.1: Consider an HDBS consisting of two LDBSs D_1 and D_2, where data item a is at D_1 and b and c are at D_2. The following global transactions are submitted to the HDBS:

$G_1 : r_{g_1}(a) w_{g_1}(b)$

$G_2 : w_{g_2}(a) r_{g_2}(c)$

Let L be the local transaction which is submitted at D_1:

L: $w_l(b) w_l(c)$

Let E_1 and E_2 be local executions at D_1 and D_2, respectively:

$E_1 : r_{g_1}(a) w_{g_2}(a)$

$E_2 : w_l(b) w_{g_1}(b) r_{g_2}(c) w_l(c)$

The operations of transaction G_1 precede those of transaction G_2 at both sites. The execution could have been produced by the GCC submitting transactions G_1 and G_2 serially, however, the global execution is not serializable. □

In this example, local transactions introduce an indirect serialization order (at D_2) between two global operations which is different from their execution order. Although there is no direct serialization order between $w_{g_1}(b)$ and $r_{g_2}(c)$, there do exists direct serialization order between $w_l(b)$ and $w_{g_1}(b)$ and direct serialization order between $r_{g_2}(c)$ and $w_l(c)$. These two direct serialization orders imply $r_{g_2}(c) \rightarrow_i w_{g_1}(b)$. However, the execution order between $w_{g_1}(b)$ and $r_{g_2}(c)$ was $w_{g_1}(b) \rightarrow_e r_{g_2}(c)$. The difference between serialization order and execution order makes it impossible, in general, for a GCC to control the serialization orders of global operations by simply controlling the submission of these global transactions at global level.

In some special cases, however, serialization orders of operations are compatible with their execution order. It is therefore possible for a GCC to maintain a certain serialization order by controlling the execution order of global operations. One example is a system in which all LCCs use two phase locking as the concurrency control protocol at local sites. In this special environment, once a local transaction gets a lock on a data item, it will not release it until it gets all the locks it wants. In this case, it would not be possible for global subtransactions to get locks held by a local transaction who has not reached its lock point yet. This would have prevented G_1 from getting the lock on data item b until L had completed both of its operations. Therefore, in this case, serial submission of

global transactions is sufficient to guarantee the global serializability [1].

Another restriction that a GCC can impose on global transactions is the type of operations. A natural restriction is to disallow write operations in global transactions. However, as the following example shows, this restriction alone is still not sufficient to guarantee the global serializability.

Example 4.2: Consider an HDBS consisting of two LDBSs D_1 and D_2, where data item a is at D_1 and b and c are at D_2. The following global transactions are submitted to the HDBS:

$G_1 : r_{g_1}(a) r_{g_1}(b)$

$G_2 : r_{g_2}(a) r_{g_2}(c)$

Let L_1 and L_2 be two local transactions submitted at D_1 and D_2, respectively:

$L_1 : w_{l_1}(a) \qquad L_2 : w_{l_2}(b) w_{l_2}(c)$

Let E_1 and E_2 be local executions at D_1 and D_2, respectively:

$E_1 : r_{g_1}(a) w_{l_1}(a) r_{g_2}(a)$

$E_2 : w_{l_2}(b) r_{g_1}(b) r_{g_2}(c) w_{l_2}(c)$

Clearly, the execution is not globally serializable. □

Since these global transactions are all read only and the local executions are serializable, there is really nothing wrong with the executions of the local transactions (because the global transactions had no effect on the data items). The final global database state is correct too. The only problem is that the global transactions could not have read the same values in a serial execution of the operations. This problem is very difficult to solve simply because a GCC has no control over local executions, although it is not obvious that it is an important problem.

Again, certain restrictions on LCCs are helpful. For example, if LCCs generate strict serializable schedules only, two restrictions (read only global transactions

[1] Actually, it has been proven that global serializability is guaranteed if a GCC also uses two phase locking as global concurrency control protocol [BS88a]. That is, a global transaction will not release any locks until it gets all locks it wants at all sites.

and serial execution) will guarantee the serializability of global executions.

In summary, if a high concurrency degree is not required or we are working in some special environment where LCCs use specific concurrency control protocols, it is possible to maintain the serializability of global execution. Otherwise, serializability is very hard to achieve at the global level. Global concurrency control can only indirectly influence local execution through global transactions. This is not sufficient to maintain global database consistency, especially when serializability is used as correctness condition.

5 Discussion of the proposed algorithms

Several global concurrency control protocols have been proposed. In this section, we try to analyze them from the following points of view [2]:

1. global serializability,
2. concurrency degree,
3. local autonomy, and
4. concurrency control approaches.

Breitbart proposed a global concurrency control protocol based on site graphs [BS88b]. The protocol works as follows. Before a global transaction is submitted, the GCC analyzes its read and write operations trying to select some sites to execute them without creating cycles in the site graph. The acyclicity of the site graph will guarantee the correctness (serializability) of the global execution. This algorithm is a good example of approach 2. It preserves local autonomy, and does not abort any global transactions. The problem with this algorithm, however, is that it allows a low concurrency degree for global transactions. One observation is that a global transaction with operations at all sites blocks the execution of

[2] The discussion in this section is based on the general HDBS model. It is possible that these discussions may not hold if the algorithms are run in some special environments.

other global transactions until it is committed. Another observation is that no two global transactions can access multiple sites concurrently.

Another example of approach 2 is the altruistic locking algorithm proposed by Alonso [AGS87] [SGA87]. This algorithm, by its name, is a lock based algorithm. The locking granularity is that of a local site. In other words, a local site can execute at most one global subtransaction (and many local transactions) at a time. In order to access a local database, the global procedure has to lock the local site and then issue a global subtransaction to the Local Transaction Manager of that site. The locking or releasing of local sites must follow specific rules [SGA87]. This algorithm has the following two problems: 1) Since the granularity is a site, the degree of concurrency for the global procedures is very low. 2) Since the effect of the local transactions is not considered, the global serializability is not guaranteed. This is illustrated by example 5.1.

Example 5.1: Consider a HDBS consisting of two LDBSs, where data items a and b are at site 1, and c and d are at site 2. The following global procedures are submitted to the HDBS:

G_1: $r_{g_1}(a)\ w_{g_1}(c)$

G_2: $w_{g_2}(b)\ r_{g_2}(d)$

Let L_1 , L_2 be the local transactions submitted at D_1 and D_2, respectively:

L_1: $w_{l_1}(a)\ r_{g_2}(b)$

L_2: $r_{l_2}(c)\ w_{l_2}(d)$

Let E_1 and E_2 be the local executions at D_1 and D_2, respectively:

E_1: $w_{l_1}(a) r_{g_1}(a) w_{g_2}(b) r_{l_1}(b)$

E_2: $w_{g_1}(c) r_{l_2}(c) w_{l_2}(d) r_{g_2}(d)$

As far as the global procedures are concerned, in E_1, G_1 locks D_1, reads data item a and releases D_1. Then G_2 locks D_1, writes data item b and then releases D_1. In E_2, G_1 locks D_2, reads data item c and releases D_2, then G_2 locks D_2, writes data item d and releases D_2. This execution is allowed in the proposed algorithm, but it is not a serializable execution. □

Unlike the previous two protocols, the optimistic algorithm proposed by Elmagarmid [EH88] is an example of approach 1. It is based on a centralized controller and uses operating system robust processes (STUBs). One of the key ideas is the STUB process. It ensures the successful execution of a global subtransaction even through it may get aborted or restarted repeatedly by the local concurrency controller. This algorithm does not violate local autonomy. However, because of the lack of consideration for local transactions, this algorithm may generate a non-serializable schedule, as shown in example 4.2.

The global concurrency control algorithms used by Pu in his superdatabases gives another example of approach 1 [Pu88]. The basic idea behind his algorithm is as follows. Every LDBS reports to the GCC the serialization order, o-element, of each global subtransaction executed on it. The GCC uses these o-elements to construct an o-vector for each global transaction. It then validates the execution of a global transaction against the set of recently committed global transactions. It does this by trying to find a consistent o-vector position among the o-vectors of the recently committed global transactions for the global transactions attempting to commit. This approach is good for the hierarchical composition of HDBSs and also provides a high degree of concurrency. However, it is not clear to us how the o-element could be defined in general, although it is obvious for those LDBSs that use two phase locking or time stamp ordering protocols for concurrency control. It is also unclear how the GCC could get these o-elements in an autonomous environment.

The last algorithm we discuss in this section is the distributed cycle detection algorithm proposed by Sugihara [Sug87]. In his algorithm, each local site keeps a local serialization graph for the transactions executed on it. The local serialization graph is kept acyclic by the local concurrency controller. The GCC validates the execution of a global transaction by invoking a distributed cycle detection algorithm to make sure that the commitment of this global transaction will not create a global cycle among the local serialization graphes. The algorithm allows high concurrency degree for the global transactions, but it has the following drawbacks: 1) It violates local autonomy by requiring the local sites to keep the local serialization graphes for the GCC. 2) In order to validate the execution of the

local transaction for the global serializability, the LCC is required to invoke the distributed cycle detection algorithm for the commitment of the local transaction. This violates execution autonomy.

In summary, all of the protocols discussed above either violate local autonomy in a certain way, allow very low concurrency degree, or fail to maintain global serializability. The main reason, we think, is the contradiction between global serializability, which requires help from the LCCs, and local autonomy which may prevent this help from being given.

6 Conclusion

In this paper, we have studied the effects of local autonomy on global concurrency control in HDBSs. We have also studied the difficulties and general approaches of global concurrency control in HDBSs. All of the difficulties result from local autonomy, which differentiates HDBSs from homogeneous distributed database systems. Because of local autonomy, concurrency control, which is centralized in nature, must be done at the global and local levels separately. A GCC, which is responsible for global database consistency, does not have enough information to do the job.

In particular, we have studied the difficulties of maintaining global serializability in HDBSs. We discussed this problem both in general by distinguishing serialization orders from execution orders, and in specific by analyzing several proposed global concurrency control algorithms. It is our claim that it is impossible, in general, to design a good global concurrency control algorithm which has a high degree of concurrency and does not violate local autonomy as long as serializability is used as the correctness criterion.

The unsuitability of serializability to some special database environments such as CAD databases and HDBSs have already been observed. The problem may be attacked in the following ways:

1. Extending serializability in some special environments whenever it is possible. This requires a good knowledge of the meaning of consistency in these

environments. Several attempts have already been made [KS88] [DE88]. However, further work is still needed in this area, especially for HDBSs.

2. Using a new concurrency control paradigm. For example, in HDBS environments, global concurrency control can be done very easily if serialization order of (global) transactions is pre-specified at the global level and then enforced at all local sites. This approach is interesting because it will result in efficient global concurrency controllers. It is also feasible in HDBS environments. The specific serialization order can be enforced at local sites either by upgrading the existing local concurrency controller (if possible), or by controlling of submission of subtransactions. The implementation detail is out of the scope of this paper and will be reported elsewhere.

References

[AGS87] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency control and recovery for global procedures in federated database systems. In *IEEE Data Engineering*, pages 5–11, September 1987.

[BG81] P. Bernstein and N. Goodman. Concurrency control in distributed systems. *ACM Computing Surveys*, 2(12), 1981.

[BS88a] Y. Breitbart and A. Silberschatz. Multibdatabase systems with a decentralized concurrency control scheme. *Distributed Processing Technical Committee Newsletter*, 10(2):35–41, November 1988.

[BS88b] Y. Breitbart and A. Silberschatz. Multibdatabase update issues. In *Proceeding of the International Conference On Management of Data*, pages 135–142, June 1988.

[BST87] Y. Breitbart, A. Silberschatz, and G. Thompson. An update mechanism for multidatabase systems. *IEEE Data Engineering*, 10(3):12–18, September 1987.

[CO82] S. Ceri and S. Owicki. On the use of optimistic methods for concurrency control in distributed database systems. In *Proc. 6th Berkely Workshop*

Distributed Data Management and Computer Networks, pages 117–129, 1982.

[DE88] W. Du and A. Elmagarmid. *QSR: A Correctness Criterion for Global Concurrency Control in InterBase.* Technical Report CSD-TR-836, Purdue University, December 1988.

[EH88] A.K. Elmagarmid and A.A. Helal. Supporting updates in heterogeneous distributed database systems. In *Proceedings of the International Conference on Data Engineering*, pages 564–569, 1988.

[EV87] F. Eliassen and J. Veijalainen. Language support for mutidatabase transactions in a cooperative, autonomous environment. In *TENCON '87, IEEE Regional Conference*, Seoul, 1987.

[GL84] V.D. Gligor and G.L. Luckenbaugh. Interconnecting heterogeneous data base management systems. *IEEE Computer*, 17(1):33–43, January 1984.

[GP85] V.D. Gligor and R. Popescu-Zeletin. Concurrency control issues in distributed heterogeneous database management systems. In F.A. Schreiber and W. Litwin, editors, *Distributed Data Sharing Systems*, North-Holland, 1985.

[KS88] H. K. Korth and G. D. Speegle. Formal model of correctness without serializability. In *Proceeding of the International Conference On Management of Data*, pages 379–386, June 1988.

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proceeding of the International Conference On Data Engineering*, pages 548–555, February 1988.

[SGA87] K. Salem, H. Garcia-Molina, and R. Alonso. *Altruistic Locking: A strategy for coping with long-live transaction.* Technical Report CS-TR-087-87, Princeton University, April 1987.

[Sug87] K. Sugihara. Concurrency control based on cycle detection. In *Proceeding of the International Conference On Data Engineering*, pages 267–274, February 1987.