1989

A Hierarchical Approach to Concurrency Control for Multidatabases

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

Yunhho Leu

Report Number:
89-919

A HIERARCHICAL APPROACH TO CONCURRENCY CONTROL FOR MULTIDATABASES

Ahmed K. Elmagarmid
Yungho Leu

CSD-TR 919
October 1989

A Hierarchical Approach to Concurrency Control for Multidatabases[1]

Ahmed K. Elmagarmid and Yungho Leu
Computer Science Department
Purdue University
West Lafayette, IN 47907
(317)-494-1998
ahmed@cs.purdue.edu

[1]This work is supported by a PYI Award from NSF under grant IRI-8857952 and grants from AT&T Foundation, Tektronix, and Mobil Oil.

Abstract

A multidatabase system is a facility that allows access to data stored in multiple autonomous and possibly heterogeneous database systems. In order to support atomic updates across multiple database systems, a global concurrency control algorithm is required. A hierarchical concurrency control approach has been proposed for multidatabase systems. However, to apply this approach, some restrictions have to be imposed on the local concurrency control algorithms. In this paper, we identify these restrictions, formalize the hierarchical concurrency control approach and prove its correctness. A new global concurrency control algorithm based on this hierarchical approach is presented.

1 Introduction

A Multidatabase System (MDBS) is a facility that allows access to data stored in multiple autonomous and possibly heterogeneous database systems. An MDBS is characterized by strong *autonomy* requirements [DELO89] [GK88] [EV87] of its local database systems implying that operations of each local database system must be unaffected by the MDBS facility. Moreover, each local database systems is allowed to leave or join an MDBS without any reprogramming or loss of data consistency in the MDBS.

Global concurrency control is required in order to allow concurrent global updates in an MDBS. A general hierarchical approach to concurrency control has been proposed for the autonomous database environment [GP86]. Many global concurrency control algorithms have been recently proposed based on this general approach [Pu88] [AGS87] [Vid87]. However, this general approach is not suitable for all MDBS environments. In this paper, we will concentrate on examining the local concurrency control restrictions under which the hierarchical approach is applicable. A new global concurrency control algorithm based on this general approach is also proposed in this paper.

The rest of this paper is organized as follows. In section 2, a transaction model is introduced, global serializability is defined and an example showing the effect of autonomy on global concurrency control is given. In section 3, the general hierarchical approach is discussed and a static property is defined. Finally, the general hierarchical approach is formalized and its correctness is proved. In section 4, a new global concurrency control algorithm is presented. In section 5, existing global concurrency control algorithms based on the hierarchical approach are surveyed. A summary of this paper is given in section 6.

2 Background

2.1 A Transaction Model for MDBS

As shown in figure 1, an MDBS is composed of a set of pre-existing local database management systems, a set of global transactions, a set of local transactions, a set of Local Transaction Managers (LTMs) and a Global Transaction Manager (GTM). A local transaction is a transaction which is issued directly to one of the local database systems. A global transaction consists of a set of subtransactions, each of which accesses one local database system on behalf of the global transaction. It is assumed that each global transaction can have at most one subtransaction per local database system (This assumption simplifies the concurrency control problem). The LTM controls the execution of local transactions and global subtransactions in the local database system, while the GTM controls the execution of the global transactions at the global level. The LTMs and the GTM as a whole are responsible for transaction management in the MDBS. Transaction management includes the concurrency control, commitment control and recovery control. Concurrency control is performed by the Local Concurrency Controller (LCC) and the Global Concurrency Controller (GCC) collectively.

2.2 Global Serializability

A transaction contains a set of read and write operations. Two operations are said to be conflicting if (1) they belong to two different transactions, (2) they access the same data item and (3) at least one of them is a write operation. A concurrency control algorithm controls the execution order of the conflicting operations such that the serializability property of the history is maintained. Serializability is used as the correctness criterion for the MDBS in this paper, therefore, an LCC has to maintain the serializability

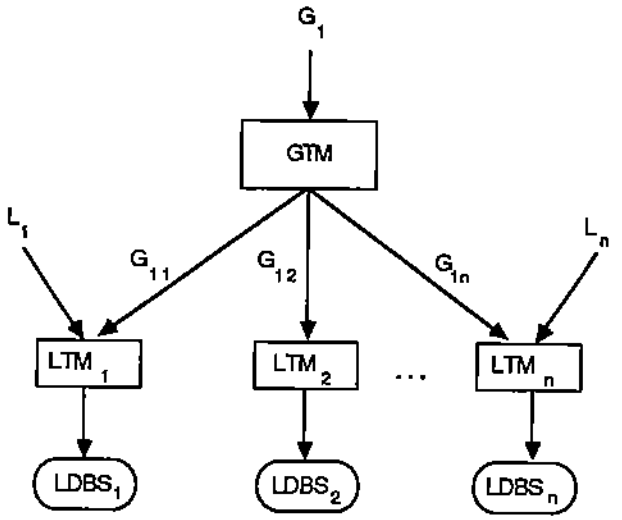

Figure 1: A Transaction Model for MDBS

for the execution of local transactions and global subtransactions. While the LCCs and GCC, as a whole, have to maintain the serializability of global and local transactions.

An execution of a set of transactions can be described by a history [BG85]. A history contains the read and write operations of the involved transactions and, especially, their order of execution. The execution of the local transactions and the global subtransactions at $LDBS_i$ constitutes the *local history* h_i. Let $\mathcal{G}$ denote the set of all global transactions and $\mathcal{L}$ denote the set of all local transactions at all local database systems. A *global history* $\mathcal{H}$ over $\mathcal{G} \cup \mathcal{L}$ is the set of all local histories, i.e. $\mathcal{H} = \{h_1, h_2, ..., h_n\}$ (assuming that there are n LDBSs). The global serializability of a global history is defined as follows:

Lemma 2.1 *Let $\mathcal{T}$ be the set of all global transactions and local transactions (i.e. $\mathcal{T} = \mathcal{G} \cup \mathcal{L}$). The execution of the set of transactions in $\mathcal{T}$ is serializable if there exists a total ordering on the transactions in $\mathcal{T}$ such that for each pair*

of conflicting operations o_i and o_j from transaction T_i and T_j respectively, o_i precedes o_j in any history $h_1, h_2, ..., h_n$ if and only if T_i precedes T_j in the total ordering.

An easy way of checking the serializability of a history is to use the serialization graph. Global serialization graphs in an MDBS environment are defined as follows:

Definition 2.1 *A* global serialization graph *(GSG) of a global history $\mathcal{H}$ is a directed graph, whose nodes are global and/or local transactions (i.e. transactions in $\mathcal{T}$). $\forall\ T_i, T_j \in \mathcal{T}$ there is an arc from T_i to T_j if there are conflicting operations o_i and o_j in T_i and T_j respectively such that o_i precedes o_j in $\mathcal{H}$.*

The *global serializability theorem* can then be stated as follows:

Lemma 2.2 (The Global Serializability Theorem) *If the global serialization graph of a global history $\mathcal{H}$ is acyclic then $\mathcal{H}$ is serializable.*

The proof of this theorem is similar to the serializability theorem in centralized database systems (see [BG85]) and is not given in this paper.

2.3 Autonomy

The difficulty in doing the global concurrency control in MDBSs is mainly due to *local autonomy*. The autonomy requirement significantly aggravates the concurrency control problem in an MDBS. The local autonomy requirement entitles the local database systems to refuse to supply local information to the GCC. Therefore, the GCC has to perform its concurrency control with incomplete local information. The difficulty of doing global concurrency control with incomplete local information can be illustrated by the following example.

Example 1: Consider an MDBS with two local databases $LDBS_1$ and $LDBS_2$. Where data item a is at $LDBS_1$ and b and c are at $LDBS_2$. The following global transactions are submitted to the MDBS:

G_1: $w_{g1}(a)r_{g1}(b)$

G_2: $w_{g2}(a)r_{g2}(c)$

Let L be a local transaction submitted to the $LDBS_2$:

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

Let h_1 and h_2 be local histories at $LDBS_1$ and $LDBS_2$, respectively:

$$\begin{array}{llllll} h_1: & w_{g1}(a) & w_{g2}(a) & & & \\ h_2: & w_l(b) & r_{g1}(b) & r_{g2}(c) & w_l(c) & \end{array}$$

$$t_0 \qquad t_1 \qquad t_2 \qquad t_3$$

$\longrightarrow$ time

A scenario of the activities of the GCC and LCCs is as follows: Assuming that at time t_2, $w_{g1}(a)$, $w_{g2}(a)$, $w_l(b)$ and $r_{g1}(b)$ have been executed in the order as shown in the local histories. Let's further assume that G_1 has been committed and G_2 has just finished its last operation which is r_{g2}; the global history at time t_2 is serializable so that GCC chooses to commit the global transaction G_2. At time t_3 the local transaction L issues its last operation $w_l(c)$, then the LCC at $LDBS_2$ commits L because the local history h_2 is serializable at that time. The GSG of this execution (Figure 2) contains a cycle. In other words, the global history is not serializable.

The problem in the above execution is that when GCC made its decision to commit G_1 and G_2, it did not know that the operation $w_l(c)$ will be processed later (this is due to the local autonomy). Local transaction L introduces an *invisible indirect order* (can not be seen by the GCC) between subtransactions G_1 and G_2. This determines the serialization order between G_1 and G_2 after G_1 and G_2 are committed.

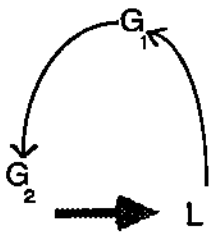

Figure 2: The GSG of a simple execution

3 The Hierarchical Approach

3.1 The General Approach and Restrictions

As suggested by Gligor and Luckenbaug [GL84], one way to design global concurrency control algorithms without violating local autonomy is to impose a control hierarchy between the GCC and the LCC. The idea of the hierarchical approach is that each LCC controls the execution of the local transactions and subtransactions on its local database system to ensure the serializability of their execution; while the GCC controls the execution of the subtransaction so that the serialization orders (see [BG85]) of the subtransactions are compatible in all local database systems. Many proposed global concurrency control algorithms [Pu88] [Vid87] [AGS87] are based on this approach. A major part of this paper is concerned with studying the required restrictions the local concurrency control algorithms so that a hierarchical approach can be imposed. We identified those concurrency control algorithms to which the hierarchical concurrency control approach is applicable. We call this class of concurrency control algorithms *static* concurrency control algorithms. Before we define static concurrency control algorithms, we introduce the following notations. The lifetime LT_i of a transaction T_i

is the time period after a transaction starts its execution and before it terminates (either commits or aborts). A domain set $\mathcal{D}$ is a total order set. In other words, there is an irreflexive, transitive binary relation $<$ on $\mathcal{D}$ such that for any a,$b \in \mathcal{D}$, either $a < b$ or $b < a$. If $a < b$ we say that a *precedes* b and vice versa. A concurrency control algorithm is said to be *static* if it has the following properties:

1. For every successfully executed transaction T_i, there exist a corresponding $d_i \in \mathcal{D}$, such that
2. For any two different transactions T_i and T_j, $d_i \neq d_j$;
3. $\exists$ a timestamp $t_i \in LT_i$ such that d_i will be determined when time t_i is reached; and
4. If T_i conflicts with T_j and is serialized before[1] T_j, then $d_i < d_j$.

The set $\mathcal{D}$ is called the *order domain*; and d_i is called the *serialization order* of transaction T_i. The time t_i when the transaction T_i is serialized is called the *serialization point* of T_i. The order set $\mathcal{D}$ can be any countable or uncountable set, if only a total order can be defined on it.

3.2 The Static Properties of the Existing Concurrency Control Algorithms

Most of the existing concurrency control algorithms are static. Moreover, most of their order domain are the same as the time domain. For example, in the two phase locking algorithm [EGLT76], locking is used to resolve the execution order of the conflicting operations. The time when a transaction acquires the locks for all data items it needs is called the *lock point* of the transaction. Lock point can be used as a serialization order of a transaction. Since for any two transactions T_i and T_j, if T_i reaches its serialization

[1] For a serializable history, T_i is said to be serialized before T_j if T_i conflicts with T_j and the conflicting operations of T_i are executed before those of T_j.

point before T_j, then T_i must be serialized before T_j. Moreover, the lock point occurs within the lifetime of the transaction. In other words, the two phase locking algorithm is static. Furthermore, the serialization point and serialization order of a transaction are the same in a two phase locking algorithm.

For timestamping concurrency control algorithm [BHG87], a timestamp assigned at the beginning of a transaction is used to resolve the conflicts. All conflicting operations must be executed according to their corresponding transaction timestamps. Since the transactions are serialized by their timestamps, the timestamp of a transaction can be used as its serialization order. This order is determined at the beginning of the transaction which is within the lifetime of the transaction. Therefore, the timestamping algorithm is static.

For optimistic concurrency control algorithm[KR81], a transaction number is assigned to a transaction at the end of its read phase. The transaction is then validated using this number. It can be reasoned that transactions are serialized according to their transaction numbers. Therefore, the transaction number can be used as its serialization order. Furthermore, a transaction number is determined at the end of a transaction read phase which occurs within the lifetime of the transaction. Due to these observations, we conclude that the optimistic concurrency control algorithm is static.

A newly developed concurrency control algorithm is the *value dates* concurrency control algorithm [LT88]. In the value dates concurrency control algorithm, a transaction must specify the time (or date) when it is going to finish. The algorithm then uses these value dates to resolve conflicts. If a data item is not available to a transaction (the operation of the transaction is late), then the transaction can either be undone or access other data item if possible. Since only the transaction with later value date can be scheduled to wait until the other terminates, the algorithm will serialize the transactions according to their value dates. Therefore, the value date

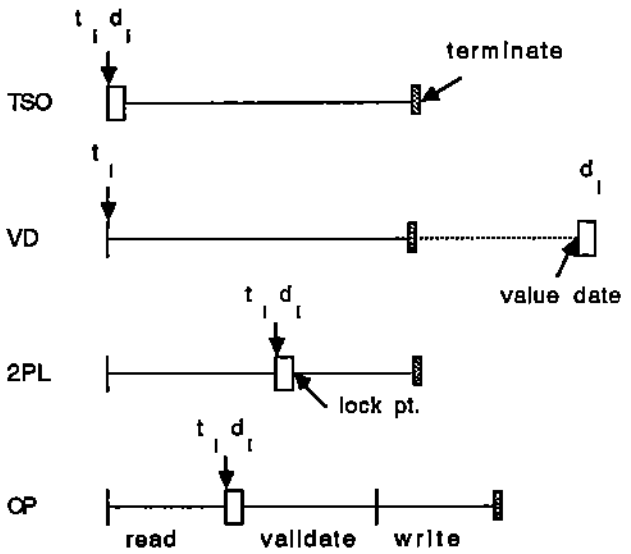

Figure 3: The serialization point and the serialization order

can be used as the serialization order. It is obvious that the order domain of the value dates algorithm is the same as the time domain. It is noted that the value date is determined within the lifetime of the transaction. Based on discussions, the value dates algorithm is static.

The occurrences of the serialization point and the serialization order for the timestamping (TSO), value date (VD), two phase locking (2PL) and the optimistic (OP) concurrency control algorithms are shown in figure 3. In figure 3, the serialization order is shown only for those concurrency control algorithms whose order domain is the time domain. It is interesting to see that for the TSO, 2PL and OP algorithms, the serialization point and the serialization have the same ordering. This means that for these algorithms, if transaction T_i is serialized before T_j, then the serialization order of T_i must precede the serialization order of T_j. This feature is very useful for designing a global concurrency control algorithm. A global concurrency control algorithm which uses this feature is presented in a section 4.

Not all of the existing concurrency control algorithms are static. For example, the *serialization graph testing* algorithm [Cas81] is not static. Even though the serialization graph testing algorithm maintains the serializability property of the transaction execution, there is no specific time for which the serialization order of a transaction can be determined. The serialization order of a transaction is affected by the execution of the concurrent transactions in such a way that it may not be determined.

3.3 The Importance of the Static Properties on Global Concurrency Control

Two important static properties are

- Serialization order is determined within the lifetime of a transaction; and,
- The serialization order is a total order.

The first property enables us to obtain the serialization order without violating local autonomy. If a transaction is not serialized in its lifetime, then other transactions must be consulted in order to determine the serialization order of it. Among them there may be some local transactions. To access information about local transaction is a violation of local autonomy. The second property is needed to prevent the invisible indirect serialization order between subtransactions introduced by the local transactions. One of the invisible indirect serialization order is shown in example 1. The invisible indirect serialization order will not occur if the concurrency control algorithm is static. This can be argued by using example 1. Consider local history h_2. If the local concurrency control algorithm in $LDBS_2$ is static, then the orders of G_1 and G_2 at $LDBS_2$ must be determined when both the G_1 and G_2 finish. If G_1 precedes G_2, then there is no valid serialization order for L. since according to the conflicts, serialization order of L must precede G_1

and be after G_2, which is not possible. In this case L will be aborted by the LCC. If the serialization order of G_2 precedes the serialization order of G_1, then the indirect serialization order introduced by L is the same as the order between G_1 and G_2; therefore is visible to the GCC.

3.4 Correctness of the hierarchical approach

Under the assumption that local concurrency control algorithms are static, it can be shown that the global concurrency control problem turns out to be a problem of maintaining the *compatibility* of the serialization orders of the global subtransactions. In this section, we formalize the hierarchical concurrency control approach and prove its correctness. To facilitate the discussion, we define a serialization function S_k for the local history h_k. Let ST_k denote the set of all the successfully executed global subtransactions and/or local transactions in h_k. The serialization function S_k is defined as follows:

Definition 3.1 *A* serialization function S_k *for the local history* h_k *is a mapping, where*

$$S_k\colon ST_k \rightarrow \mathcal{D}$$

A serialization function maps successfully executed transactions in a local history into their corresponding serialization orders.

Now, we define the compatibility of the serialization orders of the subtransactions of a set of global transaction $\mathcal{G}$ as follows.

Definition 3.2 *The serialization orders of the subtransactions of a set of global transactions* $\mathcal{G}$ *in an MDBS with n LDBSs are compatible if there exists a total order on* $\mathcal{G}$ *such that* $\forall\ G_i,\ G_j \in \mathcal{G}$, *if* G_i *precedes* G_j *in the total order, then for any k from 1 to n,* $S_k(G_{ik}) < S_k(G_{jk})$ *if both* G_{ik} *and* G_{jk} *exist.*

The correctness of the hierarchical approach is stated in the following theorem.

Theorem 3.1 (The hierarchical concurrency control theorem) *A global concurrency control algorithm is correct if the following conditions are satisfied.*

Condition 1 The local concurrency control algorithms are static and maintain the serializability of their corresponding local histories.

Condition 2 The set of the serialization orders of the subtransactions of global transactions are compatible.

In order to check the compatibility of the serialization orders, we propose the *Serialization Order Graph* (SOG) which is defined as follows.

Definition 3.3 *A serialization order graph is a directed graph in which the vertices are the set of all global transactions, and there is an arc from G_i to G_j for any pair of global transactions G_i, $G_j \in \mathcal{G}$, if and only if $\exists$ k such that both G_{ik} and G_{jk} exist and $S_k(G_{ik}) < S_k(G_{jk})$ is true.*

Example 2: Let G_1, G_2 and G_3 be three global transactions executed in an MDBS with three LDBSs. Suppose we have $S_1(G_{31}) < S_1(G_{11}) < S_1(G_{21})$, $S_2(G_{32}) < S_2(G_{22})$, and $S_3(G_{13}) < S_3(G_{23})$. G_1 does not have a subtransaction at $LDBS_2$, and G_3 does not have any subtransaction at $LDBS_3$. The serialization order graph of this global history is shown in Figure 3.

Lemma 3.1 *The serialization orders are compatible if and only if the SOG of the set of the global transactions is acyclic.*

Proof: ($\Rightarrow$) Assuming that the serialization orders are compatible, if $S_k(G_{ik})$ $< S_k(G_{jk})$ for some k then, according to the definition of compatibility,

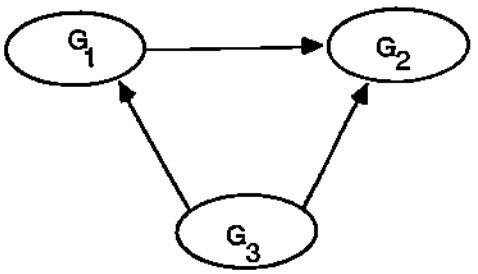

Figure 4: The serialization order graph for a global history

$S_p(G_{ip}) < S_p(G_{jp})$ for any p from 1 to n. Now assume that the SOG is cyclic, let's assume the cycle is $G_1 \rightarrow G_2 \rightarrow \cdots \rightarrow G_k \rightarrow G_1$. Since G_1 precedes G_2, there exists a number k such that $S_k(G_{1k}) < S_k(G_{2k})$. Because the serialization orders are compatible, we have $S_p(G_{1p}) < S_p(G_{2p})$ for any p from 1 to n. Similarly, $S_p(G_{2p}) < S_p(G_{3p})$ for any p. Because the '<' relation is transitive, we conclude that $S_p(G_{1p}) < S_p(G_{1p})$ for any p, which is a contradiction[2]. In other words, if the serialization orders are compatible, there should be no cycle in the SOG. □

($\Leftarrow$) If the SOG is acyclic, a total order on $\mathcal{G}$ which satisfies the condition in definition 3.2 exists and can be obtained by using topological sort on the SOG graph [Deo74]; therefore, the serialization orders are compatible. □

Proof of theorem 3.1:
According to theorem 2.1, a global history is serializable if its corresponding GSG is acyclic. Now let's assume that the global history is not serializable, then its corresponding GSG must contain a cycle. If the cycle contains only one global transaction and some local transactions, then the cycle must also be present in a local history. But condition 1 states that the local history

[2]We assume that there must be at least one subtransaction for each global transaction.

is serializable, so that a cycle should not exist in a local history. Now let's assume that the cycle contains n global transactions and a couple of local transactions, where n is greater than 1. Let the cycle be $G_i \rightarrow L_{i,1} \rightarrow L_{i,2} \rightarrow, \cdots, \rightarrow L_{i,p0} \rightarrow G_{i+1} \rightarrow L_{i+1,1} \rightarrow, \cdots, \rightarrow L_{i+1,p1} \rightarrow G_{i+2} \rightarrow, \ldots, \rightarrow L_{i+2,p2} \rightarrow G_n \rightarrow L_{n,1} \rightarrow, \ldots, L_{n,pn} \rightarrow G_i$. From $G_i \rightarrow L_{i,1}$ we deduce that there are conflict operations between G_i and $L_{i,1}$; furthermore, the conflicting operations of G_i precedes those of $L_{i,1}$. if $L_{i,1}$ is at $LDBS_k$, then we have $S_k(G_{ik}) < S_k(L_{i,1})$. From $L_{i,1} \rightarrow L_{i,2}$, we have $S_k(L_{i,1}) < S_k(L_{i,2})$, and so on. Finally we have $S_k(G_{ik}) < S_k(G_{i+1,k})$. Similarly, we have $S_{k_1}(G_{i,k_1}) < S_{k_1}(G_{i+1,k_1})$, $S_{k_2}(G_{i+1,k_2}) < S_{k_2}(G_{i+2,k_2})$, $\cdots$, $S_{k_{n-1}}(G_{n-1,k_{n-1}}) < S_{k_{n-1}}(G_{n,k_{n-1}})$ and $S_{k_n}(G_{n,k_n}) < S_{k_n}(G_{1,k_n})$ for some $k_1, k_2, \ldots, k_n \in (1, \ldots, n)$. In other words, we have a cycle in the corresponding SOG of the history. By lemma 3.1, the serialization orders are not compatible, which contradicts condition 2 of the theorem. In conclusion, we have shown that if the global history is not serializable, then the two conditions of the theorem do not hold. That is, if the two conditions in the theorem hold then the global history is serializable. □

3.5 Limitations of the Hierarchical Approach

The hierarchical approach is very restrictive in the sense that it only allows a small subset of all serializable histories. It is possible for a serializable history not to satisfy the two conditions of the theorem 3.1. This is illustrated by the following example.

Example 3: Consider the MDBS in example 1, suppose now we have the the following local histories:

$$\begin{array}{lll} h_1: & w_{g1}(a) & w_{g2}(a) \\ h_2: & r_{g2}(b) & r_{g1}(c) \end{array}$$

According to the local histories, we have $S_1(G_{11}) < S_1(G_{12})$ at $LDBS_1$ (since $w_{g1}(a)$ conflicts with $w_{g2}(a)$ and precedes $w_{g2}(a)$) and possibly we may also have $S_2(G_{22}) < S_2(G_{12})$ (this depends on which subtransaction is serialized first) at $LDBS_2$. In this case, the serialization orders are not compatible. However, the global history is serializable. This is because the fact that the subtransactions G_{12} and G_{22} at $LDBS_2$ do not conflict is not taken into account in the conditions of theorem 3.1. Based on this observation, it seems that the two conditions in theorem 3.1 are too restrictive. However, due to the lack of local information[3], we do not know whether the local transactions at the local database systems will introduce the indirect order between two non-conflicting subtransactions or not. The only thing that we can do is to assume that they do conflict, even though they may not conflict at all. In facts, the impossibility of detecting the conflict between subtransactions is the major obstacle which hinders the design of an efficient global concurrency control algorithm.

4 The Site Queue Algorithm

Using the assumption that local concurrency control algorithms are static, the GCC is reduced to maintaining the compatibility of the serialization orders of subtransactions. In this section, we present an algorithm for maintaining the compatibility of the subtransaction serialization orders. The proposed algorithm is a top down approach in the sense that the GCC decides the serialization orders of the subtransactions at the global level and then enforces them at the local level. In this section, we will also discuss a way of simulating the *prepared state* for the basic *two phase commit* protocol [Gra79].

[3] A limitation due to the local autonomy requirement

4.1 Assumptions

Before we outline the site queue algorithm, we first state our assumptions as follows:

A1 For all local concurrency control algorithms, the serialization point of a transaction is the same as its serialization order.

A2 The communication network is capable of maintaining the order of the messages it sends such that the order of the messages received at the destination site is the same as the order of the messages sent from the source site.

The first assumption is used to simplify the algorithm. This assumption can be relaxed by some minor modifications to the algorithm (see next section). The second assumption can be relaxed by appending with each message the message identifiers of those messages which are sent ahead of it. The LDBS can then use this information to restore the order of the messages.

In addition to these two assumptions, we also assume that the serialization point of a subtransaction is visible to the GCC. One way of exposing the serialization point without violating the local autonomy is discussed a latter section.

4.2 Site Queue Concurrency Control Algorithm

A server is created to maintain a subtransaction queue (Figure 5). The server and the queue constitute the global-local interface between the LDBS and the GTM. Each global transaction is decomposed into a set of subtransactions. The GCC first determines an order among the global transactions, and then submits the subtransactions of global transactions to the proper servers according to the pre-determined global transaction order. If a server

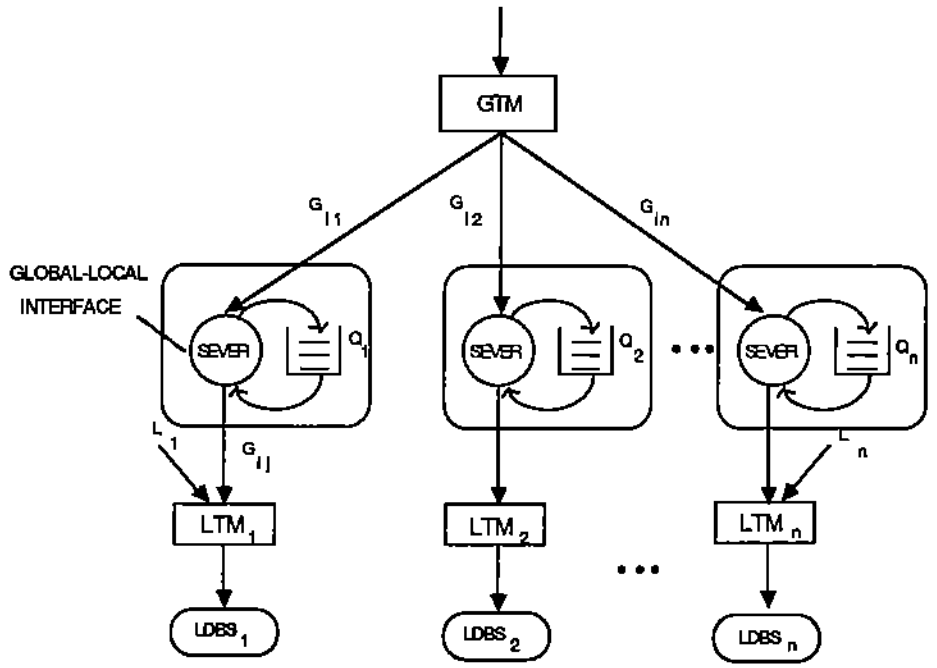

Figure 5: The Site Queue of an MDBS

receives a subtransaction, it will insert the subtransaction to the rear of its queue. The server continuously submits a subtransaction to the LTM according to the following submission rule.

Submission Rule The server can submit a subtransaction to the LTM for execution only when the subtransaction is in the front of the queue and the previously submitted subtransaction is serialized or aborted.

An outline for the server is shown in figure 6.

It is noted that since a subtransaction may not be serialized before it is aborted, an ABORT message has to be sent to the server if a subtransaction is aborted. Since the subtransactions of global transactions are queued one after the other, and are then serialized in the local database system according to their positions in the queue, the site queue algorithm maintains the

```
LOOP:
        do forever
        begin
              on receiving a subtransaction
                 begin
                       insert the subtransaction into the rear of the queue;
                       go to LOOP;
                 end;
              if the queue is not empty then
                    begin
                          submit the first subtransaction in the queue to the LTM;
                          wait for SERIALIZED or ABORT from the previously
                          submitted subtransaction;
                    end;
        end;
```

Figure 6: Pseudo code for the server

compatibility of the subtransaction serialization orders. According to theorem 3.1, the site queue algorithm maintains the serializability of the global history.

For some local concurrency control algorithms, the order domain is the the same as the time domain. However the serialization point may not be the same as the serialization order (as in VD). In this case, the server has to wait for the previously submitted subtransaction to reach its serialization order rather than its serialization point, before it can submit the next subtransaction.

4.3 Exposing Serialization Points and Simulating Prepared States

It is discussed in [GL84] that the major problem of implementing the two phase commit protocol in an MDBS is the lack of *prepared state* in the local database systems. A prepared state is one in which a transaction finishes all of its read and computation operations and has all of its updates stored in a stable storage. A transaction in prepared state is able to commit or abort according to a global decision. Since a prepared state may not be supported by the local database systems, we simulate a prepared state for a subtransaction by properly restructuring the subtransaction. This restructuring also makes the serialization point of a subtransaction visible to the outside world.

In the proposed format (Figure 6), a subtransaction contains database operations, communication primitives and control statements. All of them are enclosed within a BEGIN_TRANSACTION and an END_TRANSACTION. It is assumed that the local database system buffers the write operations in the private working area of the transaction until the transaction issues its **commit** operation. It is also assumed that the local database system supports a **rollback** operation which can recover a failed transaction. Every database operation is embedded within a conditional statement which will take a proper action when the execution of the database operation fails. At some point in the transaction when the subtransaction is first serialized, a **send** operation is inserted, which will report a SERIALIZED message to an agent of the GCC. If all the database operations are successfully executed, the subtransaction will wait for a PREPARE message from the coordinator of the two phase commit protocol. Subsequently, if a PREPARE message is received, the transaction will respond with a READY message to the coordinator, and then waits for a COMMIT or ABORT message from the coordinator. If a COMMIT message is received, the subtransaction will commit the subtransaction by issuing a **commit** operation. Otherwise, the

```
BEGIN_TRANSACTION
    First database operation;
    if (EXEC-CODE = ERROR) then go to FAILED;
    Second database operation;
    if (EXEC-CODE = ERROR) then go to FAILED;
                        .
                        .
                        .
    send(trans_id, server, "SERIALIZED");
                        .
                        .
                        .
    Last database operation;
    if (EXEC-CODE = ERROR) then go to FAILED;
WAIT:
    receive(trans_id, coordinator, message);
    if (message = "PREPARE") then go to PREPARED;
    go to WAIT;
PREPARED:
    send(trans_id, coordinator, "READY");
    receive(trans_id, coordinator, message);
    if (message = "COMMIT") then commit
    else rollback;
    go to END;
FAILED:
    send(trans_id, server, "ABORT");
    send(trans_id, coordinator, "ABORT");
    rollback;


END:
END_TRANSACTION
```

Figure 7: A format for subtransaction

subtransaction will abort itself by issuing a **rollback** operation. In case there is any failure in executing a database operation, subtransaction will abort itself and respond with an ABORT message to both the GCC agent and the coordinator. All of the above stated communications is done by using the **send** and **receive** primitives.

By restructuring a subtransaction as above, we simulate a prepared state into the subtransaction without violating the local autonomy. The role of the participant of the two phase commit protocol [BHG87] is assumed by the subtransaction, while the coordinator is the same as usual and must be implemented in one of the local database systems. Since the two phase commit protocol with simulated prepared state is very similar to the basic two phase commit protocol[4], it will not be further detailed in this paper. It is worth noting that since the updates are not stored in a stable storage, the two phase commit protocol with simulated state can not tolerate site failure. However, it can tolerate the subtransaction failure.

5 Survey of the Existing Global Concurrency Control Algorithms

Most of the proposed global concurrency control algorithms can be classified into the hierarchical concurrency control approach. Depending on how the compatibility of the serialization orders is maintained, most of the global concurrency control algorithms can be further classified into one of the two classes. In the first, a global transaction is first executed without any global control. At commit time, the execution of the global transaction is validated against the set of committed global transactions. The validation is done by the GCC by comparing the serialization orders of the subtransactions with the serialization orders of a set of recently committed global subtransactions. This is the bottom up approach. In the second, GCC controls the execution

[4]It is similar to the basic 2-PC except the logging activity.

of the subtransactions such that serialization orders are prevented from being incompatible. This is the top down approach.

The superdatabases approach proposed by [Pu88] is an example of the bottom up approach. In this approach, the LDBSs report to the superdatabase[5] the serialization order of each subtransaction under its control. The serialization order of a subtransaction is called the *O_element* (order-element). The *O_elements* of the subtransactions of a global transaction is then used to construct an *O_vector*. The superdatabase then searches for a consistent position for this *O_vector* in the set of *O_vectors* of the recently committed global transactions. If a consistent position can be found, then the global transaction is committed, otherwise, it is aborted.

Altruistic Locking protocol [AGS87] and Non-two-phase Locking protocol [Vid87] are examples of the top down approach. In these protocols, locking is used to maintain the compatibility of the serialization orders. Before submitting a subtransaction to an LDBS, the global transaction must lock the intended LDBS. A subtransaction can be submitted to a local database system only when the lock of the LDBS is available (not locked). The way that the LDBSs are locked and released must follows some correct protocols to guarantee the compatibility of the serialization orders. The Altruistic Locking protocol is a variant of the two phase locking protocol, which allows early release of locks. In the Non-two-phase Locking protocol, the LDBSs are first ordered as a rooted tree, then the tree protocol [KS86] is applied on this rooted tree. Both of these two protocols can be used as the global concurrency control protocol. It is to be noted that the static property is also required for the top down approach. The static property is needed to guarantee that the submission order of the subtransaction is the same as its serialization order. As shown in example 1, since the local concurrency control algorithm in $LDBS_2$ is not static, even though the subtransaction of G_1 is submitted before that of G_2, the serialization order of the subtransaction

[5]It is a global transaction manager.

of G_2 precedes that of G_1.

An example of a protocol that does not follow the the hierarchical approach is the site graph algorithm proposed by Breitbart [BST87]. In this algorithm, a site graph is constructed in which nodes are sites (LDBSs), and edges are global transactions. If a global transaction accesses two data items on two different sites, an edge which is labeled by this global transaction is added between these two sites. The global serializability is maintained by retaining the acyclicity of the site graph. A database operation can be executed only if it does not create a cycle in the site graph. This approach allows low degree of concurrency for global transactions, since it does not allow any two global transactions to concurrently access more than one common site. Furthermore, it is not easy to purge the graph.

Generally speaking, the bottom up approach suffers from a high abortion rate of the global transactions. This can be illustrated by the following simple analysis. Let us assume that a bottom up approach is used for global concurrency control. Because of local autonomy, every pair of subtransactions executed on the same LDBS are assumed to conflict with each other. Consider an MDBS with three LDBSs. Suppose that there are two global transactions. Each global transaction has one subtransaction on each LDBS. Let us further assume that for any pair of subtransactions on the same LDBS, the probability for the serialization order of one to precede the other is $\frac{1}{2}$. Then the probability for these two global transactions to have compatible serialization orders is $\frac{1}{4}$. If the number of global transactions increases to three in the above example. Then the probability for the serialization orders to be compatible becomes $\frac{1}{36}$, which is very low. It can be shown that when the number of concurrent global transactions becomes large, the completion rate in the bottom up approach will be small. Since aborting global transactions is costly. We conjecture that the top down approach is more efficient.

6 Conclusion

One way of doing global concurrency control in an MDBS is to impose a control hierarchy on the GCC and LCCs. In a hierarchical concurrency control, LCCs control the execution of local transactions and global subtransactions to retain the serializability of the local executions; while GCC controls the execution of global subtransactions to maintain the compatibility of the subtransaction serialization orders. However, this approach is not applicable to all MDBS environments. In this paper, we identify a class of local concurrency control algorithms on which the hierarchical concurrency control approach can be applied. This class of local concurrency control algorithms is characterized by having the static property. One contribution of this paper is to highlight this property. Other contributions are (1) to formalize the hierarchical approach and prove its correctness; (2) to propose a new deadlock free global concurrency control algorithm; and (3) to suggest a way of implementing the two phase commit protocol.

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.

[BG85] P. Bernstein and N. Goodman. Serializability theory for database. *Journal of Computer and System Sciences*, 31(3):355–374, 1985.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases Systems.* Addison-Wesley Publishing Co., 1987.

[BST87] Y. Breitbart, A. Silberschatz, and G. Thompson. An update mechanism for multidatabase systems. In *IEEE Data Engineering*, pages 135–142, 1987.

[Cas81] M.A. Casanova. *The Concurrency Conntrol Problem of Database Systems*, volume 116. Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1981.

[DELO89] W. Du, A. Elmagarmid, Y. Leu, and S. Ostermann. Effects of autonomy on global concurrency control in heterogeneous distributed database systems. In *Proceedings of the Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, Gaithersburg, MD, October 1989.

[Deo74] N. Deo. *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, 1974.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a database system. *Communication of ACM*, 19(11):624–633, 1976.

[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.

[GK88] H. Garcia-Molina and B. Kogan. Node autonomy in distributed systems. In *Proc. Int'l Conf. on Data Engineering*, pages 158–166, 1988.

[GP86] V.D. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Inform. Systems*, 11(4):287–297, 1986.

[Gra79] J.N. Gray. *Notes on database operating systems.* Operating Systems: An Advanced Course.Springer-Verlag, New York, 1979.

[KR81] H.T. Kung and J. Robinson. On optimistic methods for concurrency control. *Transactions on Database Systems*, 6(2):213–226, June 1981.

[KS86] H. Korth and A. Silberschatz. *Database System Concepts.* McGrawHill, 1986.

[LT88] W. Litwin and H. Tirri. Flexible concurrency control using value dates. *IEEE Distributed Processing Technical Committee Newsletter*, 10(2):42–49, November 1988.

[Pu88] C. Pu. Superdatabases for composition of heterogeneous databases. In *Proceedings of the International Conference on Data Engineering*, 1988.

[Vid87] K. Vidyasankar. Non-two-phase locking protocols for global concurrency control in distributed heterogeneous database systems. In *CIPS Edmonton*, 1987.