Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1991

On the Impossibility of Atomic Commitment in Multidatabase Systems

James G. Mullen

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

Report Number:

91-029

Mullen, James G. and Elmagarmid, Ahmed K., "On the Impossibility of Atomic Commitment in Multidatabase Systems" (1991). *Department of Computer Science Technical Reports.* Paper 876.
https://docs.lib.purdue.edu/cstech/876

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

ON THE IMPOSSIBILITY OF ATOMIC COMMITMENT IN MULTIDATABASE SYSTEMS

James G. Mullen
Ahmed K. Elmagarmid

CSD-TR-91-029
April 1991

On the Impossibility of Atomic Commitment in Multidatabase Systems

James G. Mullen and Ahmed K. Elmagarmid

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

This paper examines the problem of performing the atomic commitment of (global) transactions in multidatabase systems. We show that if the autonomy of the local database systems is preserved, it is impossible to perform atomic commitment in general, even under the assumption that there are no system failures. We also show that even when it is assumed that all local database systems use strict two phase locking (an assumption useful for performing global concurrency control), atomic commitment is impossible if even a single system failure can occur.

1 Introduction

Multidatabase systems combine (local) database systems into a global distributed database system. Multidatabase systems differ from traditional distributed database systems in that the database systems that they combine may be heterogeneous and autonomous. Heterogeneity may occur in the database management systems (DBMSs) and in the semantics of the data, however, dealing with the heterogeneity issues is beyond the scope of this paper. Autonomy, or local autonomy, refers to the ability of local database systems in the multidatabase to choose their own design and operational behavior. Local autonomy is often necessary in multidatabase systems because they combine pre-existing database systems that are not easily modified.

Transaction management is an important issue in multidatabase systems, and much research has been done on concurrency control in multidatabase systems, generally under the

assumption that there are no system failures, e.g. [EL87, BS88b, LT88, Pu88, DE89]. Recently, research has started to look at the effects of failures on transaction management in multidatabase system, e.g. [BST90, WV90].

One goal of transaction management is to prevent failures from allowing the atomic commitment of transactions (either all or none of the effects of a transactions take place). *Atomic commitment protocols* are used in distributed database systems to ensure that the effects of a transaction are either uniformly committed or uniformly aborted at each participating site in the database system, even in the event of failures. Two phase commit (2PC) [LS76, Gra78] is a common and simple atomic commitment protocol used in traditional homogeneous distributed database systems. 2PC consists of two phases: a voting phase, and a decision phase. In the voting phase each participating site votes *yes* or *no* corresponding to whether it wants to commit or abort the transaction. If a site votes *yes* it is considered to be in a *prepared-to-commit* state where it has not committed the transaction, but it guarantees that it is able to commit it. If a site votes *no*, it aborts the transaction. In the decision phase the 2PC coordinator looks at all the votes, and if they are all *yes*, it sends a commit message to each participant. Otherwise it sends an abort message to the sites that voted *yes*.

In multidatabase systems, the goal of atomic commitment protocols is to ensure that all the subtransactions of a global transaction are either uniformly committed or uniformly aborted. However, atomic commitment is more difficult to implement in the multidatabase environment, because of the desire to preserve the autonomy of the local database systems. For example, the two phase commit protocol cannot be directly implemented in multidatabase systems, because assuming that the local database management systems support a (visible) prepared-to-commit state would violate local autonomy.

We show in this paper that atomic commitment *in general* is not possible in multidatabase systems without violating local autonomy. This is true even without the occurrence of system failures. We also show that if one assumes that all local database systems use strict two phase locking [BHG87] as their concurrency control method (an assumption useful for performing concurrency control, but a violation of local autonomy), atomic commitment is impossible if even a single system failure can occur.

The organization of the remainder of this paper is as follows. Section 2 covers background material and presents our assumptions. Section 3 presents our impossibility theorems for atomic commitment in multidatabase systems. Section 4 covers related work, and section 5 presents our conclusions.

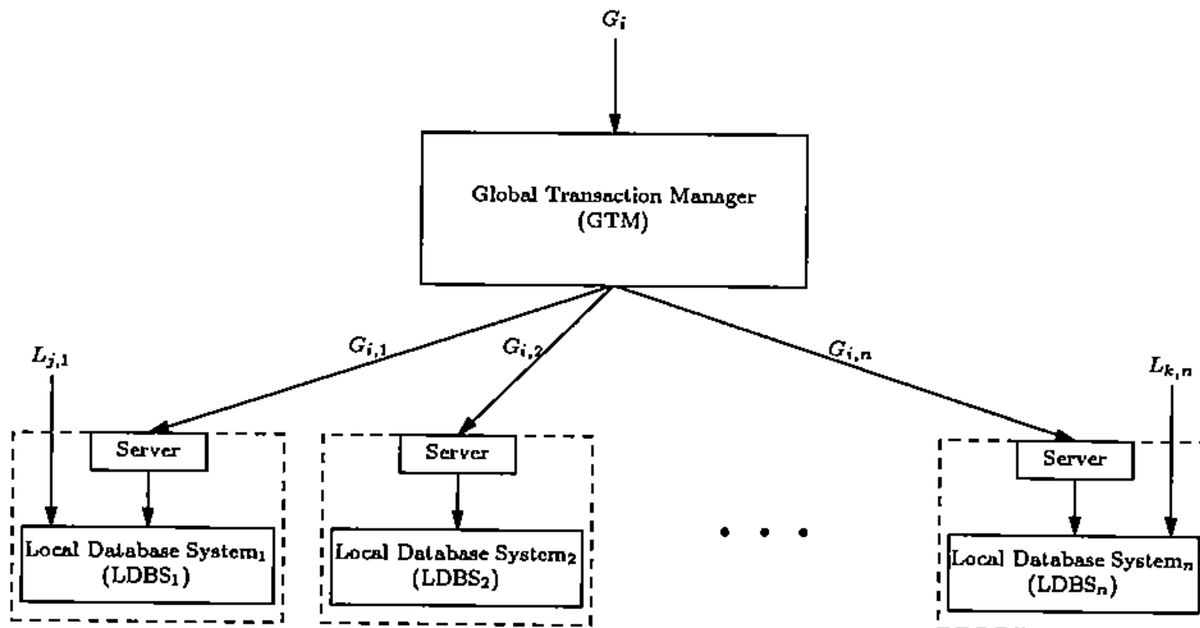

Figure 1: Mulitdatabase Model.

2 Background

Multidatabase Model. A multidatabase consists of a Global Transaction Manager (GTM) and Local Database Systems (LDBSs) connected by a communication network. Each Local Database System is considered to consist of a Local Database Management System (LDBMS) and a database. Users can submit *global transactions* (transactions that possibly access multiple systems' data) to the GTM, and *local transactions* (transactions that only access one system's data) directly to an LDBMS. Each system that has an LDBS is considered to have a server that accepts transaction operations from the GTM and sends them to the LDBMS.

From the user's view, a transaction is considered to be the execution of a program with embedded transaction operations. The view of a transaction to the GTM and LDBMSs will be a (finite) stream of read, write, commit, and abort operations and we will use the "standard" formal definition of transactions as found in [BHG87]. We assume, by definition, that transactions halt on all inputs and preserve database consistency. The notation we use has been extended to indicate which LDBS an operation accesses, and whether the operation belongs to a local or global transaction. Some examples of our notation and their definitions are given in table 2.

Notation	Definition
G_i	Global transaction number i
$G_{i,j}$	Global subtransaction at LDBMS_j of transaction number i
$L_{i,j}$	Local transaction number i running at LDBMS_j
$r_{g_{i,j}}[x]$	read data item x belonging to global subtransaction $G_{i,j}$
$r_{l_{i,j}}[x]$	read data item x belonging to local transaction $L_{i,j}$

Table 1: Notation Examples

The multidatabase model used in this paper assumes the following:

1. The only assumption, in general, that can be made about the LDBMSs is that they support the ACIDity properties (Atomicity, Consistency, Isolation, and Durability) [HR83] of their transactions. As a result, one can assume that the schedules that the LDBMSs produce are (conflict) serializable and recoverable, although not necessarily strict. If any additional assumptions are made (e.g. LDBMSs use strict two phase locking) or any modifications to the LDBMSs are required, local autonomy is considered to be violated.

2. LDBMSs cannot distinguish between the local transactions and global subtransactions they execute.

3. The only interface the GTM has to the LDBMSs is the standard transaction operation interface the LDBMSs provide to users on their local system. So, in general, the GTM has no direct knowledge of, or control over, the local transactions being executed. However, we allow the GTM to submit *control transactions* (transactions submitted by the GTM itself, and not by a user). For example, a *compensating transaction* (used to undo the effects of another transactions) [GM83] would be considered a control transaction. The transaction operations executed by a control transaction are considered to be *control operations*.

4. The GTM separates each global transaction into one subtransaction for each LDBMS that the global transaction accesses. If more than one subtransaction is executed at a given LDBMS it is possible, in general, that global serialization order will be violated [GPZ86]. For example, suppose a global transaction has two subtransactions $G1$ and $G1'$ that execute at the same LDBMS. A non-serializable order such as $G1 \rightarrow L \rightarrow G1'$ could result.

Note that assumption (1) describes our definition of violating local database autonomy. Many difficulties are caused by the desire in multidatabase systems not to violate local autonomy [DELO89, DEK90], and this desire is not only a pedagogical one. In real systems

it may be very difficult to violate local autonomy, even if one so desired. A company might not have source code versions of the DBMSs it is using, might not have permission to make changes, or might not have the resources to make the required modifications, especially if the modifications have to be integrated with each new release of DBMS software.

Correctness of Atomic Commitment Protocols. We define an atomic commitment protocol (ACP) to be correct if for each global transaction G_i submitted to the GTM, the GTM:

1. Uniformly commits or aborts all subtransactions of G_i in a finite amount of failure-free time.

2. Preserves database consistency. Since we assume that each transaction (when executed in isolation and with no failures) preserves database consistency, it must do so when using the ACP.

3. Commits all subtransactions if no other global or local transactions are currently executing and there are no failures.

Condition (1) is basically a standard condition, however we emphasize the fact that the ACP must commit or abort all subtransactions in a finite amount of failure-free time to prevent protocols that could possibly block a transaction indefinitely (for reasons other than persistent system failures) from being considered. Condition (2) is standard. Condition (3) is made to disallow protocols of the form "abort all transactions" or "abort all transactions except for read-only transactions" from being considered. That is, an ACP must not limit the class of transactions it excepts. Any transaction that runs in isolation and without failures should commit.

3 Atomic Commitment in Multidatabase Systems

In this section we first show that it is impossible to implement an atomic commitment protocol without violating local autonomy in multidatabase systems. This is true even in the absence of system failures. It clearly follows, and we state it as a corollary, that the two phase commit protocol cannot be implemented or simulated in multidatabase systems without violating local autonomy. Since (global) concurrency control may not be possible without violating local autonomy, we also show that atomic commitment is not possible under conditions where concurrency control is possible. That is, we show that even when one assumes that all LDBMSs use strict two phase locking as their concurrency control method, atomic commitment is not possible if even a single system failure can occur.

Theorem 1 *It is impossible to implement an atomic commitment protocol in a multidatabase system without violating local autonomy, even assuming no system failures occur.*

Proof: The following is a proof by counter example. That is, we show a situation where there are no system failures, yet it is impossible to atomically commit a global transaction. Suppose a global transaction is submitted by a user that transfers $100 from account x to account y. The program for this transaction is as follows:

```
procedure A_Transfer
begin
    Start;
    temp := Read(x);
    if temp < 100 then begin
        output("insufficient funds");
        Abort
    end
    else begin
        Write(x, temp - 100);
        temp := Read(y);
        if temp > 100000 then begin
            output("FDIC insurance limit already exceeded");
            Abort
        else begin
            Write(y, temp + 100);
            Commit;
            output("transfer completed")
        end
    end;
    return
end
```

Account x is stored at $LDBMS_1$ and y is stored at $LDBMS_2$. Assume that the LDBMSs use two phase locking certifiers (as defined in [BHG87]) to perform concurrency control. That is, each transaction operation is processed as received and when the transaction's commit operation is received the read set and write set of the transaction are checked to see if they conflict with the read or write set of some other current transaction. If so, the transaction is aborted, otherwise it is committed. If there are no transaction failures (e.g. insufficient funds) the transaction program will submit the following order of operations to the GTM.

$$G_1 : r_{g1}[x] \; w_{g1}[x] \; r_{g1}[y] \; w_{g1}[y] \; c_{g1}$$

which will be logically divided into two subtransactions:

$$G_{1,1} : r_{g1,1}[x] \; w_{g1,1}[x] \; c_{g1,1}$$
$$G_{1,2} : r_{g1,2}[y] \; w_{g1,2}[y] \; c_{g1,2}$$

The global transaction assigns the values of the following functions to x and y:

$$x := f_1(x,y) = \begin{cases} x - 100 & \text{if } x \geq 100 \text{ and } y \leq 100000 \\ x & \text{otherwise} \end{cases}$$

$$y := f_2(x,y) = \begin{cases} y + 100 & \text{if } x \geq 100 \text{ and } y \leq 100000 \\ y & \text{otherwise} \end{cases}$$

First we will show that it is possible for the GTM to get into a state (even without failures) where it has committed one of the subtransactions of G_1 and aborted the other. Clearly, $c_{g_{1,1}}$ must be submitted before, at the same time, or after $c_{g_{1,2}}$. We will assume that $c_{g_{1,1}}$ is submitted before, or at the same time as $c_{g_{1,2}}$. The proof assuming the other case of $c_{g_{1,1}}$ being submitted after $c_{g_{1,2}}$ is analogous to the proof presented, so it is omitted. We will assume that no local transactions executed at LDBMS$_1$ while $G_{1,1}$ executed, and that $x >= 100$. Therefore, $G_{1,1}$ commits and the history at LDBMS$_1$ (up to the execution of $c_{g_{1,1}}$) looks as follows (where o_x is a control operation for $x = 1$ to k, and $i = j = k = 0$ implies there are no control operations):

$$H_1 : \; o_1 \; o_2 \; \ldots \; o_i \; \overbrace{\overbrace{r_{g_{1,1}}[x]} \; o_{i+1} \; \ldots \; o_j \; \overbrace{w_{g_{1,1}}[x]} \; o_{j+1} \; \ldots \; o_k \; \overbrace{c_{g_{1,1}}}}^{G_{1,1}}$$

The control operations are included so that we will consider any possible submission of operations the GTM might make. Since the submission of operations is the only interface the GTM has to the LDBMSs, we are considering all possible atomic commitment protocols the GTM might attempt.

We assume that no local transactions run at LDBMS$_2$ prior to the last operation the GTM submits to LDBMS$_2$ before $c_{g_{1,2}}$:

$$H_2 : \; o_1 \; o_2 \; \ldots \; o_i \; \overbrace{\overbrace{r_{g_{1,2}}[y]} \; o_{i+1} \; \ldots \; o_j \; \overbrace{w_{g_{1,2}}[y]}}^{G_{1,2}} \; o_{j+1} \; \ldots \; o_k$$

Suppose that just before $c_{g_{1,2}}$ is submitted (but after o_k) a local transaction $L_{1,2}$ executes so that the following history is generated at LDBMS$_2$:

$$H_2 : \; o_1 \; o_2 \; \ldots \; o_i \; \overbrace{\overbrace{r_{g_{1,2}}[y]} \; o_{i+1} \; \ldots \; o_j \; \overbrace{w_{g_{1,2}}[y]} \; o_{j+1} \; \ldots \; o_k \; \underbrace{r_{l_{1,2}}[y]}_{L_{1,2}} \; \overbrace{a_{g_{1,2}}}}^{G_{1,2}} \; \underbrace{c_{l_{1,2}}}_{L_{1,2}} \; o_{k+1} \; \ldots \; o_l$$

No possible operations submitted by the GTM would prevent $r_{l_{1,2}}[y]$ from executing and causing $G_{1,2}$ to abort. So, we end up in a state where $G_{1,1}$ is committed and $G_{1,2}$ is aborted. Note that this behavior is dependent on our assumption that $LDBMS_2$ is using a two phase locking certifier. If strict two phase locking was used, this history could not occur, since $r_{l_{1,2}}[y]$ would be prevented from executing until $c_{g_{1,2}}$ executed and the (write) lock for item y was released.

The database is now in an inconsistent state, since $G_{1,1}$ has been committed, and $G_{1,2}$ has been aborted. However, if the inconsistency can be corrected before other transactions view the inconsistency, it won't matter. Since we assume that the GTM could prevent other *global* transactions from running while the inconsistency is resolved, the problem becomes one of resolving the inconsistencies before local transactions access the data or showing that it does not matter if local transactions access the data before the inconsistency is resolved. Exactly one of the following two approaches must be taken to resolve the inconsistency:

1. Undo (compensate) the effects of the committed subtransaction $G_{1,1}$:

$$x := f_1^{-1}(x,y) = \begin{cases} x + 100 & \text{if } x \geq 100 \text{ and } y \leq 100000 \\ x & \text{otherwise} \end{cases}$$

2. Redo and commit the effects of the aborted subtransaction $G_{1,2}$:

$$y := f_2(x,y)$$

(Undo) If local transactions could be prevented from running, then $G_{1,1}$ could be undone (or compensated) by either assigning x the value of $f_1^{-1}(x,y)$, or by saving the value of x before $f_1(x,y)$ was executed, and then restoring it. However, the GTM cannot prevent the execution of local transaction $L_{1,1}$ in between the execution of $c_{g_{1,1}}$ and o_{k+1}.

$$H_1 : o_1\ o_2 \ldots o_i\ \overbrace{\widehat{r_{g_{1,1}}[x]}\ o_{i+1} \ldots o_j\ \widehat{w_{g_{1,1}}[x]}\ o_{j+1} \ldots o_k\ \widehat{c_{g_{1,1}}}}^{G_{1,1}}\ \underbrace{r_{l_{1,1}}[x]\ w_{l_{1,1}}[x]\ c_{l_{1,1}}}_{L_{1,1}}\ o_{k+1} \ldots o_l$$

In general the only way $L_{1,1}$ would be prevented from committing is if $o_1 \ldots o_k$ contained an $r[x]$ or $w[x]$ operation that belonged to a transaction that was executing when $c_{l_{1,1}}$ was submitted. This, however, would not be possible, in general, because this condition would also cause $G_{1,1}$ to abort. The GTM cannot possibly know in advance that $L_{1,1}$ will execute, so, in this case, if it was submitting operations such that $G_{1,1}$ would abort, condition (3) of our ACP correctness conditions would be violated. The GTM would be aborting a transaction that ran in isolation without failures.

In this case, since local transactions may execute before $G_{1,1}$ is undone, database consistency may not be preserved. Assume, for example, that $L_{1,1}$ is as follows:

```
procedure Interest
begin
    Start;
    temp := Read(x);
    Write(x, temp * 1.01);
    Commit;
    return
end
```

$L_{1,1}$ performs the following assignment:

$$x := g(x) = x \times 1.01$$

Also assume that x has the value 100 before $G_{1,1}$ is executed. Due to the effects of $G_{1,1}$, $L_{1,1}$ reads the value 0. Therefore, $L_{1,1}$ also writes the value 0 back to x. So, in effect, the interest payment is lost, thus violating local database consistency.

It is impossible for the GTM even to make the local database eventually consistent, that is, to make the database consistent after $L_{1,1}$ has executed. The GTM, not having any direct knowledge of local transactions, cannot discern if $L_{1,1}$ multiplied x by 1.01 or added 0 to it. In the first case, x should be set to 101, and in the second case x should be set to 100. Therefore, the appropriate value for x cannot be determined by the GTM.

The GTM cannot prevent local transactions from executing after $G_{1,1}$ has executed, and it cannot determine how to restore the consistency of the database after the local transactions have executed. So, it is impossible for the GTM to successfully undo the effects of $G_{1,1}$. Condition (2) of our ACP correctness conditions will be violated, since database consistency cannot be preserved.

(Redo) Assuming the GTM knows how to redo $G_{1,2}$, the problem is how to prevent local transactions from accessing y until $G_{1,2}$ can be redone. However, this is not possible for the same reason that the GTM cannot prevent local transaction from executing before it can undo $G_{1,1}$. The following local history could result:

$$H_2: o_1\ o_2 \ldots o_i\ \overbrace{\overbrace{r_{g_{1,2}}[y]}\ o_{i+1} \ldots o_j\ \overbrace{w_{g_{1,2}}[y]}\ o_{j+1} \ldots o_k\ \underbrace{r_{l_{1,2}}[y]}_{L_{1,2}}\ \overbrace{a_{g_{1,2}}}}^{G_{1,2}}\ \underbrace{c_{l_{1,2}}}_{L_{1,2}}\ \underbrace{r_{l_{2,2}}[y]\ w_{l_{2,2}}[y]\ c_{l_{2,2}}}_{L_{2,2}}\ o_{k+1} \ldots o_l$$

While the execution of this local transaction may not cause a local database inconsistency, since one might be able to consider $L_{2,2}$ to come before G_1 in serialization order, it is possible for the local transaction to cause attempts to redo $G_{1,2}$ to abort indefinitely. Suppose $L_{2,2}$ is as follows:

```
procedure Big_Deposit
begin
    Start;
    temp := Read(y);
    Write(y, temp + 100000);
    Commit;
    return
end
```

$L_{2,2}$ will prevent $G_{1,2}$ from being redone. A transaction failure will occur if it is attempted, because the amount of money in the account is now too large. Therefore, $G_{1,2}$ could be blocked indefinitely, even though no system failures occur. This violates condition (1) of our ACP correctness conditions.

Therefore, we have shown an example where no matter what operations the GTM submits (its only form of control) it cannot atomically commit a transaction. It cannot guarantee that both subtransactions of a transaction will commit. And, if one does commit and the other aborts, there is no way to resolve the inconsistency created without causing another inconsistency or possibly blocking the transaction indefinitely, both of which are violations of our correctness conditions. □

Since two phase commit (2PC) is an instance of an atomic commitment protocol, it clearly follows that 2PC cannot be implemented or simulated in multidatabase systems.

Corollary 1 *It is impossible to implement the two phase commit protocol in multidatabase systems without violating local autonomy.*

It is not clear that one can devise a correct concurrency control method for systems with the above assumptions (i.e. no violation of local autonomy). However, if one makes the assumption that each LDBMS uses strict two phase locking for its concurrency control method, then correct concurrency control is possible [BS88a]. However, it is still not possible to implement atomic commitment protocols even with this assumption (which violates local autonomy).

Theorem 2 *If all the LDBMSs in a multidatabase system are assumed to use strict two phase locking as their concurrency control method, it is impossible to implement an atomic commitment protocol that can tolerate even a single system failure.*

Proof: Assume the same system and transactions as in the proof for theorem 1, with the exception that the LDBMSs use strict two phase locking as their concurrency control method. Although one of the subtransactions could not abort for the same reason as in the proof for the first theorem, a system failure could cause it to abort in the same way.

$$H_2: \; o_1 \; o_2 \; \dots \; o_i \; \overbrace{\overbrace{r_{g_{1,2}}[y]} \; o_{i+1} \; \dots \; o_j \; \overbrace{w_{g_{1,2}}[y]} \; o_{j+1} \; \dots \; o_k \; \underbrace{*******}_{System Failure} \; \overbrace{a_{g_{1,2}}}}^{G_{1,2}} \; o_{k+1} \; \dots \; o_l$$

Assuming that a system failure has caused the abort, the rest of the proof is analogous to the proof for theorem 1. □

4 Related Work

In [BST90] and [WV90] methods for transaction management in multidatabase systems are presented that support serializable global histories, and correct operation even in the event of system failures. These methods make the following assumptions:

1. All local database systems use strict two phase locking (a violation of design autonomy).
2. The DLU (Denied Local Updates) [WV90] property holds. Data items in the databases can either be updated globally or updated locally, but not both. We see two ways to implement this property. One is by violating local autonomy to implement the property, and the other is to limit the class of global transactions accepted by only allowing read-only global transactions (i.e no data items can be updated globally).
3. The global transaction manager can be in a state where it blocks indefinitely. However, this is not likely, since the DLU property is maintained.

In [EJK91] a method is shown that does not assume the DLU property, but allows global transactions to perform updates. The method does require some assumption about the LDBMS concurrency control method (strict two phase locking is a sufficient assumption), and the class of global transactions allowed is restricted. Basically, subtransactions are not allowed to have cyclical functional dependencies. For example, the transfer transaction example used in this paper would not be allowed because $G_{1,1}$ would be considered to depend on $G_{1,2}$, and $G_{1,2}$ would be considered to depend on $G_{1,1}$.

5 Conclusions

Implementing atomic commitment protocols in multidatabase systems is difficult because of the desire to preserve local autonomy. We have shown that, in general, implementing

atomic commitment is impossible without violating local autonomy, even without system failures. And, that it is also impossible to perform atomic commitment even when one violates local autonomy by assuming local database systems may only use strict two phase locking for concurrency control (an assumption useful for global concurrency control). It clearly follows that specific atomic commitment protocols such as two phase commit, or three phase commit, cannot be implemented or simulated under the above conditions.

As a result, if one desires reliable transaction management in a multidatabase system, one needs to make additional restrictions or assumptions than those required to perform correct concurrency control. We see three possible general approaches:

1. Make further violations of local autonomy, e.g. require each LDBMS to support the DLU property or a visible prepared to commit state.
2. Limit the transaction class allowed, e.g. allow only read-only global transactions.
3. Use a different transaction model, e.g. support only some of the ACIDity properties of global transactions.

References

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading Mass., 1987.

[BS88a] Yuri Breitbart and Avi Silberschatz. Multidatabase Systems with a Decentralized Concurrency Control Scheme. *Distributed Processing Technical Committee Newsletter*, 10(2):35–41, November 1988.

[BS88b] Yuri Breitbart and Avi Silberschatz. Multidatabase Update Issues. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 135–142, June 1988.

[BST90] Yuri Breitbart, Avi Silberschatz, and Glen R. Thompson. Reliable Transaction Management in Multidatabase Systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 215–224, May 1990.

[DE89] Weimin Du and Ahmed K. Elmagarmid. Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the 15th VLDB Conference*, pages 347–355, August 1989.

[DEK90] Weimin Du, Ahmed K. Elmagarmid, and Won Kim. Effects of Local Autonomy on Heterogeneous Distributed Database Systems. Technical Report ACT-OODS-EI-059-90, MCC, February 1990.

[DELO89] W. Du, A. K. Elmagarmid, Y. Leu, and S. D. Ostermann. Effects of Local 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*, pages 113–120, October 1989.

[EJK91] Ahmed K. Elmagarmid, Jin Jing, and Won Kim. Global Commitment in Multidatabase Systems. Technical Report CSD-TR 91-017, Purdue University, March 1991.

[EL87] Ahmed K. Elmagarmid and Y. Leu. An Optimistic Concurrency Control Algorithm for Heterogeneous Distributed Database Systems. *IEEE Data Engineering Bulletin*, 10(3):26–32, September 1987.

[GM83] H. Garcia-Molina. Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GPZ86] Virgil Gligor and Radu Popescu-Zeletin. Transaction Management in Distributed Heterogeneous Database Management Systems. *Information Systems*, 11(4):287–297, 1986.

[Gra78] J. N. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer Verlag, Berlin, 1978.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[LS76] B. Lampson and H. Sturgis. Crash Recovery in a Distributed Data Storage System. Technical report, Computer Science Laboratory, Xerox Palo Alto Research Center, 1976.

[LT88] W. Litwin and H. Tirri. Flexible Concurrency Control Using Value Dates. *IEEE Distributed Processing Technical Committe 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*, pages 548–555, February 1988.

[WV90] Antoni Wolski and Jari Veijalainen. 2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase. In *Proceedings of the PARBASE-90 Conference*, March 1990.