2PC AGENT METHOD: ACHIEVING SERIALIZABILITY IN PRESENCE OF FAILURES IN A HETEROGENEOUS MULTIDATABASE

Antoni Wolski and Jari Veijalainen

Technical Research Centre of Finland
Laboratory for Information Processing
Lehtisaarentie 2A, 00340 Helsinki
Finland

Abstract

A method for integrated concurrency control and recovery, applicable to heterogeneous multidatabase systems is proposed[1]. The role of the participant in the two-phase commit protocol is laid on an entity called 2PC Agent associated with the local database system. The main importance of the method is in preserving global serializability in the presence of unilateral aborts and site failures. The method requires the participating database systems to use the strict two-phase locking or a comparable rigorous concurrency control policy.

Introduction

There has recently been much interest in integrating pre-existing databases managed by heterogeneous database management systems (DBMS). This is understood to be caused by the need to eliminate "islands of information" [11] and, generally, the necessity to improve the interoperability of database systems [23]. There are various architectures supporting these objectives. The *multidatabase* architecture [25] is characterized by preserving various aspects of local system *autonomy* [31, 13] to a great extent. The database systems retain their *design autonomy*, i.e. neither their functional characteristics and the existing interfaces can be modified, nor the database structures changed. In addition, they have *execution autonomy* which is reflected by their freedom to execute a database operation in any locally suitable order or to abort an operation. The *federated database* architecture [19] represents an interoperability approach based on a global framework for schema information exchange.

In order to fully utilize integrated databases, facilities to support transactions spanning distinct databases are needed. The autonomy properties of the systems make them incapable of participating in conventional global concurrency control as well as commit and recovery procedures required to attain the full quality of transaction processing. We propose a method for integrated concurrency control and recovery, applicable to a heterogeneous multidatabase system. Such a system includes ready-made database management systems (called *Participating DBMSs* here)

---

[1] This paper is an enhanced version of the original PARBASE-90 publication. The main changes are the introduction of the two-step subtransaction certification and the inclusion of the certification algorithm descriptions. The sketch of the proof has been updated accordingly.

28 January, 1991. To appear in: N. Rishe, S. Navathe and D. Tal (eds.), Databases: Theory, Design and Applications, IEEE Computer Society Press, 1991.

located, essentially, at distinct computing nodes (*Participating Sites*). The Participating DBMSs are heterogeneous with respect to local DBMS software.

Traditionally, the concept of a correctness of transactions in heterogeneous databases [16] has been based on recoverable and serializable transaction executions [3]. Other transaction models were also proposed [2, 14, 22, 28], with the emphasis on compensation based recovery which was extensively studied,e.g. in [32]. We shall focus our attention on serializability-preserving concurrency control and recovery methods in the sequel.

Most of the latest research work in the area has concentrated on the concurrency control issues limited to failure-free operation. The example methods are the *site graph* method [4], the *altruistic locking* [29], the *cycle detection* method of [30], the optimistic algorithm of [11], the integration method using observability and controllability [21], the *superdatabases* [27], the *top-down approach* of [10], the *value date* method [24] and the *ticket methods* [15]. Problems with allowing l*ocal transactions* have also led to a new correctness criterion, the *quasi serializability*, which is a weaker notion than conflict serializability [8]. Local transactions are transactions executing within the scope of a Participating DBMS, and they are not available for analysis by any entity outside the Participating DBMS.

The common assumption made in the above studies is that the local concurrency control mechanism of a participating local database system is unknown. This leads to approaches which are characterized by serious limitations imposed on the concurrency of global transactions. Also, the above methods typically require a centralized global transaction manager, and some of them require modifications to participating database management systems. For a comparison of some of the methods, see [9, 7].

A notable progress in dealing with failures is represented by the *commit graph* method of [6]. Under certain restrictions the method guarantees the conflict serializability of global and local transactions, and the deadlock detection. Typical failures are taken care of. The local transaction managers need not be modified. The method is based on a centralized transaction management facility utilizing a pessimistic and highly restrictive concurrency control policy.

In this work, we set up a set of similar goals reflecting a practical point of view. We assume the participating DBMSs retain their *design autonomy* and the *execution autonomy* with respect to global requests, which is reflected, among others, by their freedom to *unilaterally abort* an operation at any time. Thus, the recovery from such failures must be included in the method. We also allow for submission of local transactions through the existing local interface although we shall have to restrict them considerably.

The significance of our approach is in that we consider centralized concurrency control solutions impractical, and we intend to take advantage of decentralized dynamic concurrency control policies (e.g. locking) which lead to higher concurrency rates than centralized methods [1].

We assume the objective of the transaction management system is to preserve the traditional characteristics of a transaction (so-called ACID): atomicity, consistency, isolation and durability [18]. The view serializability [3] is assumed as a sufficient consistency criterion.

Typically, in distributed database systems, the ACID properties are attained by means of the basic *two-phase commit protocol* (2PC) [17] or its variations [26], and the related recovery protocols. In the basic 2PC scheme, a *Coordinator* responsible for the global transaction commitment communicates with *Participants* executing the subtransactions. It is typical of the scheme that every Participant has to move a subtransaction to a recoverable *prepared state* before the

global transaction is finally committed. In this state, the Participant does not have the execution autonomy — it is obliged by the protocol to execute the final decision command (Commit or Abort) issued by the Coordinator.

Systems supporting the prepared state are called here *two-phased DBMSs*, whereas systems without an appropriate 2PC interface will be called *single-phased DBMSs*. It is evident that if arbitrary single-phased DBMSs are used as Participating DBMSs and, additionally, submission of local transactions is allowed, then the objective to guarantee ACID properties cannot be met in general. The fact that most of the existing systems are single-phased, and thus neither support the prepared state nor have an external interface for participating in the 2PC protocol, is a major obstacle in the way of heterogeneous DBMS integration.

Therefore, our approach is based on the important restricting assumption that the Participating DBMSs are capable of producing histories of the type produced by strict two-phase locking [3] schedulers. The assumption is derived from the observation that most of the commercially available systems use the *strict two-phase locking* (S2PL) [12] policy. Since this approach is the prevailing one, the design autonomy of the Participating DBMS is not too much reduced by this requirement in practice, while opening up practical possibilities to achieve the ACID properties.

As concerns the applicability of strict 2PL to distributed databases, it has been shown that, in the absence of failures, a global transaction scheduler using the *distributed two-phase locking* protocol produces serializable histories for global transactions [3]. It has also been proved that this is true for a mix of global and local transactions [5]. We shall use the above result as the basis of normal operation, i.e. in absence of failures. In the following, we shall concentrate on achieving atomic transaction commitment and also recovery from the two most common types of failures, i.e. the Participating Site system failure and the unilateral subtransaction abort by a Participating DBMS.

The resulting *2PC Agent Method* for integrating single-phased participating DBMSs into the 2PC scheme is presented below. The assumed architecture and requirements are described in the next section. A short outline of the method follows. Then, a simplified proof of correctness is presented. At the end, the remaining problems are discussed. A note on the prototype implementation of the method is included too.

Assumptions and requirements

Overall architecture

The architecture model of a multidatabase transaction management system we are proposing is shown in Fig. 1. It is built of software modules (boxes) and interfaces (horizontal segments). An interface is described in terms of *commands* which facilitate data exchange and control flow between the interconnected modules. The interaction through an interface consists of issuing a command and the subsequent response returned in the opposite direction. All the interfaces of the model are synchronous, i.e. a command belonging to a given transaction can not be started before the processing of the previous command of the same transaction has been completed at the interface.

By an *operation* we mean the execution of a command. An operation is said to be performed after the result or confirmation is received through the interface in question. The modules are multithreaded in the sense that operations belonging to different transactions may be processed in parallel.

A (*multidatabase*) *global transaction* is understood as a finite series of database operations (in practical sense, e.g. executions of SQL commands) each of which originates within the application execution and provides return values to the application at the *Global Interface* (GI). A *global subtransaction* is a series of all the operations of a global transaction, performed at the *2PC interface* of a particular site. Thus, similarly to transaction models used elsewhere [16, 5, 11, 9], for every global transaction there is at most one global subtransaction per site. A *local subtransaction* is a series of operations pertaining to a global subtransaction, performed at the local DBMS interface. The distinction between the global and local subtransactions is essential in achieving the ability to deal with transaction failures afflicted by the Participating DBMS, as will be shown in the sequel.

The commands of a transaction are not known a priori to the system when the transaction is started. In particular, the invocation and parameters of some commands may depend on the results of other operations.

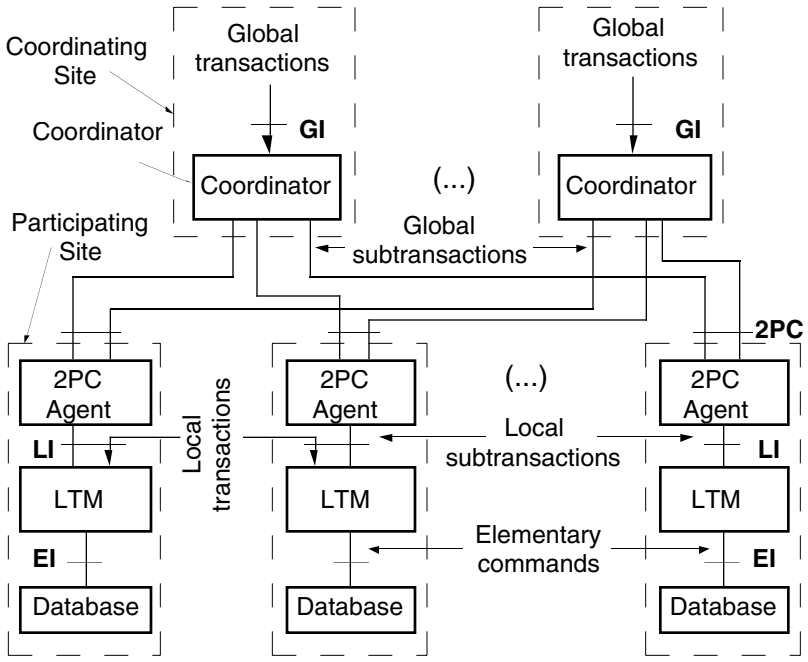


Fig. 1. Proposed architecture of a multidatabase transaction management system.

The assumptions about GI are as follows:

(GI.1) The interface supports high level database manipulation command (e.g. SQL DML: Select, Update, Insert and Delete) that are transformable (by each underlaying DBMS) to a sequence of Read and Write commands performed on elementary database objects;

(GI.2) Each database manipulation operation is explicitly related to a given Participating DBMS;

(GI.3) The interface supports transaction management commands Begintransaction, Commit and Abort.

The Coordinator decomposes global transactions into global subtransactions, submits the corresponding commands to the Participating Sites and returns the results to the application. Upon receiving the global Commit, it starts the distributed commitment procedure according to the basic 2PC protocol [17]. The Participant role is played by the 2PC Agent (2PCA) modules. There is a single instance of the 2PCA associated with a Participating DBMS. The connecting lines in Fig. 1 denote potential associations between the modules.

The functionality of the 2PC Agent interface is defined by the 2PC protocol. In particular, the Prepare command (PREPARE message) is a request to move the subtransaction to the prepared state. The response may either be affirmative (READY message) or negative (REFUSE message). The decision commands Commit or Abort (COMMIT or ABORT message) have to be accepted and acknowledged unconditionally.

The *Local Transaction Manager* (LTM) modules, represent the transactional aspects of the Participating DBMSs. There is a single LTM (and the corresponding 2PCA) per site. In fact, the concept of a site may be generalized into that of a computing environment associated with a given LTM.

There is an important premise that the LTMs are pre-existing systems and their design autonomy is maximally preserved.

A global transaction results in one or more local subtransactions seen at LI. In a failure-free operation there is only one local subtransaction per global transaction. Local transactions (which are not seen by the 2PCA) also enter the LTM at LI. The LTM can not differentiate between the local transactions and the local subtransactions.

In addition to the above components, for the sake of model completeness, we assume a hypothetical Elementary Interface (EI) where the elementary commands Read and Write deal with elementary database objects. The latter ones are the objects recognized by the specific concurrency control method applied by the LTM (e.g. having granularity used by the locking scheme). The transactional commands Begintransaction, Commit and Abort are as such not supported at this interface, but the LTM decomposes them into suitable elementary operations.

The main assumptions about the Local Interface are the following:

(1PC) There exists a single-phase transactional interface to the LTM supporting the same database commands as those at the GI level and the local commands Begintransaction, Commit and Abort.

(DLU) Denied Local Updates. Local transactions do not update objects read or written by a local subtransaction, while the corresponding global subtransaction is in the prepared state.

(RTT) Real Time Transparency. Any two identical sequences of data manipulation commands executed at arbitrary points of time produce the same results (in terms of database state changes and command responses) provided the database objects read by them have identical values in either case.

The LTMs have the following characteristics:

(SRS) Serializable and rigorous histories. LTM enforces histories that are conflict serializable, and *rigorous* meaning they are strict [3] and additionally such that no data object may be written until the transactions that previously read it commit or abort [15]. Typically, it is sufficient to have the strict locking policy (S2PL) whereby all objects accessed by a transaction are unavailable to other transactions until

Commit/Abort. For better clarity, the examples included in the paper, and some of the argumentation, are based on the assumption that LTMs apply the S2PL policy.

(UAN) Unilateral Abort Notification. If LTM unilaterally aborts a local subtransaction, this information is passed over to the 2PCA, through the LI interface, on the next synchronous command return.

(TW) Trustworthiness. After a fixed number of resubmissions, any global subtransaction that should be committed can be committed.

(LL) LTM Log. LTM maintains its own internal transaction log. By working off the log, LTM is capable of:

- rolling back any local (sub)transaction or a local transaction and
- undoing and redoing local (sub)transactions and local transactions during the (LTM-driven) site recovery, if necessary.

(RWMC) Read-Write model compatibility. LTM transforms the high level database manipulation commands into the series of the elementary Read and Write.commands [3]. The elementary commands are observable at a hypothetical Elementary Interface (EI).

Failure modes

We are considering two types of failures at Participating Sites:

- ***Site failure*** — a system failure (system crash) at a Participating Site. No program state is retained. Non-volatile repositories (logs) are available for recovery.
- ***Subtransaction failure*** — a local subtransaction is aborted unilaterally by the LTM. The state of the 2PC Agent and the LTM are preserved. This failure may occur either (or both) before or after achieving the prepared state by the global subtransaction.

We are not dealing with failures types that do not imply special treatment in an environment discussed, as compared to a homogeneous database system. Such failure types are, e.g.: lost or corrupted messages, messages arriving out of order, or any type of Coordinator failures. Also, no special attention need to be paid to subtransactions that have not reached the prepared state because they may be "legally" aborted anyway.

It is the responsibility of the 2PC Agent to recover all the transactions that had been in the prepared state as the failure occurred. In case of a site failure, this also includes restoring of the local execution order that corresponds to the global order.

Objectives of the 2PC Agent Method

The 2PCA method, under the assumptions stated above, assures the view serializability [3] of global histories. A simplified version of the method produces quasi serializable [8] histories[1].

[1] We use a slightly weaker notion of quasi serializability than that in [Du&Elmagarmid89]; To us, a history H is quasi serializable if its projection on global transactions $H[_g]$ is serializable and each local history $H[^i]$ is serializable.

Description of the 2PC Agent method

General

The prepared state recovery scheme is based on the idea of the *subtransaction resubmission* which is a repeated execution of all the commands belonging to a global subtransaction when a corresponding local subtransactions had been aborted by the LTM. A transaction resubmission results in a new local subtransaction. In the process of recovery, the 2PCA may generate many local subtransactions for a given global subtransaction. Then, all but the last one (in the history) are in the aborted state. The last one may be incomplete, aborted or committed.

Agent Log

A 2PC Agent maintains its own log (Agent Log) solely for the purpose of recovery of subtransactions that had reached the prepared state.

Every time a global subtransaction operation has been completed, the 2PCA writes the corresponding database language command to the log. Thus a log entry implicitly indicates successful lock reservations as well. The 2PCA also writes the PREPARED record before the READY message is sent.

Each time a PREPARED record is written, the Agent Log is forced to disk (or other stable storage) to facilitate site recovery. We call this *force writing* of the record.

When 2PCA receives the COMMIT message, it writes the COMMIT record in the log. This is to be performed as a part of the local subtransaction so that it can be committed together with the other operations of the subtransaction. This guarantees that either both the global decision and the local transaction result are recorded in the persistent storage, or none of them is recorded. This way of operation is easily achievable by implementing the Agent Log in the database under the control of the LTM.

Failure time concurrency control

In a failure-free operation, the serializability of global transactions is guaranteed by the strict two-phase locking (or other equivalent) mechanism of the associated LTMs. The mechanism fails if there are any unilateral aborts of prepared subtransactions. In case of a local failure it may re-schedule the transactions in such a way that the subtransaction execution order will change. For example, assume we have two global subtransactions executing at a given node *i:*

$T^i_1[x]$ — obtained lock on object x, and is in prepared state;

$T^i_2[x]$ — waiting for lock on object x.

The resulting execution order is: $T^i_1[x] < T^i_2[x]$ ("<" means "executed before") which complies with some global serialization order.

Now, assume T^i_1 is aborted by LTM. This has the consequence that T^i_1 is rolled back and the locks held by this transaction are released. Before 2PC Agent manages to recover the state of T^i_1 (i.e. resubmit T^i_1), LTM schedules T^i_2 for execution. The resulting execution order would be $T^i_2 < T^i_1$ which violates the global serialization order (a cycle appears in the serialization graph).

The main problem here is to preserve, at each site, the same local order among local subtransactions that would be attained by way of locking in the failure-free situation. There are at least two approaches to this problem — one *pessimistic* and the other *optimistic*.

In the pessimistic approach, the 2PC Agent has to maintain locks on database objects solely for the sake of recovery from subtransaction failures. As no 2PC Agent has control over physical database objects, a logical locking scheme has to be applied. Possible techniques are *predicate locks* [12], *precision locks* [20] or some coarse granularity methods like table locks.

Application of the pessimistic approach would mean duplicating of the locking mechanism existing in the LTM and a high performance penalty. Since, under the SRS property, the global histories in a failure-free situation are serializable without any intervention of the 2PC Agent, we are proposing an alternative optimistic approach here.

At first, the commands of an unprepared subtransaction are executed without any ordering activity of the 2PC Agent, relying on the local serialization mechanism. Then, at arrival of the PREPARE message, the 2PCA performs the *prepare certification*. The objective of the prepare certification is to ensure that the original local operation order has been preserved in spite of subtransaction failures. The prepare certification alone guarantees quasi serializability of global histories in the presence of failures. It is the basis of the *single-step certifier* of the 2PC Agent. Once a subtransaction has been prepare certified, it enters the prepared state. If it is rejected, the REFUSE response is issued or — as an optimization — the certification is retried later.

At the arrival of the COMMIT message, the 2PC Agent may also perform the second certification step — the *commit certification*. The goal of this step is to ensure that no cycles have been introduced into the global serialization graph by (read-only) local transactions executed during the subtransaction failures. The goal is achieved by delaying commitment of certain subtransactions. If a subtransaction is commit certified, it may be immediately committed. Otherwise it is scheduled for a repeated certification at some later point of time. A *two-step certifier* performing both the prepare and commit certification guarantees view serializability of global histories under the stated assumptions. Both certification methods are discussed below.

Prepare certification

The prepare certification is performed in such a way that the following invariant is not violated:

> ***2PCA Correctness Invariant:***
>
> a) no global subtransaction is moved to the prepared state if the corresponding local subtransaction is unilaterally aborted and
>
> b) no two global subtransactions have conflicting elementary database operations while they are in the prepared state.

As usual, a conflict takes place between the unprepared local subtransaction T^i_u and an another local subtransaction T^i_a at node *i* if:

(i) T^i_u reads or writes a local database object written by T^i_a at EI,

(ii) T^i_u writes an object read by T^i_a at EI.

The subtle issue in the definition of the conflict is that the elementary database objects and operations, and thus the definition of conflicts, might be different from site to site. Still, we assume that, at each site, the local system somehow orders its elementary database operations in such a way that the SRS property holds for the local history. Thus, a "locally conflict equiva-

lent" serial history can always be found. Since we only need a serialization *order* of *local subtransactions* at a site, the possible differences in the implementation of the concept of "conflict" between two different sites do not actually matter, as long as each implementation of the conflict is correct (i.e. each history is view serializable).

If the 2PCA Correctness Invariant can really be kept by the 2PC Agent, then the subtransactions can be committed without jeopardizing the serializability among global transactions no matter when and if the COMMIT message arrives.

It is basically possible to maintain the Correctness Invariant if only such global subtransactions are moved to the prepared state that do not have conflicting operations with the already prepared ones. How is it possible to find those global subtransactions? We say that a local subtransaction or a local transaction (both denoted in the sequel as "local (sub)transaction") is *active* if it is neither locally committed nor aborted. Then the following fact holds:

> ***2PCA Implementation Basis:***
>
> If two local (sub)transactions are active at the same time and they have completed all their operations, then they cannot have conflicting elementary database operations, provided that the Participant LTM produces SRS histories.

To see that the Implementation Basis is correct, let us assume there is some conflict. This is possible only if both conflicting transactions are active, or at least one is committed. The latter is in contradiction with the assumption that both are active. Let them both be active. A result from the conflict between active transactions under rigorousness is that at least one transaction must be waiting for some data object (or lock on it) to be released by the other one. This means necessarily that one transaction would not have completed all its operations, which is in contradiction with the assumption that all operations are completed. Thus, there cannot be conflicting operations.

The certifier using the Implementation Basis needs primarily to ascertain that there is a moment in the past at which both the prepared subtransaction to be tested and the local subtransaction to be certified have been active and have performed all their operations. If such a moment can be found with each of the prepared subtransactions separately, then the global transaction has passed the test and can be moved to the prepared state, otherwise the state transition is not allowed (at that moment).

Resubmissions of operations might cause the Correctness Invariant to be violated since the local database state could have been changed during a unilateral abort by local subtransactions or local transactions. This is, however, impossible: since the prepared original local subtransactions do not have conflicting elementary operations, and since the local database recovers correctly (cf. LL property), the resubmitted local subtransaction cannot access (under RTT) new database objects as compared to the originally accessed. Only intervening local transactions could cause the database state to evolve in such a way that conflicts might arise between resubmitted global subtransactions (that is, the read-write sets of the resubmitted local subtransactions might be different from the original ones). However, based on DLU it can be shown that the resubmitted commands result in a local subtransaction that is structurally identical to the original one. Thus, the Correctness Invariant can, at least in principle, be kept in the environment.

Technically, the certifier algorithm is based on *alive time intervals* rather than on distinct points of time. A global subtransaction is alive if the corresponding local subtransaction is active. The certifier keeps the (last) alive interval of each prepared global subtransaction in a centralized

data structure and performs interval intersection test between each prepared subtransaction and the one to be certified. If the 2PCA discovers a non-intersecting alive interval pair, there is possibly a conflict between the prepared subtransaction and the subtransaction T^i_u to be certified. Thus, the certification fails and the transaction T^i_u is aborted by 2PCA (the REFUSE message follows), or the certification is retried later. If there is a non-empty intersection with each prepared transaction, the certification is said to be successful and the subtransaction T^i_u also enters the prepared state.

How does the 2PC Agent know that a subtransaction is alive? The local subtransactions are regularly checked for a unilateral abort (based on the UAN property, this is possible) and the upper end of the alive intervals are updated at the same pace, as long as the local subtransaction is active. The upper end of the interval is not updated while the local subtransaction is unilaterally aborted and is being recovered.

Commit certification

When a COMMIT message arrives, the 2PC Agent should commit the local subtransaction. There is still one complication due to the local transactions. Even under DLU, they might cause an intersite cycle to emerge into the global serialization graph, if resubmissions and conflicts occur in a certain way. Example 3 of [6] illustrates such a case. In this example, two global transactions T_1 and T_2 operate on two sites *a* and *b* but they do not conflict directly with each other. There are also two local transaction T_a and T_b which have conflicting operations with both global transactions at sites *a* and *b*, respectively. Thus, T_1 and T_2 conflict *indirectly* [8] with each other. In a failure-free situation a possible global serialization order is: $T_1 < T_a < T_2 < T_b$. However, If T^b_1 gets unilaterally aborted while being in the prepared state (and subsequently recovered by the 2PCA), the local serialization order at site b could become $T^b_2 < T_b < T^b_1$. The result would be a global history being neither conflict nor view equivalent to any serial history.

The commit certification prevents this from happening. Generally, the approach is to delay the commitment of a global subtransaction (in our case, T^b_2) in case the commitment would potentially lead to an illegal local serialization order. One way is to derive a total order of global transaction commitments, and to enforce it upon the global subtransactions at each site. An arbitrary global order could, however, contradict with the order generated dynamically by the local SRS schedulers. Therefore, for any set of directly or indirectly conflicting global transactions such an order should be *analogous* [15] to the global serialization order established during normal (failure-free) operation of the system.

Once the 2PCA knows the correct order, it may autonomously decide whether a prepared subtransaction should be committed or not. If all the subtransactions that have lower order position were also committed, the subtransaction in question is commit certified and it will be immediately locally committed. Otherwise it will await repeated certification performed at a later point of time.

The problem is how to derive the order unambiguously at each site, and in time for the commit certification to be performed. Various synchronization techniques can be applied. Advantage can be taken of the fact that, for directly or indirectly conflicting global transactions, the corresponding global subtransactions enter — at each site — the prepared state in the order which is analogous to the global serialization order. An optimal ordering technique being a good tradeoff of complexity and restrictiveness is for further study.

Assume the order $T_1 < T_2$ is imposed on the global transactions in the example above. If the example was run in a lock based system, the effect of the commit certification would be in creating a deadlock illustrated by the following wait-for-graph [3]: $T^b_2 \rightarrow T^b_1 \rightarrow T_b \rightarrow T^b_2$. The first arc represents waiting for the correct commit order, imposed by the 2PCA. The other arcs represent a typical waiting for locked objects within the LTM. The deadlock has to be resolved by aborting T_b or T^b_2. Because neither LTM nor 2PCA can detect the deadlock autonomously, a timeout based deadlock resolution have to be used by LTM or 2PCA.

If the Coordinators send COMMIT messages synchronously (i.e. they wait for the acknowledgement before sending the next COMMIT message) a global deadlock involving Coordinators may also occur. This may be avoided by the following optimization. The COMMIT message is acknowledged immediately, regardless of the commit certification outcome. If the commit certification had failed, a PROMISED TO COMMIT record is force written into the Agent Log before acknowledging the COMMIT message. The record will facilitate the subtransaction state recovery in case of a site failure. Also, the response time of the global Commit command is improved with this implementation. The disadvantage of the approach is that the database objects affected by the transaction might not be actually released before the local commit is eventually performed.

Failure-free operation

The following is the summary of the 2PCA algorithm in a failure-free situation:

1. Accept the Begintransaction command and record them it the Agent Log.
2. Accept the database language commands, pass them to the LTM and record them in the Agent Log upon successful completion; maintain the timestamp of the last command completion (per subtransaction).
3. Accept the Prepare command; do the prepare certification; if the certification is successful check if the transaction is alive; if it is — force write the PREPARED record in the Agent Log, update the list of prepared transactions and set the limits of the alive interval as <last command timestamp,current timestamp>; if prepare certification fails or the subtransaction gets unilaterally aborted in the mean time, respond with REFUSE, clean the Agent Log and abort the subtransaction, if still active.
4. Periodically check whether the prepared subtransaction is alive; if it is alive update the alive interval accordingly, if not — start subtransaction recovery (see next section).
5. Accept the Commit command; do the commit certification; if the certification is successful remove the subtransaction from the prepare list, write the COMMIT record into the Agent Log, commit the subtransaction in the LTM, and acknowledge the COMMIT command to the Coordinator; if not successful — force write the PROMISED TO COMMIT record into the Agent Log, acknowledge the command to the Coordinator and repeatedly retry the certification later on until it is successful.
6. Repeatedly clean the Agent Log by removing all the committed transactions.

Recovery from subtransaction failures

If there has happened a local subtransaction unilateral abort then

(i) in the unprepared state: "forget" the subtransaction, leading effectively to the global transaction abort;

(ii) in the prepared state:

1. Resubmit (in the original order) the commands of the global subtransaction from the Agent Log.
2. Once all the operations of the aborted transaction are again completed, reinitialize the alive interval to just the point of time of the completion of the last recovered operation.
3. Resume normal operation.

Recovery from site failures

A participating site failure may be considered a massive subtransaction failure because LTM rolls back all uncommitted transactions during the LTM-driven site recovery, including the ones that had been in the prepared state before the failure occurred. Additionally, the volatile state of 2PCA is lost. The Agent Log contains, however, the specifications of all the prepared global subtransactions. The subtransactions that had been promised to commit are also recorded in the log.

We are concentrating on the subtransaction state recovery in the following. Consequently, we assume the connections with the Coordinators can be re-established and the Coordinators retransmit messages (e.g. PREPARE and COMMIT) infinitively if they are not responded to.

The recovery sequence is as follows:

1. Restart the LTM while disallowing local transactions.
2. Clean the Agent Log by removing data pertaining to unprepared and committed transactions.
3. Resubmit all the prepared subtransactions and the promised to commit transactions. If the transaction has a PROMISED TO COMMIT record in the Log, then commit it locally as above. Otherwise, initialize the alive intervals.
4. Enable connections to the Coordinators and allow for local transactions.
5. Resume normal operation.

The global subtransactions may be resubmitted in any order. Still, the global history would be view serializable. This is, in short, because:

(i) The local transactions are prohibited to access the local database while the 2PCA recovers. This Denied Local Access (DLA) restriction holds during the 2PCA recovery. Clearly, DLA implies DLU.

(ii) Since DLU is guaranteed to hold before and during the recovery, the Conflict Invariant is kept. Thus, no conflict is possible among the global subtransactions during the 2PCA recovery. Consequently, no new direct arcs between global transactions can emerge in the global serialization graph due to the recovery of a 2PCA.

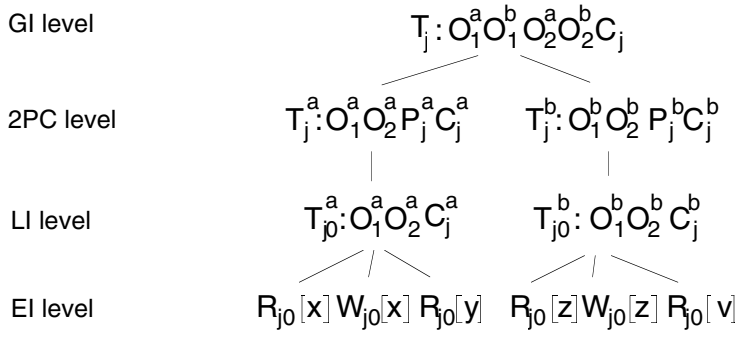
(iii) Thanks to the DLA, no local transaction can access the same data any resubmitted local subtransaction accesses during the 2PCA recovery. Thus, local transactions cannot cause new arcs to emerge into the global serialization graph during 2PCA recovery.

(iv) Since no new arcs can emerge into the serialization graph during 2PCA recovery, no new cycles can occur either. Thus, if the 2PCA method produces view serializable histories during failure-free operation and in the presence of subtransaction failures, then it produces them all the time.

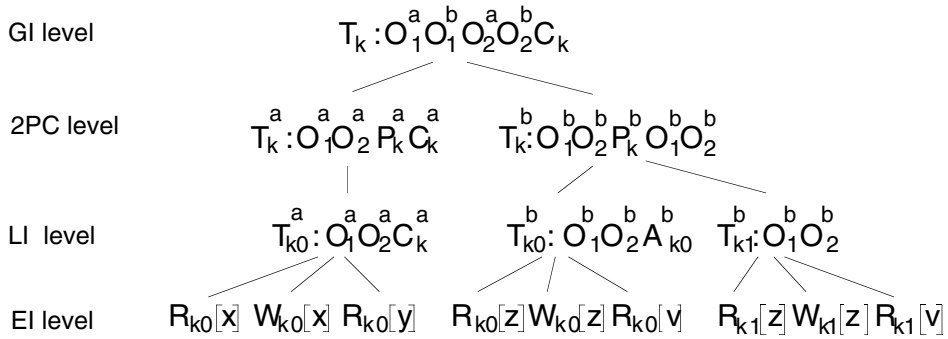
Correctness of the method

We shall use the model and the results of [3]. However, due to unilateral aborts and subsequent resubmissions of operations, there might be, in our case, an arbitrary number of local abort operations in a global transaction and they may occur *after* the Coordinator has decided to globally commit the global transaction. What is a transaction that is globally committed but that has locally aborted subtransactions? To us, it is globally committed but *incomplete*. It becomes complete due to resubmissions of operations from the 2PC Agent log and a subsequent local commit. In [3] the committed projection of a history, C(H), contains exactly all committed transactions in H. We include only the globally committed complete transactions into the committed projection. In addition, our C(H) includes all aborted local subtransactions that belong to globally committed transactions. We cannot simply ignore them in C(H) because a local abort causes data objects to be released for other global and local transactions in any phase of execution and this can lead to globally nonserializable or unrecoverable histories.

A transaction execution is modelled by means of a finite sequence of *execution trees*, and the results are proven for any execution tree. Each individual tree is a snapshot of a certain phase of the execution of one transaction and it is contained in the tree modelling the next phase. An example of an execution tree of a complete committed and incomplete committed transaction is shown in Fig. 2a and 2b, respectively.



(a)



(b)

Fig. 2. Illustration of (a) a committed and complete global transaction Tj and (b) committed and incomplete global transaction Tk with an aborted local subtransaction.

If an operation is listed in a node of an execution tree, this indicates that the operation has been completely executed at a given interface and at the interfaces below it. The only exceptions are the global C (Commit) and A (Abort) operations, one of which occurs in the root node, whenever the Coordinator has got the global abort or commit request from the application and recorded the decision in the Coordinator log. The leaf level of the tree, i.e. the operations of the local (sub)transactions, represents a collection of traditional transaction histories [3], each consisting of a sequence of R (Read) and W (Write) operations.

Fig. 2b, for instance, models a global transaction T_k, whose global Commit operation has not been successfully acted upon yet, and hence it is seen at the GI level but not at the 2PC level. In the mean time the original local subtransaction T^b_{k0} had become aborted and the 2PCA resubmitted it in the form of T^b_{k1}. The latter local subtransaction is locally neither committed nor aborted, which makes T_k incomplete.

The example above shows that we must model the local commits and aborts separately from global commits and aborts. Distinction between global and local commits and aborts is to us also necessary due to the need to model the purely local transactions separately from the global ones. Further, we must mark explicitly the prepared global subtransactions with P operations to be able to reason about the properties of the overall system. A P operation occurs in a 2PCA node of an execution tree, if and only if the status of the corresponding global subtransaction has been recorded permanently in the 2PCA log.

Because of these changes we must redefine the basic concepts like conflict or view equivalence and the corresponding serializability concepts in the spirit of [3].

> ***Notations:*** T_j denotes the jth transaction, C_j the global commit of T_j at the root, A_j the global abort at the root. C^i_j denotes both the local commit and the commit at the 2PCA at site i. P^i_j denotes the prepare operation of the global subtransaction T^i_j at ith 2PCA. Finally, A^i_{jk} is the local abort of the kth local subtransaction of T_j at site i. $R_{jk}[X^i]$ ($W_{jk}[X^i]$) denotes a read (write) operation, that accesses entity X^i and that belongs to local (sub)transaction T^i_{jk}.❑
>
> ***Definition 1: Histories.*** A transaction history $H(T_j)$ is the sequence of R, and W operations at the leaves of an execution tree T_j, enhanced with C_j, C^i_j, A_j, A^i_{jk} and P^i_j operations if they occur higher in a node of the execution tree. The order of operations in the transaction history is consistent with their real time execution order. A (global) history H is an interleaving of a finite collection of transaction histories $H(T_1)$, $H(T_2)$,...,$H(T_k)$.❑
>
> ***Definition 2: Serialization graph SG(H).*** Let H be a history. SG(H) is a directed graph whose nodes are the global and local transactions T_h, T_j,... occurring in C(H) and there is an arc from T_h to T_j, $j \neq h$, iff there is a pair of conflicting operations ($R_{hk}[X^i] <_H W_{jt}[X^i]$ or $W_{hk}[X^i] <_H W_{jt}[X^i]$) in H.❑

Together with the existing theory and new definitions we can prove the needed results. We assume that local systems produce conflict serializable and rigorous histories. At each local site, that is, in any restriction of a global history H onto ith site, $H[^i]$, we can directly use the results of [3]. Our main concern is then to show that the global system only produces conflict serializable histories. The result can be shown only if there are no unilateral aborts after a 2PCA has voted READY for a global subtransaction. If there are unilateral aborts of the global subtransactions in the prepared state, the conflict serializability cannot be proven in general any more. The conflict serializability can still be proven for a restriction of any history H onto global transactions, $H[_g]$, provided DLU is obeyed. Because the local histories $H[^i]$ are rigorous

and serializable, the result means that at any case 2PCA method yields quasi serializable histories.

The cyclicity of serialization graphs SG(H) in general also makes it questionable whether resubmitting operations makes any sense or under what conditions it makes sense. What are the reasonable local transactions? Using the DLU assumption to restrict local transactions and assuming that local systems only produce conflict serializable and rigorous histories it is possible to show that all histories emerging are view serializable in spite of cyclicity of SG(H). This is our main result. It makes the strategy to use resubmissions reasonable but also illustrates the most severe limitation of the method. Here we only sketch the results, detailed definitions and proofs can be found in [33].

Theorem 1: Serializability theorem. A history H is serializable iff SG(H) is acyclic.

Proof: We can use directly the theorem 2.1 of [3], since we can define the concepts in a similar way. The only difference is that the definition of conflicts above includes site identity.❑

We continue by showing that using local aborting is appropriate in any phase of global or local transaction. The theorem is the basis of the implementations that use local aborts to resolve any problems like deadlocks, program errors, etc. It also shows that resubmitting of operations is safe at least as long as they are not committed.

Theorem 2. Let H be a serializable history and H´ be a history that is formed from H by adding some R, W, or A operations into it so that H is a prefix of H´. Then H´ is also conflict serializable.

Proof sketch: Follows from the fact the serialization graph SG(H) does not change if neither new committed transactions are added into H in H´, nor new operations are added to transactions committed in H. Thus SG(H) = SG(H´) and Theorem 1 yields the result.❑

Unfortunately, one cannot include arbitrary local commit operations into H´ so that SG(H´) still remains acyclic. The Coordinators and LTMs should, of course, commit only such transactions that the acyclicity of SG(H´) is preserved. We call a transaction *simple* if it is a local transaction or if it is a global transaction that has at most one local subtransaction at a site. If no local subtransaction T^i_{j0} was unilaterally aborted after the global subtransaction T^i_j was moved into the prepared state at any site i where T_j has a global subtransaction, no resubmissions are needed and the whole global transaction T_j remains simple (see Fig.3a). The following theorem shows that the overall system produces conflict serializable histories as long as no unilateral abort happens after the ith 2PC Agent moved the local subtransaction to a prepared state (P^i_j occurs in H).

Theorem 3. Let H be a history. If all globally committed transactions in H are simple and each local system produces only locally serializable and rigorous histories then H is globally conflict serializable.

Proof sketch: Each local transaction is by definition simple and each globally committed transaction has this property by assumption. Thus, for any committed completed global transaction there is at most one local subtransaction at each site. Based on the rigorousness and serializability of the local histories we can show that the global serialization graph consists of acyclic subgraphs of the local serialization graphs $SG^i(H)$ defined over locally committed (sub)transactions only. Assuming that there is a cycle in the union of the graphs, i.e. in SG(H), leads to a contradiction with respect to the order of global commits in H (cf. [5], or p. 78 of [3]).❑

Note that a sufficient condition for the rigorousness and serializability of local history $H[^i]$ is that the ith local system uses strict 2PL (see [3], p. 78). It is not, however, a necessary condition. Any other method, or combination of methods that guarantees local serializability and rigorousness, suffices.

The previous two results are the strongest general ones that we can achieve in the sense that no restrictions are enforced upon transactions. If we drop the fairly unrealistic assumption that unilateral aborts do not occur after a global subtransaction T^i_j has been moved to the prepared state (i.e. after a P^i_j operation in H) then the conflict serializability of an arbitrary history H cannot be any more proven. For this reason we restrict local transactions by DLU. Even under DLU there might be cycles in the SG(H), that is, the system does not produce conflict serializable histories. But maybe they still are view serializable? This is indeed true under DLU.

To show that the 2PC Agent method really makes some sense in the presence of arbitrary unilateral aborts we show that, under DLU, the global histories are view serializable. To proceed to this result let us first prove that ignoring the local transactions (or prohibiting them or submitting them as global ones) yields serializable global restrictions, $H[_g]$, of histories H onto global transactions.

> ***Theorem 4.*** Let H be a history and the restrictions of it into local histories, $H[^i]$, be locally serializable and rigorous. Under 2PCA policy the restriction of H onto global transactions, $H[_g]$, is conflict serializable, i.e. $SG(H[_g])$ is acyclic.

Proof sketch: We drop the local transactions from the history H and look at the reduced global history $H[_g]$. On the basis of the properties of the 2PCA and local LTMs and DLU we show that no cycles can emerge into the serialization graph $SG(H[_g])$, even if resubmissions occur. The reason is that 2PC Agent maintains the Correctness Invariant and thus prohibits such global subtransactions to be moved to the prepared state that have conflicting elementary operations with the already prepared subtransactions at that site. Consequently, when a prepared subtransaction is committed, it cannot have conflicting elementary database operations with any other global subtransaction that was simultaneously in the prepared state at that site. Hence, there cannot be cycles due to such transactions in $SG(H[_g])$. Assuming other cycles in $SG(H[_g])$ leads to the same contradiction as in the proof of Theorem 3. ❑

> ***Theorem 5.*** If H is conflict serializable then it is view serializable.

Proof sketch: We can argument exactly like in Th 2.4. of [3], in spite that the definitions had to be modified.❑

> ***Theorem 6.*** If local transactions obey DLU and local systems produce locally rigorous and conflict serializable histories then 2PC Agent method produces view serializable global histories.

Proof sketch: Let H´ be an arbitrary prefix of H and SG(H´) be the serialization graph of H´. If it is acyclic the claim follows from theorem 5 for it and all its prefixes. Let us assume that there is a cycle in SG(H´). Our strategy is to show that in spite of the cycle in SG(H´), we are able to find a serial history H´s that is view equivalent to C(H´). To find this we reduce C(H´) to $C^{1j}(H´)$ in such a way that all aborted local (sub)transactions are removed. Consequently, all intrasite cycles among global subtransactions and local subtransactions at a site, caused by resubmissions, disappear from $SG(C^{1j}(H´))$. The prepare certification guarantees that no two subtransactions can reach prepared state at the same time at any site if they have conflicting operations in the original local subtransaction. From DLU it follows that they will neither have conflicting operations in their possible resubmissions. Together these facts guarantee that the

Correctness Invariant holds, which in turn means that no cycles can occur in the restriction of H onto the global transactions, $H[_g]$.

The commit certification orders the local commits of the local transactions into the same order at each each site. This prohibits intersite cycles which involve at least two global transactions and at least one local transaction at a site in $SG(C^{1j}(H'))$. Thus, the two-step certifier guarantees that there are no cycles at all in $SG(C^{1j}(H'))$ and a topological sort of it can be performed. By theorem 5, the reduced history $C^{1j}(H')$ is view equivalent to a resulting sorted serial one, $C^{ij}(H's)$. After that, we add the removed subtransactions back to both histories to get H´s and C(H´). The conflict order of local transactions and local committed subtransactions cannot be different in the four histories due to local serializability and rigorousness. Based on this we show that the global reads-form relation remains identical in C(H´) and H´s, since it is identical with that of the reduced histories. The argumentation is based on DLU. The final writes also remain the same in both histories as each added write becomes later aborted and thus cannot change the final values.❑

The previous result shows that forward recovery (resubmissions) used by the 2PCA is reasonable under DLU. It allows globally written data to be read by local transactions but not updated while a global subtransaction is in the prepared state. Let us look at Fig. 2b. It is easy to show that if we did not restrict in any way the local transactions, then the view serializability cannot be proven for histories with a cyclic SG(H). The reason is that the resubmitted operations, e.g. the local subtransaction T^b_{k1} in Fig 3b, might read other values than the first submission, T^b_{k0}, did since an intervening local transaction, say T_l, would have been able to update a local object between T^b_{k0} and T^b_{k1}. Therefore, T^b_{k1} gets another view than T^b_{k0}, and consequently T^b_{k1} might also update the local database in a way that differs from that of T^b_{k0}.

Remaining problems

We did not address here the problem of when does a transaction run to end. As was shown in [16] global and local transactions might deadlock so that neither a global nor local deadlock detection is possible. Theorem 2 above shows that it is safe to abort any transaction on the basis of time-out or some other criteria by the 2PC Agent or by the local system at any time — even if the global subtransaction was in the prepared state and already globally committed. It is especially important to understand that also site failures lead to such a local abort, as a side-effect. Thus, a sufficient condition for hidden deadlocks [16] to be resolved is that the sites fail from time to time — or that 2PC Agent aborts local subtransactions that wait too long. To show that global and local transactions really run into completion we have to show further that no arbitrary many livelocks occur. Actually, TW assumption guarantees this. The occasional site failures also suffice to guarantee that Coordinator failures do not block local or global transactions forever. The overall issue of termination is studied more closely in [33].

We did not address here the problems of optimizing the certifier. There are basically three possibilities. The certifier could store all alive intervals of subtransaction instead of only the last one and use any of them. Another idea is that the prepare certification can be retried several times if it fails. Under which condition this makes sense is for further study. Third possibility is to check explicitly conflicts between operations of the subtransactions if the prepare certification fails between the prepared subtransaction and that to be certified. This optimization functions correctly only if the local conflicts can really be deduced by looking only at the commands known to the 2PCA. However, usage of triggers at the local database make this assump-

tion invalid in general. It is for further study, what would be a suitable conflict detection mechanism based on the logged commands, if the other prerequisites would be fulfilled.

Can the DLU assumption be relieved? Evidently, but then we have to look at the semantics of transactions. By removing the DLU restriction totally, we would arrive at the situation where the method produces locally serializable histories in all situations. In order for such global histories to be meaningful, the semantics of the transactions would have to be restricted as compared to the general case.

There are certain performance deficiencies related to the proposed architecture and method when compared to a homogeneous system. The prepared state certifier consumes some additional resources, especially when a site is in a state of failure (unilateral transaction abort). Additionally, not all types of read-related optimization [26] of the 2PC/2PL protocol are possible. The read locks can not be released at the reception of the PREPARE message because the 2PCA has no explicit control of the database locks maintained by the LTM. However, the read-only global subtransactions may be recognized and committed by the 2PCA at the reception of the PREPARE message.

Prototype implementation

A prototype multidatabase transaction management system called HERMES has been implemented at the Laboratory for Information Processing of the Technical Research Centre of Finland (VTT) in Helsinki. HERMES includes the two-step certifier based 2PC Agent software. It is associated with the INGRES DBMS (Ingres Corp.) playing the role a single-phased database system. The other DBMS is the two-phased SQL Server (Sybase Inc.). The database access protocol of the SQL Server product family is used in the system, together with the related network interface software. The system has been implemented on Sun (Sun Microsystems, Inc.) and VAX (DEC) computers in a mixed UNIX and VMS operating systems environment.

Conclusions

The 2PC Agent Method enables to integrate ordinary database systems with a single phased transactional interface into a multidatabase system in which the Coordinators can run a standard two-phase commit protocol. The basic assumptions of the local systems are that they produce locally serializable and rigorous histories and that the transactional interface allows database manipulation operations to be dynamically generated. Local systems need not be modified in any way, and unilateral aborts and site failures are allowed in any phase of execution. Unilateral aborts are remedied by resubmitting the aborted operations by the 2PC Agent from its log. The local transactions are restricted in such a way that they do not update the data retrieved or updated by the global transactions being in the prepared state (the DLU assumption).

In spite of the unilateral aborts and otherwise arbitrary local transactions, the method guarantees conflict serializability among global transactions. The overall histories produced are view serializable if the two-step certifier (including both the prepare and commit certification) is used, albeit they are not necessarily conflict serializable. Any history is quasi serializable using the single-step certifier (prepare certification only).

The significant cost of the method is represented by two log force-writes per transaction at a Participant Site with a single-step certifier. This is the same as in a conventional 2PC imple-

mentation. The two-step certifier requires occasionally an additional force write per transaction at some sites. All the algorithms of the 2PC Agent method are of polynomial complexity.

The major advantage of the method, as compared to the approach of Breitbart, Silberschatz and Thomson [6] is that it does not call for any centralized transaction management or recovery activity in a multidatabase system

Acknowledgments

The authors wish to thank all the people who gave critical comments to the draft paper and provided encouragement to continue the work. Special thanks go to Stefano Ceri and Jim Gray, who helped to clarify the problems and the objectives in the early phase of the work, and to Gomer Thomas for pointing out a deficiency in the original PARBASE-90 version of the paper.

Bibliography

[1] R. Agrawal and D. J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation", *ACM TODS*, vol. 10, no. 4 (December 1985), pp. 529-564.

[2] R. Alonso, H. Garcia-Molina and K. Salem, "Concurrency Control and Recovery for Global Procedures in Federated Database Systems", *Quarterly Bull. IEEE Tech. Comm. on Database Engineering*, vol. 10, no. 3 (September 1987), pp. 5-11.

[3] P. A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency control and recovery in database systems", Addison-Wesley Publishing Company, 1987.

[4] Y. Breitbart and A. Silberschatz, "Multidatabase Update Issues", *Proc.1988 ACM SIGMOD Conf.* (Chicago, 1 - 3 June), pp. 135 - 142.

[5] Y. Breitbart and A. Silberschatz, "Multidatabase Systems with a Decentralized Concurrency Control Scheme", *IEEE Distributed Processing Tech. Comm. Newsletter*, vol. 10, no. 2 (November 1988), pp. 35-41.

[6] Y. Breitbart, A. Silberschatz and G. R. Thompson, "Reliable Transaction Management in a Multidatabase System", *Proc. 1990 ACM SIGMOD Conf.* (Atlantic City, 23 - 25 May), pp. 215 - 224.

[7] Y. Breitbart, "Multidatabase interoperability", *SIGMOD Record*, vol. 19, no. 3 (September 1990), pp. 52 - 60.

[8] W. Du and A. K. Elmagarmid, "Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase", *Proc. 15th VLDB Conf.* (Amsterdam, 22 - 25 August, 1989) pp. 347 - 355.

[9] W. Du, A. K. Elmagarmid, Y Leu and S.D. Osterman, "Effects of Local Autonomy on Global Concurrency Control in Heterogeneous Distributed Database Systems", *Proc. Second Int. Conf. on Data and Knowledge Systems for Manufacturing and Engineering*, October 1989.

[10] A. K. Elmagarmid and W. Du, "A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems", *Proc. IEEE Sixth Int. Conf. on Data Engineering* (Los Angeles, 5 - 9 February, 1990), pp. 37 - 46.

[11] A. K. Elmagarmid and A. A. Helal, "Supporting Updates in Heterogeneous Database Systems", *Proc. IEEE Fourth Conf. Data Engineering* (Los Angeles, 1 - 5 February, 1988), pp. 564 - 569.

[12] K. Eswaran, J. N. Gray, R. Lorie and I. Traiger, "On the Notions of Consistency and Predicate Locks in a Database System", *Comm. ACM*, vol. 19, no. 11 (November 1976), pp. 624-633.

[13] H. Garcia-Molina and B. Kogan, "Node Autonomy in Distributed Systems", *Proc. IEEE Int. Symp. on Databases in Parallel and Distributed Systems* (Austin, 5 - 7 December, 1988), pp. 158 - 166.

[14] H. Garcia-Molina and K. Salem, "Sagas", *Proc. 1987 ACM SIGMOD Conf.* (San Francisco, 27 - 29 May), pp. 249 - 251.

[15] D. Georgakopoulos, M. Rusinkiewicz and A. Sheth, "On Serializability of Multidatabase Transactions through Forced Local Conflict", *Proc. IEEE Seventh Intl. Conf. on Data Engineering* (Kobe, 8 - 12 April, 1991).

[16] V. Gligor and R. Popescu-Zeletin, "Transaction Management in Distributed Heterogeneous Database Management Systems", *Information Systems*, vol. 11, no. 4 (1986), pp. 287-297.

[17] J. N. Gray, "Notes on database operating systems", in: R. Bayer.et al.(eds.), *Operating Systems: An Advanced Cource*. Berlin, Springer-Verlag 1978, pp.393-481.

[18] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Comp. Surveys*, vol. 15. no. 4 (December 1983), pp. 287-317.

[19] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management", *ACM Trans. on Office Inf. Sys.*, vol. 3, no. 3 (1985), pp. 253-278.

[20] J.R. Jordan, J. Banerjee and R.B. Batman, "Precision Locks", *Proc. 1981 ACM SIGMOD Conf.* (Ann Arbor, 29 April - 1 May), pp. 143-147.

[21] Y. Kambayashi, "Integration of Different Concurrency Control Mechanisms in Heterogeneous Distributed Databases", *Proc. Second Int. Symp. on Interoperable Information Systems (INTAP)*, 1988, pp. 313-320.

[22] H. F. Korth, E. Levy and A. Silberschatz, "Compensating Transactions: A New Recovery Paradigm", *Proc. 16th VLDB Conf.* (Brisbane, 13 - 16 August, 1990).

[23] W. Litwin and A. Abdellatif, "Multidatabase Interoperability", *IEEE Computer*, vol. 19, no. 12 (December 1986), pp. 10-18.

[24] W. Litwin, H. Tirri, "Flexible Concurency Control using Value Dates", *IEEE Distributed Processing Tech. Comm. Newsletter*, vol. 10, no. 2 (November 1988), pp. 42-49.

[25] W. Litwin and A. Zeroual, "Advances in Multidatabase Systems", in: *Research into Networks and Distributed Applications (Proc. EUTECO '88, Conf.)*, R. Speth (ed.), Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 1137-1151.

[26] C. Mohan, B. Lindsay and R. Obermarck, "Transaction Management in the R* Distributed Database Management System", *ACM TODS*, vol. 11, no. 4 (December 1986), pp. 378-396.

[27] C. Pu, "Superdatabases for Composition of Heterogeneous Databases", *Proc. IEEE Fourth Conf. Data Engineering* (Los Angeles, 1 - 5 February, 1988), pp. 548 - 555.

[28] M.E. Rusinkiewicz, A.K. Elmagarmid, Y.Leu and W. Litwin, "Extending the Transaction Model to Capture more Meaning", *ACM SIGMOD Record*, vol. 19, no. 1 (March 1990), pp. 3-7.

[29] K. Salem, H. Garcia-Molina and R. Alonso, "Altruistic Locking: A Strategy for Coping with Long Lived Transactions", CS-TR-087-87, Department of Computer Science, Princeton University, 1987.

[30] K.Sugihara, "Concurrency Control Based on Distributed Cycle Detection", *Proc. IEEE Third Int. Conf. Data Engineering* (Los Angeles, 3 - 5 February, 1987), pp. 267 - 274.

[31] J. Veijalainen and R. Popescu-Zeletin, "Multidatabase Systems in ISO/OSI Environment", in: *Standards in Infomation Technology and Industrial Control* (Proc. Workshop on Standards and Economic Development in Information Technologies, Athens, April 14-16, 1986), N.E. Malagardis and T.J. Williams (eds.), Elsevier Science Publishers B.V. (North-Holland), 1988, pp. 83-96.

[32] J. Veijalainen, "Transaction Concepts in Autonomous Database Environments", GMD-Bericht Nr. 183, Munich, Germany, Oldenbourg Verlag 1990 (Ph.D. thesis).

[33] J. Veijalainen and A. Wolski, "The 2PC Agent Method and its Correctness", Research Notes no. 1192, Technical Research Centre of Finland, 1990.