Prepare and Commit Certification for Decentralized Transaction Management in Rigorous Heterogeneous Multidatabases

Jari Veijalainen
(Veijalainen@tik.vtt.fi)

Antoni Wolski
(Wolski@tik.vtt.fi)

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

Abstract

Algorithms for scheduling of distributed transactions in a heterogeneous multidatabase, in the presence of failures, are presented. The algorithms of prepare certification and commit certification protect against serialization errors called global view distortions and local view distortions. View serializable overall histories are guaranteed in the presence of most typical failures. The assumptions are, among others, that the participating database systems produce rigorous histories, e.g. by using the strict two-phase locking policy, and that no local transaction may update the data accessed by a global transaction that is in the prepared state. The main advantage of the method, as compared to other known solutions, is that it is totally decentralized.

1 Introduction

A *multidatabase system* can be seen as a system achieving some degree of interoperability of pre-existing local database systems (LDBS). For an overview of issues in multidatabases see e.g. [10] and [7]. In the following, we shall concentrate on the problem of how atomicity, consistency, isolation and durability, i.e. the ACID properties [16] of *transactions* could be preserved in a *heterogeneous multidatabase system* (HMDBS). Especially we undertake to satisfy the above quality requirements for a mix of global and local transactions submitted to a multidatabase, in the presence typical failures (transaction aborts).

It is characteristic for the multidatabase systems that the local database systems retain their autonomy, especially the *design* and *execution autonomy* (D-autonomy and E-autonomy) [12, 20]. Resulting from the D-autonomy, the LDBS's may be *heterogeneous* in many respects. From our point of view, heterogeneity means that the implementation of the database commands, like the SQL commands SELECT, UPDATE, DELETE, INSERT, COMMIT, or ROLLBACK, is different at different LDBS, and not fully known to the constructor of a HMDBS.

In order to maximally preserve D-autonomy of the LDBS, the commands available at the local interface (LI) of a LDBS should not be modified or new commands added. We only assume that each LDBS offers, at its LI, a full set of data manipulation (e.g. SQL) commands, including transaction management commands that together correspond to the conventional Commit, and Abort [5].

It seems very difficult to achieve the ACID properties of global transactions only assuming that each LDBS allows serializable histories [9].

As was shown in [9], in a failure-free situation, global histories containing arbitrary local and global transactions are conflict serializable if the LDBS's apply *rigorous* schedulers. Rigorous means serializable and *strict* in the sense of [5], and furthermore such that no data object may be written until the transaction that previously read it commits or aborts [9]. Guided by this, we assume the LDBS's produce rigorous (SRS) local histories. The rigorousness is, for example, achieved by the strict two-phase locking (S2PL) policy [5] whereby all the locks are kept until the transaction terminates. Since commercial database management systems usually use this policy, the SRS requirement does not reduce the D-autonomy too much.

Unfortunately, we cannot assume that no failures happen at an LDBS. Preserving D- and E-autonomy of an LDBS means that it can roll back a single transaction at any time. We call such an event *unilateral abort,* without making difference between single and collective abort (i.e. site crash). This may happen, in a real system, even after all the database commands have been executed. The reasons are various implementation-dependent issues, like the log buffer overflow (INGRES), or unexpected system bugs.

Globally, a unilateral abort means that an LDBS can *refuse to* execute a COMMIT message. This possibility jeopardizes the atomicity of the global Commit. A solution is to use the two-phase commit protocol (2PC) [17] within the HMDBS. If 2PC is used, the *prepared state* of the subtransactions has to be implemented. If all the LDBS's in the system support the 2PC interface and the prepared state, the stated goals can be easily met. We direct our efforts towards accommodating traditional, non-2PC LDBS's in the HMDBS. We believe that such systems will exist for a long time, in particular systems based on the older database technology. Consequently, we do not assume any particular data model in our algorithms.

The main problem is then, how to design a correct *Distributed Transaction Manager* (DTM) [15], on top of D- and E-autonomous LDBS's that do not support the prepared state. Architecturally, our DTM is based on the *2PC Agent Method* we presented earlier [22] wherein the concepts of an *agent*, *subtransaction resubmission* and *subtransaction certification* were introduced. In this paper, we propose new Certifier algorithms that represent an improvement in the sense of generality and technical implementability over previously known methods.

Section 2 includes a short presentation of the 2PC Agent architecture. The transaction model is discussed in section 3. The new algorithms are presented in sections 4 and 5. In section 6 we review related works. A few notes on the prototype implementation are given in section 7. The Certifier algorithms are summarized in the Appendix.

2 The Distributed Transaction Manager architecture

We present the proposed architecture and the assumptions briefly. For a detailed definition, see [22]. The architecture model is illustrated in Fig. 1. The DTM consists of *Coordinators* located at the *Coordinating Sites* and the *2PC Agent* (2PCA) modules located at the *Participating Sites* in connection with the *Local Transaction Managers* (LTM). An LTM represents the transactional aspects of an LDBS. There is a single instance of the 2PCA associated with an LTM.

The 2PC protocol involves a Coordinator and Participants. The Participant role is played by the 2PCA modules in the proposed system. The Coordinator sends BEGIN, PREPARE and COMMIT (or ROLLBACK) messages. The Participant may send READY or REFUSE in response to PREPARE, and it acknowledges the Coordinator's decision messages with COMMIT-ACK or ROLLBACK-ACK. The states of the Participant are the *idle* state (before receiving BEGIN), the *active* state (between receiving BEGIN and transmitting READY or REFUSE) and the *prepared* state (between transmitting READY and responding to COMMIT or ROLLBACK). The data manipulation commands are sent while the Participants are in the active state. While the Participant is in the prepared state, the data accessed by the transaction are called *bound data.*

The 2PC messages travel through a medium called Network. In order not to have to deal with failures of purely telecommunications nature, we assume the messages are nor corrupted, lost or out of order.

Functionally, the Coordinator decomposes global transactions into *global subtransactions*, submits them, command by command, to the Participating Sites (at most one global subtransaction per site) and returns the results to the application which performs the necessary computation. The global subtransactions contain database manipulation commands (e.g. SQL) only. Upon receiving the global Commit from the application, the Coordinator starts the standard 2PC distributed commitment protocol.

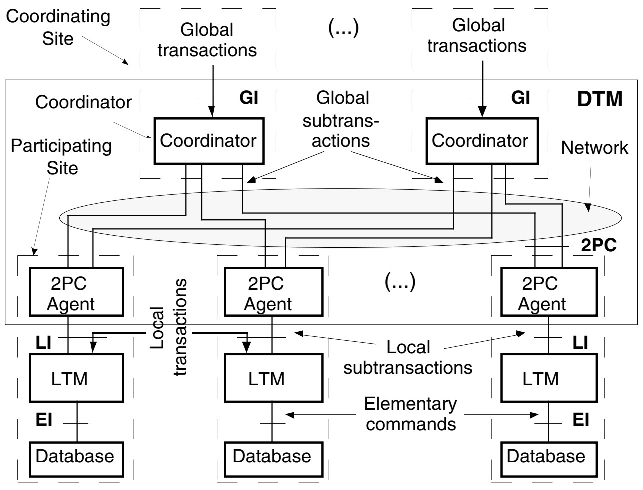


Fig. 1. An architecture of a multidatabase transaction management system.

The 2PCA decomposes the global subtransactions into *local subtransactions* visible at the local interface (LI). The LTM further decomposes the local subtransactions to the elementary Read and Write commands observable at the elementary interface (EI).

The essential assumptions about the LTM are the following:

DDF Deterministic Decomposition Function. The LTM transforms the high level database manipulation commands O^i into a sequence of elementary commands R and W. There is a time-independent deterministic decomposition function $D(O^i,S^i)$

defined over the set of all DML commands $\mathbf{O}^i$ applicable at ith database, and set of concrete[1] database states $\mathbf{S}^i$ of the ith database.

RR Rollback Recovery. If a transaction is aborted, the LTM restores the concrete before images for all data items affected by the transaction.

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 data items read by them have identical values in either case.

SRS Serializable and Rigorous histories. The LTM produces rigorous histories.

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

UAN Unilateral Abort Notification. The 2PCA is notified about any unilateral abort that has happened.

A *local transaction* is a transaction submitted directly to the LTM. The DTM has no knowledge of local transactions. An important assumption about the local transactions is as follows:

DLU Denied Local Updates. If a data item belongs to bound data of a global transaction, no local transaction may update it, albeit it may read it.

The main principle of the 2PCA method is to maintain the prepared state within the 2PCA, on behalf of the LTM. If a global subtransaction being in the prepared state is unilaterally aborted, the 2PCA submits a new *local subtransaction* expressed by the same commands (e.g. SQL) as the ones originally submitted. This is called the *subtransaction resubmission.* Note that the application specific computation is done by the application at the Coordinating Site before the global Commit is issued, and it is not affected by the resubmission. For that reason we require that the decomposition of a command is preserved when the command is resubmitted.

Because the bound data of the subtransaction are released by the LTM for the time period between the failure occurs and the commands are resubmitted, this may cause serializability errors illustrated in the sequel. To avoid the errors, we propose one or two *subtransaction certification* steps to be performed by the 2PCA, whereby the potential concurrency anomalies induced by the failure are detected.

The *2PCA Certifier* we outlined in [22] assumed that the conflicts detection would be based on the knowledge of the commands visible at 2PC interface, presumably by way of a predicate calculation of some kind. We show below that the conflict detection can be based solely on the SRS assumption, and that the algorithm is LTM-independent and very general. We also address below the *indirect conflicts* of the type shown in [11] and [8].

[1] For concrete/abstract states see [20]

3 The transaction model

Unilateral aborts are the source of all problems addressed by our method. If no unilateral aborts of prepared local subtransactions occur, then no anomalies can occur [22]. If the 2PCA detects a unilateral abort, it resubmits all the commands of the global subtransaction from its log (Agent log), thus creating a new local subtransaction.

The original and each resubmitted local subtransaction appears as an independent transaction to the LTM which treats them accordingly. From the global serializability point of view, however, they belong to the same transaction.

We use *execution trees* as modeling instruments, to elucidate the essential structure of the transactions in this architecture, similarly to [1] and [4].

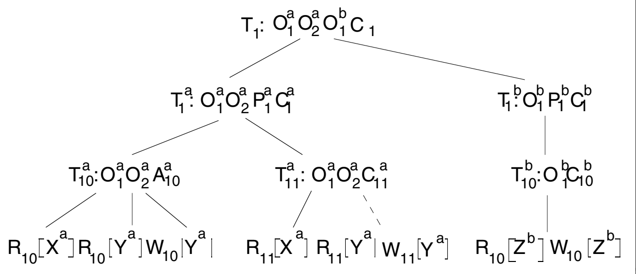
The leaf level of the tree consists of indexed R and W operations, produced by the LTM from the DML commands, as prescribed by the decomposition function D (cf. the DDF assumption). Thus, e.g. $R_{ik}[X^s]$ denotes a Read operation of the ith global transaction and the kth resubmitted local subtransaction which accesses data item X^s, at site *s*. The resubmission index is omitted in local transactions, e.g. $R_i[X^s]$.

A kth transaction execution is modelled by means of a *sequence of execution trees*, $T_{k(0)}$, $T_{k(1)}$,... Each individual tree $T_{k(j)}$ is a snapshot of a certain phase of the execution of the kth transaction, and each $T_{k(j)}$ is contained in $T_{k(j+1)}$ modelling the next phase. A new tree is generated whenever a command becomes completely executed. Thus, if an operation is listed in a node of an execution tree, this indicates that the command has been completely executed at a given interface and at the interfaces below it. The global Commit and Abort operations are excluded from the above rule. Either of them occurs in the root node, whenever the Coordinator has recorded, in a stable storage, the decision to abort (A_k) or to commit (C_k) the global transaction T_k.

The Prepare operation ($P^s{}_k$) occurs in a 2PCA node if the 2PC Agent has recorded, in its log, the decision to send the READY message to the Coordinator (i.e. when the subtransaction $T^s{}_k$ has been moved to the prepared state).

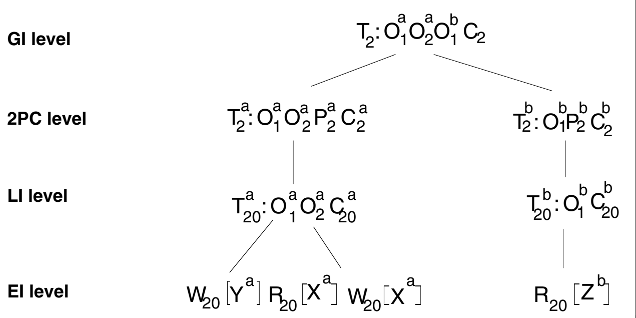
The transaction T_1 in Fig. 2 illustrates a case when a subtransaction $T^a{}_1$ became locally aborted ($A^a{}_{10}$) then resubmitted ($T^a{}_{11}$) and, eventually, locally committed ($C^a{}_{11}$).

Global transaction T_1:
O^a_1: SELECT C FROM TAB_A WHERE ID='X';
O^a_2: UPDATE TAB_A SET C=C+1 WHERE ID='Y';
O^b_1: UPDATE TAB_B SET C=C+1 WHERE ID='Z'.



T_1: $O^a_1 O^a_2 O^b_1 C_1$
T^a_1: $O^a_1 O^a_2 P^a_1 C^a_1$
T^b_1: $O^b_1 P^b_1 C^b_1$
T^a_{10}: $O^a_1 O^a_2 A^a_{10}$
T^a_{11}: $O^a_1 O^a_2 C^a_{11}$
T^b_{10}: $O^b_1 C^b_{10}$
$R_{10}[X^a]$ $R_{10}[Y^a]$ $W_{10}[Y^a]$
$R_{11}[X^a]$ $R_{11}[Y^a]$ $W_{11}[Y^a]$
$R_{10}[Z^b]$ $W_{10}[Z^b]$

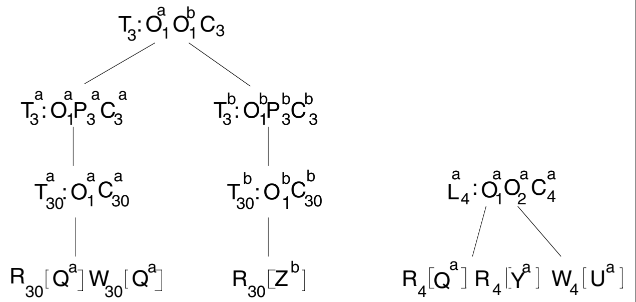
Global transaction T_2:
O^a_1: DELETE TAB_A WHERE ID='Y';
O^a_2: UPDATE TAB_A SET C=C+1 WHERE ID='X';
O^b_1: SELECT C FROM TAB_B WHERE ID='Z'.



GI level
2PC level
LI level
EI level
T_2: $O^a_1 O^a_2 O^b_1 C_2$
T^a_2: $O^a_1 O^a_2 P^a_2 C^a_2$
T^b_2: $O^b_1 P^b_2 C^b_2$
T^a_{20}: $O^a_1 O^a_2 C^a_{20}$
T^b_{20}: $O^b_1 C^b_{20}$
$W_{20}[Y^a]$ $R_{20}[X^a]$ $W_{20}[X^a]$
$R_{20}[Z^b]$

Global transaction T_3:
O^b_1: SELECT C FROM TAB_B WHERE ID='Z';
O^a_1: UPDATE TAB_A SET C=C+1 WHERE ID='Q'.

Local transaction L_4:
O^a_1: SELECT C FROM TAB_A WHERE ID='Y' OR ID='Q';
O^a_2: DELETE TAB_A WHERE ID='U'.



T_3: $O^a_1 O^b_1 C_3$
T^a_3: $O^a_1 P^a_3 C^a_3$
T^b_3: $O^b_1 P^b_3 C^b_3$
T^a_{30}: $O^a_1 C^a_{30}$
T^b_{30}: $O^b_1 C^b_{30}$
L^a_4: $O^a_1 O^a_2 C^a_4$
$R_{30}[Q^a]$ $W_{30}[Q^a]$
$R_{30}[Z^b]$
$R_4[Q^a]$ $R_4[Y^a]$ $W_4[U^a]$

Fig. 2. Examples of transactions.

All the transactions shown in Fig. 2 are *committed* (global C^i performed) *and complete*, meaning the local commit operations C^x_{ik} have been performed at all the sites involved. The data items X^a, Y^a, etc. are assumed to be single concrete table rows at site a.

From an execution tree, we form the corresponding *transaction history* $H(T_k)$ which closely corresponds to a *transaction* in [5]. It contains all R and W operations at the leaf level, all A and C operations, and all P operations, that occur in the tree T_k on higher levels.

The key idea for using the sequence of trees is to set up a necessary order $<_{H(Tk)}$ between R, W, P, C, and A operations in $H(T_k)$. The order is equivalent to the order of the indices $_{(j)}$ of the trees in the sequence $T_{k(0)}$, $T_{k(1)}, \ldots T_{k(j)}$ where the operations in question occur for the first time. Based on the properties of the different system components and the 2PC protocol, the following order holds in any transaction history [21]²:

(1) $P^i_k <_{H(Tk)} C_k <_{H(Tk)} C^s_k$ for any indices i , s, k.

Concurrent executions are modelled through linear histories. Each history, denoted by H, is an element of the shuffle $H(T_1)*H(T_2)*\ldots*H(T_n)$ [19], where T_k denotes a particular tree $T_{k(j)}$. We denote the total order of H by $<_H$.

What kind of concurrency anomalies can occur in a history? Let local history $H(^i)$ be a projection of H onto the operations of the ith site. The conflict (and view) serializability, as defined in [5], is guaranteed for each $H(^i)$, contained in H, on the basis of the SRS property.

The following history is formed of $H(T_1)$ and $H(T_2)$ from Fig. 2:

H_1: $R_{10}[X^a]$ $R_{10}[Y^a]$ $W_{10}[Y^a]$ $R_{10}[Z^b]$ $W_{10}[Z^b]$ P^a_1 P^b_1 C_1 A^a_{10} C^b_{10} $W_{20}[Y^a]$ $R_{20}[X^a]$ $W_{20}[X^a]$ $R_{20}[Z^b]$ $W_{20}[Z^b]$ P^a_2 P^b_2 C^a_{20} C^b_{20} $R_{11}[X^a]$ C^a_{11}.

Noticing that T^a_{10} is locally aborted at site a, the history

$H_1(^a)$:$R_{10}[X^a]$ $R_{10}[Y^a]$ $W_{10}[Y^a]$ P^a_1 A^a_{10} $W_{20}[Y^a]$ $R_{20}[X^a]$ $W_{20}[X^a]$ P^a_2 C^a_{20} $R_{11}[X^a]$ C^a_{11},

would be locally serializable in the traditional sense of [5], where the local committed projection $C^a(H_1)$ would contain only the R and W operations following A^a_{10} in $H_1(^a)$. The resubmission of a local subtransaction causes in it, however, a serious problem, called here the *global view distortion*: T^a_{11} reads X^a in $H_1(^a)$ from T_2, whereas T^a_{10} reads X^a from T_0 (not shown here). There is no serial history containing $H(T_1)$ and $H(T_2)$ where T_1 could get two views. Furthermore, the decomposition $D(T^a_{11}) = R_{11}[X^a]$ differs from $D(T^1_{10}) = R_{10}[X^a]$ $R_{10}[Y^a]$ $W_{10}[Y^a]$, since T^a_{20} deleted Y^a. This is impossible in a serial history. It is evident that from the global point of view, H_1 should not be regarded as "serializable"—even if both local projections are.

What is the "serializability" the DTM should guarantee? Since in a serial history no concurrency anomalies can occur, we can use the serial history as a yardstick of

² Theorem 8 in [21]

correctness, as usual. We base the definition of the serializability of a history on its committed projection C(H) and its equivalence to a serial history, as in [5]. We only include the globally committed complete transactions into our committed projection. In addition to C(H) in [5], our C(H) includes *all unilaterally aborted local subtransactions that belong to globally committed complete transactions.* With the C(H) so defined, the above concurrency anomalies can be captured.

The (conflict and view) equivalence relations and serialization graphs (SG) are then defined in the spirit of [5]. The definitions are given in [22] and a complete treatment in [21]. The view equivalence is defined in the spirit of [5], i.e. only committed writes are taken into account as final writes. As in [5], SG(H) may be cyclic but H—still view serializable. Since (even under DLU) we can not enforce acyclic SG(H), we use the redefined view serializability as the ultimate correctness criterion of the 2PCA Certifier.

4 Coping with global view distortion

4.1 Principles of prepare certification

The global view distortion problem means that a resubmitted local subtransaction T^i_{kj}, j>0, gets another view and—in the worst case—has another decomposition than the original local subtransaction T^i_{k0}. The reason are other local subtransactions and—if DLU is not obeyed—also local transactions that might update the bound data between T^i_{k0} and T^i_{kj}.

How can a 2PCA Certifier act to prohibit the global view distortion? It can have a direct effect on the global subtransactions at that site, and an indirect effect on the other subtransactions through the 2PC protocol exchanges with the Coordinator.

At its site the Certifier can perform three things:

a) it can choose the order of a command execution,
b) it can abort a local subtransaction, and
c) it can resubmit commands to create a new local subtransaction.

Which possibilities should it use? Since only committed subtransactions can jeopardize the serializability, and since no global transaction is committed unless it is moved to the prepared state, the Certifier does not need to intervene in the execution order before the PREPARE-message arrives. The idea of the *prepare certification* is that the Certifier, having received the PREPARE message, ascertains that the subtransaction can be moved to the prepared state, without a danger that a later commitment of the same global subtransaction violates serializability among the global subtransactions. If the check fails, the Certifier aborts the local subtransaction and issues REFUSE to the Coordinator, otherwise it moves the global subtransaction to the prepared state and issues READY to the Coordinator. Thus, the transactions that might cause global view distortion are filtered out, i.e. aborted. Let the Certifier enforce the following invariant:

Correctness Invariant (CI):

1) no two global subtransactions with conflicting local subtransactions can be simultaneously in the prepared state at a site, and

2) no global subtransaction with a unilaterally aborted local subtransaction is moved to the prepared state;

We have shown that if CI and DLU are obeyed, then the global view distortion does not occur [21].

How the conflicting subtransactions could be detected? Let a subtransaction be *alive* if all its DML commands are completely executed and it has been neither locally committed nor aborted. We base the conflict detection algorithm on the following conjecture:

Conflict Detection Basis: If two local subtransactions are alive at the same time and the LTM produces locally rigorous histories, then the subtransactions have neither directly nor indirectly conflicting elementary database operations.

To see that the claim holds let us assume there is some R—W or W—W local conflict between two local subtransactions T^i_{kj} and T^i_{ht} and let them both be alive. Thus the decomposition of each command has been completely performed by the LDBS and the retrieved data or return values of update commands handed over to the 2PC Agent. Due to our modeling principles it holds for the history $H(^i)$: $O_{kj}[X^i] <_H O_{ht}[X^i]$ or $O_{ht}[X^i] <_H O_{kj}[X^i]$ and there is neither local commit nor abort in H for either subtransaction. Thus, the local history $H(^i)$ is not rigorous. Since we assumed local rigorousness, nonpending commands and aliveness, the direct conflict assumption must be false.

Similarly, assuming an indirect conflict through a local transaction L_o leads either to order: $O_{kj}[X^i] <_H O_{o1}[X^i] <_H O_{o1}[Y^i] <_H O_{ht}[Y^i]$ or to order $O_{ht}[Y^i] <_H O_{o1}[Y^i] <_H O_{o1}[X^i] <_H C_o <_H O_{kj}[X^i]$. Neither history is rigorous. Adding the needed local commit C^i_{kj} (C^i_{ht}) would contradict with the assumption that T^i_{kj} (T^i_{ht}) is alive. Thus, the indirect conflict assumption is false.❑

4.2 Basic prepare certification based on alive time intervals

The above Conflict Detection Basis is implemented in the Certifier by checking, whether the transaction to be certified was alive at the same time in the past when all of the subtransactions currently in the prepared state were also alive. The *alive check* algorithm (see the Appendix) that checks whether a local subtransaction is alive, is based on UAN. The Certifier maintains the *alive time intervals* for all the subtransactions in the prepared state at a site. The conflict-freeness as assured by the following rule:

> **Alive time intersection rule:** if the intersection of the two alive time intervals is non-empty then there is no conflict between the corresponding subtransactions.

Moving a new global subtransaction to the prepared state means algorithmically testing whether the alive time interval to be inserted has a non-empty intersection with all other intervals known to the Certifier.

The transaction to be certified has only one alive time interval known to Certifier, namely the time between the last performed operation and the time of the checking moment itself. Which alive time interval of the prepared transactions should be used during the certification, and therefore stored? Provided that the overall decomposition of the jth local subtransaction at the ith site, $D(T^i_{kj})$, is the same as the decomposition of the first local transaction, $D(T^i_{k0})$, then it does not actually matter. This is because there are no conflicting operations in two arbitrary global subtransactions T^i_k and T^i_s, provided there are no conflicting operations in any of the local subtransactions T^i_{kl} and T^i_{sr}, $l \geq 0$, $r \geq 0$. Especially, there will be no conflicts in the local subtransactions possibly resubmitted in the future, i.e. assuming stable decomposition, *it is possible to deduce the conflict-freeness in the future from the information in the past.*

Using DLU and induction on the number of local subtransactions in $H(^i)$, CI can be shown to hold, no matter which existing interval is used during certification [21]. The easiest way to implement the Certifier is to simply *store the last* alive time interval for each global subtransaction being in the prepared state. As an optimization, several of them might be stored.

After the prepare certification has been performed, the alive checks are performed at regular intervals. An interval is updated as a result of a successful alive check. If the check fails then no interval is updated. A new interval is always initiated after the resubmission of all the commands is complete.

5 Local view distortion

5.1 Illustration of the problem

We are dealing with a local view distortion when local transactions get non-serializable views caused by unilateral aborts.

Consider a history formed of $H(T_1)$, $H(T_3)$, and $H(L_4)$ from Fig. 2:

H_2: $R_{10}[X^a]\ R_{10}[Y^a]\ W_{10}[Y^a]\ R_{10}[Z^b]\ W_{10}[Z^b]\ P^a_1\ P^b_1\ C_1\ A^a_{10}\ C^b_{10}\ R_{30}[Z^b]\ R_{30}[Q^a]\ W_{30}[Q^a]\ P^a_3\ P^b_3\ C_3\ C^a_{30}\ C^b_{30}\ R_4[Q^a]\ R_4[Y^a]\ W_4[U^a]\ C_4\ R_{11}[X^a]\ R_{11}[Y^a]\ W_{11}[Y^a]\ C^a_{11},$

which causes the cycle $T_1 \longrightarrow T_3 \longrightarrow L_4 \longrightarrow T_1$ in SG(H). Also, H_2 is not view equivalent to any serial history, since L_4 reads Q^a from T_3 and Y^a from a hypothetical initializing transaction T_0 (and not T_1), however T_3 reads Z^b from T_1. Also state serializability [19] is jeopardized, since L_4 writes U^a based on an inconsistent view on Q^a and Y^a.

In H_2, T_3 and T_1 a have a direct conflict through $W_{10}[Z^b]$, $R_{30}[Z^b]$. Local view distortion can also occur in a situation, where the global transactions do not have direct conflicts, but the local transactions cause them indirectly. For instance, in

H_3: $W_{50}[X^a]\ W_{50}[U^b]\ W_{60}[Z^b]\ W_{60}[Y^a]\ P^a_1\ P^a_2\ A^a_{50}\ C_2\ P^b_2 P^b_1\ A^b_{60}\ C_1\ C^b_{50}\ R_7[U^b] R_7[Z^b]\ C_7\ C^a_{60}\ R_8[X^a] R_8[Y^a]\ W_8[V^a]\ C_8\ W_{51}[X^a]\ C^a_{51}\ W^b_{61}[Z]\ C^b_{61}.$

formed of $H(T_5)$, $H(T_6)$, $H(L_7)$, and $H(L_8)$, there are local transactions L_7 and L_8 that get non-serializable views.

The local view distortion is possible in H_2 and H_3, because the local commits of the global transactions are in reversed orders at different sites. In H_2, for instance, $C^b_{10} <_{H2} C^b_{30}$ and $C^a_{30} <_{H2} C^a_{11}$. If the commits were in the same order, this order would be a *global view serialization order*. This is because, under SRS, the order of local commits is the unique local serialization order between conflicting transactions and it is always a *possible* serialization order [9].

Generalizing the observation, we can make use of a *commit order graph* of history H, CG(H). Its nodes are those transactions T_k that have at least one local commit C^x_{kj} in H. There is an arc from T_k to T_i iff $C^x_{kj} <_H C^x_{ig}$ for some x in H.

Evidently, local view distortion is possible in H only if CG(C(H)) is cyclic; if it is acyclic, then it can be topologically sorted. Thus, a serial history H_s can be formed that contains exactly the same transaction histories $H(T_k)$ as C(H) and has the same order of local commits as C(H). For a local transaction L_o this means that exactly the same local commits C^i_{kj} precede its commit C_o in both histories. Under rigorousness, $O_{k0}[X^i] < C^i_{kj} < O_{o0}[X^i] < C_o$ is the only possible order for conflicting operations in H_s and C(H). From this it follows, especially, that a local transaction reads the same data items from the same local (sub)transactions in both histories. Also the final writes are the same. H_s and C(H) are thus view equivalent provided no global view distortion does occur. This requires CI, SRS and DLU to hold [21][3]. Based on the acyclicity of CG(C(H)), the view equivalence can be shown to hold for each prefix H' of H. Thus, as a whole, H is view serializable.

5.2 Commit certification

To prohibit the local view distortion, it is sufficient that the commit order graph is kept acyclic. The Certifiers can achieve this by issuing the commit operations for the local subtransactions in a *globally unique total order.*

Which order should be chosen? A simple possibility is to guarantee that the transaction identifiers are picked up from a totally ordered set used by each Certifier. This approach was discussed e.g. in [13]. This would be quite restrictive, because it would require all global transactions to be serialized in the same order even if they could not have caused any problems. Another deficiency is that, if the local systems serialized the transactions in an order that differs from the predefined order, a deadlock might occur among Coordinators, or global or local transactions might become aborted in vain.

Evidently, the best choice would be to use a locally determined necessary serialization order, because it minimizes the unnecessary aborts. How does a Certifier find this order? What guarantees that all Certifiers find the same order?

Let us assume that the local serialization order of two global subtransactions $T^i{}_j$ and $T^i{}_k$ has been determined by the LTM before one of them is moved to the prepared state. In this case the prepare operations of $T^i{}_j$ and $T^i{}_k$ are in the *same order at each site* they have subtransactions at. That is, $P^s{}_k <_H P^s{}_j$ or $P^s{}_j <_H P^s{}_k$ for any index *s*[4], and additionally, their order is the *serialization order.*

To see this, let $T^i{}_k$ be moving from the active to the prepared state and $T^i{}_{k0}$ thus alive. By assumption, it conflicts directly or indirectly with $T^i{}_j$. Let the operations at the end of the conflict chain be $O_{jy}[Y^i]$ and $O_{k0}[X^i]$. Assume $O_{k0}[X^i] <_H O_{jy}[Y^i]$. Under rigorousness, X^i must have been committed in $T^i{}_{k0}$, which is, by inequality (1), in contradiction with our assumption that it is about to be prepared. Thus, $O_{jy}[Y^i] <_H O_{k0}[X^i] <_H P^i{}_k$ must hold. Again, by rigorousness and by the existence of a chain of locally conflicting operations between $O_{jy}[Y^i]$ and $O_{k0}[X^i]$, we can deduce that there are also local commits between each conflicting pair. Using (1) and CI we thus get $O_{j0}[Y^i] <_H P^i{}_j <_H C^i{}_{jy} <_H O_{k0}[X^i] <_H P^i{}_k$ if y=0, or $P^i{}_j <_H A^i{}_{j0} <_H O_{jy}[Y^i] <_H C^i{}_{jy} <_H O_{k0}[X^i] <_H P^i{}_k$, if y>0. Reapplying inequality (1) holding for $H(T_k)$ and $H(T_j)$, we get $p^g{}_j <_H C_j <_H C^i{}_{jy} <_H O_{k0}[X^i] <_H P^s{}_k$, i.e. $P^g{}_j <_H P^s{}_k$ for any site indices g and s.❑

The observation could be used operationally by recording, at each site, the order in which subtransaction entered the prepared state. During the commit certification, this order would again be kept. This approach would be enough to guarantee acyclicity of CG(H) for local view distortions as in H_2. Unfortunately, this does not work, if, as in H_3, the original local subtransactions do not conflict directly or indirectly. Therefore, the prepare operations might occur in arbitrary orders at site a and b. Using the order of the prepare operations independently by the Certifiers might lead exactly to a cyclic CG(H), which we tried to avoid.

One solution is to use a globally determined order that is equivalent with the unique serialization order, if it exists. For this purpose a *serial number* of a transaction T_j, SN(j), is used. SN(j) is unique and picked from a totally ordered set. The latter requirement can then be formulated:

(2) If T_x precedes T_y in a local serialization order, then $SN(x) < SN(y)$ holds.

To fulfil (2), numbers SN(x) and SN(y) and must be determined during the execution. It is possible to do it when all original DML commands have been executed and the conflicts have been detected by the LTMs, i.e. when the application submits the Commit to the Coordinator. At this moment, the Coordinator gives a globally unique serial number to the transaction. This number is transmitted with the PREPARE messages to each participating site. Each Certifier stores it. When the COMMIT message later arrives, the Certifier enforces the local commits in the order of the serial numbers known to it. The algorithm is presented in the Appendix.

Let also the generation of the serial number be denoted by SN(). Assuming that a serial number increases with the real time, the condition above becomes fulfilled: the inequality $O_{j0}[Y^i] <_H SN(j) <_H P^s{}_{SN(j)} <_H C^i{}_{SN(j)y} <_H O_{k0}[..] <_H SN(k) <_H P^s{}_{SN(k)} <_H C_k <_H C^s{}_{SN(k)g}$ holds for any site indices *s*, provided there is a direct or indirect conflict assumed above at site *i*.

How the serial number could be generated? Basically, using any suitable technique like a centralized counter or a logical distributed clock. However, these are cumbersome techniques in an autonomous environment. It is appealing to use real time site clocks, expanded with the unique site identifier, for this purpose, too. The amount of the time drift among the clocks has no influence on the correctness of the Certifier. The drift may cause unnecessary aborts, only.

[3] Th. 19. ibid.

[4] Under CI and DLU, if a pair of local subtransactions $T^i{}_{ks}$ and $T^i{}_{jy}$ conflict, then the original ones must also conflict, because their decomposition cannot differ from each other. Note, that under DLU, the local transactions like L_o, can also update data read and updated by global transactions, while the data are not bound.

It seems, that if the amount of the drift is kept within the time of four message exchanges over the network, the solution is as good as an ideally synchronized one. This is for further study.

5.3 Prepare certification extension

What then guarantees that all Certifiers find the same order? Unfortunately, this is not guaranteed for those global transactions that do not conflict directly, without additional measures. The reason is that the COMMIT message of T_k could overtake the PREPARE message of T_j at site s, even if T_j reached the prepared state at site *i* earlier than T_k. Thus, the order

$SN(j)\ P^i{}_{SN(j)}\ SN(k)\ P^i{}_{SN(k)}\ P^s{}_{SN(k)}\ C^s{}_{SN(k)}\ P^s{}_{SN(j)}\ C^i{}_{SN(j)}\ C^i{}_{SN(k)}\ C^s{}_{SN(j)}$

is possible in H_x – and $CG(H_x)$ is clearly cyclic.

To avoid this problem, the prepare certification must be extended to abort such transactions that might cause cycles in CG(H). This is done by recording the so-far biggest serial number of a committed subtransaction at each Certifier. If a PREPARE message arrives with a smaller number, the subtransaction, eg. $P^s{}_{SN(j)}$ above, will be aborted during the prepare certification.

6 Related work

We shall confine the discussion to works striving for serializability in the presence of failures. Researchers have made different assumptions about the LTM and the nature of failures. Both in [2] and [3] a simulated prepared state and a resubmission was proposed to deal with some failures, but the unilateral aborts in the prepared state were not included. The idea of resubmission may be found also in [14] to serve the needs of site recovery but not subtransaction recovery. In [18] various schemes are proposed, some of them guaranteeing serializability and some of them not, yet a special global scheduler (e.g. lock-based) has to be used to maintain serializability. Neither local transactions are allowed in the system nor unilateral aborts are taken care of.

A notable progress was made in [8] proposing a method we shall call the Commit Graph method (CGM). The following is a detailed comparison of the 2PCA Certifier method (2CM in the sequel) with CGM. The methods share a similar objective to guarantee the global serializability in the presence of failures of similar types, but the proposed solutions are significantly different.

Architecture

The DTM of CGM uses a centralized scheduler while the scheduling in the 2CM is decentralized. There is a dedicated 2PCA for each LTM in 2CM. On the other hand, the agents (called servers) of the CGM do not share a common state at a site—they are instantiated for each global subtransaction separately. Both methods allow for unilateral aborts and they do the global subtransaction recovery by resubmitting the database commands belonging to the transactions aborted in the wake of a failure. The 2PC protocol is used in both methods. The assumptions about the LTMs are also similar, however with the difference that 2CM allows for any implementation resulting in local SRS histories, as compared to a less general assumption of the S2PL policy, in CGM.

Dealing with the global view distortions

CGM assumes a global S2PL lock manager is used by the DTM. This, together with the partitioning of data, protects against the global view distortion. However, it is not obvious how the global lock manager can be implemented in a contemporary environment unless some coarse granularity (e.g. site, database or table) locking is applied. Also, unless the granularity level is at least database, the triggers would not be allowed with this solution.

In 2CM, the basic prepare certification protects against the global view distortions.

Restrictions imposed on the local transactions

The local transactions are restricted by the DLU assumption in 2CM. In CGM, the restriction is imposed in a less general way by partitioning the data items into the *locally updateable set* and the *globally updateable set*. As concerns reads, an additional restriction is that those global transactions that update data items, are not allowed to read the locally updateable set. Both approaches have seemingly similar practical consequences of which the most important is that the results of the local transactions are not readily available to global transactions.

Dealing with the local view distortions

The commit graph of CGM is instrumental in protecting against the local view distortions. It is an undirected graph whose nodes are global transactions and Participating Sites. An edge connects a transaction node T_j with a site node S_i iff the global subtransaction $T^i{}_j$ is in the prepared state. The loop in the graph signals a potential conflict among global and local transactions. Thus the granularity of the potential conflict detection is that of a site.

In 2CM, the commit certification and the extension of the prepare certification protect against the local view distortions.

Resolving deadlocks

In 2CM, the timeout based deadlock resolution is assumed to be used. On the other hand, CGM employs an elaborate combination of three graphs, enabling to detect potential deadlock situations including all real global deadlocks, also the ones involving local transactions.

Restrictiveness

It is difficult to formally compare the restrictiveness of the methods. If we assume that neither checking the order of the arriving PREPARE messages, nor too long a time between alive time checks ever cause aborts, 2CM is less restrictive than CGM: in a failure-free situation it does not abort any transactions. Thus, there are histories accepted by the 2PCA Certifier but rejected by a CGM based DTM because of the site-level granularity in the commit graph. If failures occur, the comparisons become more complicated and are for further study. The effective performance of 2CM is also for further study.

7 Notes on the implementation of the Certifier

The Certifier algorithms have been implemented in the HERMES prototype system at the Laboratory for Information Processing of the Technical Research Centre of Finland (VTT). The system incorporates two commercial database products: the SQL Server (Sybase Inc.) and INGRES (Ask Computer Systems, Inc.) A commercial implementation of the 2PC protocol (by Sybase Inc.) is also used. The 2PCA Certifier is implemented on a VAX/VMS site in connection with the INGRES DBMS, using the single-phase transaction interface of INGRES.

8 Conclusions

We have presented a complete Distributed Transaction Manager, applicable in an environment of the design and execution autonomous database systems. Its key component is the 2PCA Certifier. The presented Certifier algorithms allow to take advantage of e.g. the dynamic, strict two-phase locking offered by the contemporary DBMSs. They also allow for recovery from unilateral aborts. The basic prepare certification protects against the serialization errors among global transactions, resulting from unilateral aborts. The commit certification and the prepare certification extension assure that local transactions also get a correct view of the global data in presence of failures.

A DTM based on the 2PCA Certifier does not require any centralized component in the architecture of a multidatabase transaction system. It is based on simple algorithms that can be replicated onto as many sites as needed. The interactions between the sites in HDBMS is carried out by way of the 2PC protocol.

Bibliography

[1] F. Bancihlon, W. Kim and H.F. Korth., "A Model of CAD Transactions", *Proc. 11th VLDB Conf.* (Stockholm, August 1985), pp. 35-33.

[2] Ken Barker, "Transaction Management on Multidatabase Systems", TR 90-23 (Ph.D. thesis), August 1990, Dept . of Computing Science, The Univ. of Alberta, Edmonton, Alberta, Canada.

[3] K. Barker and M. T. Özsu, "Reliable Transaction Execution in Multidatabase Systems", *Proc. First International Workshop on Interoperability in Multidatabase Systems* (IMS'91, Kyoto, Japan, April 7-9, 1991), pp. 344-347.

[4] C. Beeri, P. A. Bernstein and N. Goodman, "A Model for Concurrency in Nested Transactions Systems", *Journal of ACM*, Vol. 36, No. 2 (April 1989), pp. 230-269.

[5] P. A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency control and recovery in database systems", Addison-Wesley Publ. Comp., 1987.

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

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

[8] 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.

[9] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, A. Silberschatz, "On Rigorous Transaction Scheduling", *IEEE Trans. on Software Eng.*, Vol. 17, No. 9 (Sept. 1991), pp. 954 - 960.

[10] Special Issue on Heterogeneous Databases, *ACM Comp. Surveys*, Vol. 22, No. 3. (September 1990).

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

[12] F. Eliassen and J. Veijalainen, "Language support for Multi-database Transactions in a Cooperative, Autonomous Environment", *Proc. TENCON 87* (Seoul, 25-28 August , 1987), pp. 277 - 281.

[13] A. K. Elmagarmid and W. Du, "A Paradigm for Concurrency Control in Heterogeneous Distributed Database Systems", *Proc. IEEE Int. Conf. on Data Eng.*, Los Angeles, February 1990.

[14] D. Georgakopoulos, "Multidatabase Recoverability and Recovery", *Proc. First Internat. Workshop on Interoperability in Multidatabase Systems* (IMS'91, Kyoto, April 7-9, 1991), pp. 348-355.

[15] V. Gligor and R. Popescu-Zeletin, "Transaction Management in Distributed Heterogeneous Data-

base Management Systems", *Information Systems*, Vol. 11, No. 4 (1986), pp. 287 - 297.

[16] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery", *ACM Comp. Surveys*, Vol. 15. No. 4 (December 1983).

[17] ISO/IEC 9804: 1990. "Information technology—Open Systems Interconnection—Service definition for the Commitment, Concurrency and Recovery service element", International standard, ISO/IEC, 1990.

[18] P. Muth and T. Rakow, "Atomic Commitment for Integrated Database Systems", *Proc. 7th Conf. Data Eng.* (Kobe, 8-12 April, 1991), pp. 296-304.

[19] C. H. Papadimitriou, "The Theory of Database Concurrency Control", Computer Science Press, 1986.

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

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

[22] Antoni Wolski and Jari Veijalainen, "2PC Agent Method: Achieving Serializability in Presence of Failures in a Heterogeneous Multidatabase", *Proc. IEEE PARBASE-90 Conf.* (Miami Beach, 7-9 March, 1990) pp. 321 - 330.

Appendix. The algorithms of the 2PCA Certifier

The following are the essential algorithms of the certifier. The algorithms are depicted as state transitions within the states of the Participant in the 2PC protocol.

A. Alive check

```
when in prepared state
upon  alive check interval timeout
{
        check whether the transaction is alive;
        If alive then           // there has been no failure
        {       update the end of the alive time interval;
                return to prepared state;
        }
        else            // unilaterally aborted.
        {
                resubmit commands from the Agent log;
                set beginning of the new alive time interval;
                return to prepared state ;
        }
}
```

B. Extended prepare certification

```
when in active state
upon  receiving  the PREPARE message from the
Coordinator
{       // extended prepare certification.
        check whether there is an "older" subtransaction
        already  in the committed state;
        if true then  // PREPARE out of order—certification
        extension failed, abort the global transaction.
        {       respond REFUSE to the Coordinator;
                return to  idle state ;
        }
        else  //certification extension OK
        {       // basic prepare certification.
                check whether the alive interval of this
                transaction has a non-empty intersection with
                each of the alive intervals of the subtransactions
                being in the prepared state at this 2PCA;
                if true then  //basic prepare certification OK
                {
                        insert the transaction into the alive
                        interval table ;
                        check whether the transaction is
                        alive;
                        If alive then  //  no failure;
                        {
                                force write the prepare record in the
                                Agent log;
                                READY to the Coordinator;
                                set the alive check interval
                                timeout;
                                return to prepared state;
                        }
                        else            //unilaterally aborted.
                        {
                                REFUSE to the Coordinator;
                                remove the transaction from the
                                alive interval table;
                                return to  idle state ;
                        }
                }
                else  //prepare certification failed: abort.
                        REFUSE to the Coordinator;
                        return to  idle state ;
                }
}
```

C. Commit certification

```
when in prepared state
upon  receiving  the COMMIT message from the Coordinator
or
upon  commit certification  retry time out
{
        check whether all the subtransactions that are in the
        alive interval table have a bigger serial number than the
        transaction to be certified;
        if true then            //commit certification OK
        {
                write the commit record to the Agent log;
                commit the local subtransaction  and the commit
                record in the Agent log;
                COMMIT-ACK to the Coordinator;
                delete the transaction from the alive interval
                table;
                return to  idle state ;
        }
        else  //commit certification failed
        {
                set the commit certification retry timeout to
                retry the commit certification at a later time;
                return to prepared state ;
        }
}
```