Recovery in Multidatabase Systems

Angelo Brayner
Federal University of Ceara
brayner@lia.ufc.br

Theo Härder
University of Kaiserslautern
haerder@informatik.uni-kl.de

Abstract

A *multidatabase* consists of a collection of autonomous local databases. Systems used to manage multidatabases are called multidatabase systems (MDBSs). In such a system, *global transactions* are executed under the control of the MDBS. Independently, *local transactions* are submitted directly to a local DBS (LDBS) by local applications. An MDBS should provide a mechanism to globally manage transactions. However, global transactions are long-living and involve operations on multiple and autonomous local databases. Moreover, MDBSs do not have any information about the existence and execution order of local transactions. Thus, conventional approaches to manage transactions are unsuitable for MDBSs. In this paper, we address the reliability problem in MDBSs. For this purpose, we propose two types of protocols for making MDBSs resilient to failures. One type of protocol should enforce that, when a given global transaction completes its execution, it has the same state (committed or aborted) at every site it has run. The other type determines the actions to be triggered after failures in a multidatabase environment. These protocols can reduce the frequency of global transaction undo after the occurrence of failures, and make the MDBS able to deal with failures which may occur in a multidatabase environment.

1 Introduction

An MDBS integrates a set of autonomous and heterogeneous local DBSs. In turn, each local DBS consists of a local DBMS and a database. Users can access information from multiple sources through global transactions. Operations belonging to global transactions are executed by local DBMSs. Besides global transactions, there exist local transactions in a multidatabase environment. Local transactions result from the execution of local applications. Such applications are typically pre-existing with regard to the integration realized by MDBSs.

A computer system is subject to failures. Such failures may provoke loss of information. Hence, MDBSs should be able to react in failure situations in order to restore the multidatabase to a consistent state, without human intervention, that is, automatically. However, ensuring reliability in MDBSs is a very complex task. First of all, more types of failures may occur in MDBSs (e.g., a communication failure which can isolate a local DBS from the MDBS) than in centralized DBMSs. Second, in MDBSs, there is a tradeoff between preserving local autonomy and providing an efficient global recovery mechanism. Since several existing DBMSs do not support the two-phase commit (2PC) protocol [3, 14], we have to assume that such a protocol can not be used when designing a recovery mechanism for MDBSs.

In order to make transaction processing in MDBSs resilient to failures, two types of protocols are required. One type of protocol should enforce that, when a given global transaction completes its execution, it has the same state (committed or aborted) at every site it has run. Such protocols ensure what we call commit atomicity. They are called commit protocols. The other type of protocols, denoted recovery protocols, determines the actions to be triggered after failures in a multidatabase environment.

In this work, we will describe a commit and a recovery strategy. They can be implemented to ensure transaction processing reliability in an MDBS which does not use a 2PC protocol. The proposed commit strategy guarantees commit atomicity. In turn, the recovery strategy, denoted ReMT (*Recovering Multidatabase Transactions*), enables the MDBS to deal with the specific failures in multidatabase environments. Our proposal is suitable to a wide variety of multidatabase applications, such as CAD, CASE, GIS and WFMS.

This paper is structured as follows. In the next section, a reference architecture for an MDBS is presented. In Section 3 the most critical failures which may occur in a multidatabase environment will

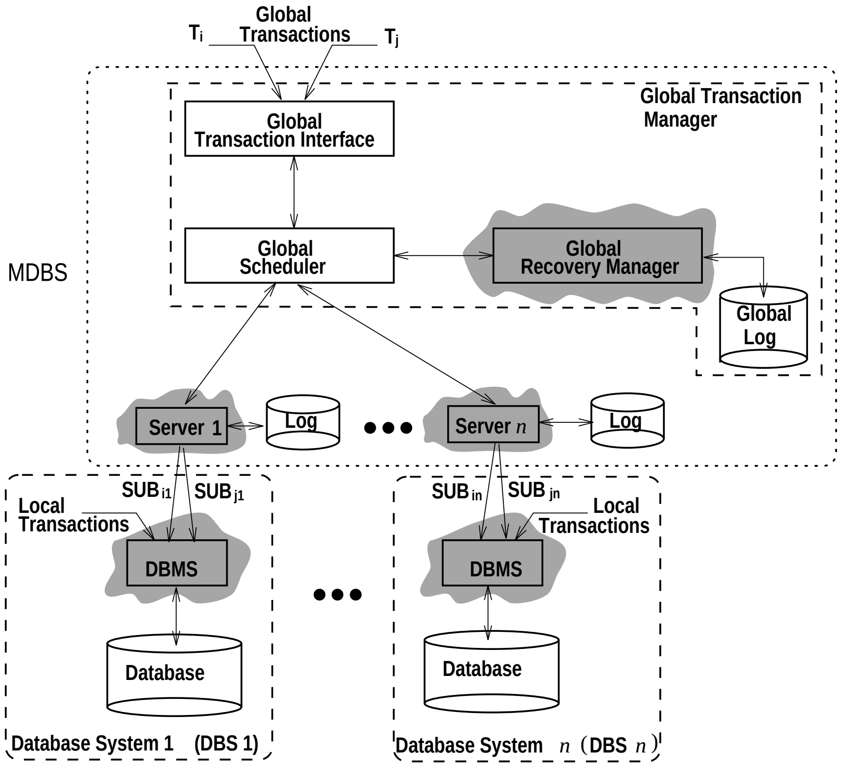

Figure 1: A model for MDBS with recovery support.

be described. Moreover, we will discuss and analyze the problems which arise when these failures happen. A logging strategy is then provided in Section 4. A protocol for ensuring commit atomicity is described and analyzed in Section 5. Thereafter, a recovery strategy for recovering from failures in MDBSs is presented in Section 6. Section 7 concludes this paper.

2 The Multidatabase System Model

The architecture of an MDBS basically consists of the Global Transaction Manager (GTM), a set of Interface Servers (servers, for short), and multiple LDBSs. To each LDBS, there is an associated server. An LDBS consists of a DBMS and at least one database. The GTM comprises three modules: Global Transaction Interface (GTI), Global Scheduler (GS), and Global Recovery Manager (GRM). An MDBS architecture is depicted in Figure 1.

Users interact with the local DBMS (LDBMS) by means of transactions. Two classes of transactions are supported in a multidatabase environment:

- **Local transactions** which are transactions executed by an LDBMS outside the control of the MDBS and
- **Global transactions** which comprise transactions submitted by the MDBS to LDBMSs. Global transactions may be executed in more than one local system. Thus, we define a global transaction G_i as a set of subsequences $\{\text{SUB}_{i,1}, \text{SUB}_{i,2}, \text{SUB}_{i,3}, \ldots, \text{SUB}_{i,m}\}$ where each $\text{SUB}_{i,k}$ is executed at the local system LDBS_k as an ordinary local transaction.

We will assume that, after a subsequence $SUB_{i,k}$ of a global transaction G_i ends its execution at the local system $LDBS_k$, objects updated by this subsequence may be viewed by other global transactions or local transactions. This represents an attractive property, since local resources are not locked during long periods of time on behalf of global transactions [6].

Local autonomy is a key property presented by multidatabase technology. For that reason, we consider the following facets of autonomy: *(i)* by joining a DBS to an MDBS, no modification in the DBMS software should be made and *(ii)* the LDBMS has full control over the processing of transactions which it executes. This means, the LDBMS can unilaterally abort a transaction at any time, even when the MDBS has already decided to commit the corresponding global transaction.

Unfortunately we have to give up some aspects of local autonomy, in order to provide a global recovery mechanism with a higher degree of applicability than the existing proposals. For that reason, we relax local autonomy in the following way:

1. After executing the analysis pass of the recovery procedures from local system failures, the LDBMS passes on to the MDBS information about *loser transactions*[1].

2. After an LDBMS completes the recovery actions for local system failures, the interface server associated with the local system assumes the control of the recovery processing. The server then starts to execute recovery actions for subsequences which were considered losers by the LDBMS. During the execution of these actions, no local transaction can be submitted to the restarted DBMS.

3. An interface server can obtain the information about the state (e.g., *active* or *committed*) of a subsequence running at the local DBS.

4. Each LDBMS implements the two-phase locking (2PL) protocol [9] in order to enforce local serializability.

Since failures are spread throughout an MDBS (see Section 3), we have decided to distribute the recovery actions among the GTM and servers. By doing this, we optimize the recovery process. This is because each component can perform recovery actions without interfering in the processing of components which were not involved in failures.

We assume that every local DBMS provides an efficient recovery component. However, a local recovery component does not perform recovery actions on behalf of the GTM or servers. When servers or the GTM need to execute recovery actions at a local system, these actions must be submitted to the local system as local transactions.

In Figure 1, we present a model for an MDBS. The components which provide recovery actions are shadowed. In order to perform recovery procedures, the GTM and the servers need to access information, called log data or log records, stored in two different types of log files: global log and server log. Thus, in an MDBS, we have a global log file and several server log files, one for each server defined for the MDBS. These two types of log files are stored in non-volatile memory and represent a sequence of log records. Of course, each local DBMS has a log file. Due to local autonomy, only the LDBMS can manipulate the local log file.

3 Failures in an MDBS

Each component of an MDBS can fail indenpendently, this implies that more types of failures may occur in MDBSs than in centralized DBMSs. We categorize the most critical failures which may occur in a multidatabase environment as follows:

Transaction Failures. A particular global transaction may fail (abort). A transaction abort can be caused by a decision of the GTM (for example, it can be involved in a global deadlock) or can be requested by the transaction itself. When a global transaction fails, its effect should be, somehow, undone at each local database on which it has executed update operations. Recall that we have made the following assumption, after a subsequence $SUB_{i,k}$ ends its execution at a local system, objects updated by this subsequence may be viewed by other global transactions or local transactions. Hence, if a global

[1]Transactions which do not have reached their commit points before the failure.

transaction G_i (to which $SUB_{i,k}$ belongs) aborts, we can not restore the state of the local database which existed before the execution of $SUB_{i,k}$. The recovery mechanism must be able to overcome this problem. A particular subsequence of a global transaction may also abort. Several reasons may cause such aborts. However, there are two situations of subsequence aborts which should be handled in a different manner:

1. A subsequence is aborted on behalf of the LDBMS. This can happen, for instance, when the LDBMS recognizes that the subsequence is involved in a local deadlock;

2. A subsequence decides to abort. When the subsequence identifies some internal error condition (e.g., violation of some integrity constraints or bad input), it aborts its execution.

Local System Failures. LDBSs reside in different and autonomous computer systems (sites). When the contents of the main memory of the computer system in which a particular LDBS resides are lost or corrupted (due to operating system or hardware faults), we say that the local system has failed. When a local system $LDBS_k$ is restarted after a failure, the local DBMS of $LDBS_k$ should be capable to restore the most recent transaction-consistent state of the local database [15]. The local recovery actions are performed outside the control of the GTM. The MDBS should provide a strategy to guarantee commit atomicity of global transactions which have performed operations at the failed site.

Server Failures. Servers act as interfaces between an MDBS and the LDBSs. When a server associated to a local system $LDBS_k$ fails, subsequences of global transactions submitted to $LDBS_k$ can not be executed, although the site of $LDBS_k$ is in operation. We assume that the GTM can identify that a given server is not in operation.

GTM Failures. Such a failure can be caused by a bug in the GTM code, a hardware failure or an operating system fault at the site at which the GTM is running. A failed and restarting GTM should execute recovery actions to guarantee that the local DBMSs reflect the effects of committed subsequences, although the respective global transactions have not yet been committed. In order to provide this feature, the GRM must avoid to undo global transactions which were not committed when the failure has occured.

Communication Failures. The components of an MDBS are interconnected via communication links. These links are also subject to failures. Typically, such failures break the communication among some of the components of an MDBS. For instance, the link between a local system and a server may be broken. In such a situation, the local system and the server will continue to work correctly. Notwithstanding, this can lead the local system to abort the execution of some subsequences (which are being executed at the local system) by timeout.

4 Logging Strategy

A key goal of our recovery strategy is to reduce the frequency of global transaction undo after the occurrence of failures. For that reason, we need to save more information in the log files than it is recorded in centralized databases. In this section, we describe what type of information and how it is stored in the log files.

4.1 Global Logging Method

The GRM writes log records for each operation of a global transaction in the global log file. In addition, the GRM forces a log record for the application's state to the global log file whenever a subsequence initiates its operation.

The information about the application's state must be given by the application itself, since the GRM has no knowledge about it. The GRM can obtain this information using the following strategy. When the GRM receives a begin-of-subsequence operation (see Section 5), it can require the application to which the subsequence belongs to generate an installation point[2]. The application then generates the installation point and sends it to the GRM. Thereafter, the GRM forces a log record for the installation point to the global log file.

A log record of the global log file contains the following fields:

- LSN (*Log Sequence Number*). An integer which uniquely identifies a log record. The LSN increases monotonically.
- LRT (*Log Record Type*). An array of characters which identifies the type of the record. The possible values for an LRT are:
  - BOT (*Begin Of Transaction*). The log record describes the begin of a global transaction.
  - BOS (*Begin Of Subsequence*). The log record describes the begin of a subsequence belonging to the global transaction TRID.
  - DBO (*Database Operation*). The log record contains information about a read or write operation executed by SUBID. The field LRC contains the information about the type of the operation (read or write) and on which database object the operation is executed.
  - IP (*Installation Point*). It records the execution state of the application when a subsequence SUBID initiates its execution.
  - ST (*State*). The log record contains information about the state of a global transaction TRID. In this case, the field SUBID is empty. It can also indicate the state of a subsequence SUBID belonging to the global transaction TRID. We postpone the discussion about the states of global transactions and subsequences until Section 6.
- LRC (*Log Record Content*). An array of characters which contains information depending on the value of LRT. For example, if the value of LRT is 'IP', then the LRC contains information about the initial execution state of a subsequence.
- TRID (*Transaction IDentifier*). An integer which identifies the global transaction whose information is stored in the log record.
- SUBID (*SUBsequence IDentifier*). An integer which identifies the subsequence belonging to the global transaction TRID.

Therefore, the global log contains information about the execution of global transactions and their subsequences. Furthermore, the global log stores the initial execution state of each subsequence (for example, initial values of local variables in the program that execute the subsequence) of a multidatabase transaction.

4.2 Logging Protocol in Servers

Each server log contains information about subsequences executed at a particular LDBS. This information provides the necessary support to the server for performing recovery actions after server failures, local systems failures and communication failures.

A server generates a log record for each operation of a subsequence as well as a log record for the application state whenever a subsequence initiates its operation. The information about the application's state is forwarded by the GTM, when it submits a begin-of-subsequence operation to the server. The server forces these records to the server log file immediately. A server log record has the same structure of global log record.

[2]An installation point (also called application checkpoint) reflects the state of an application which should be saved in non-volatile memory [20].

5 Strategy for Ensuring Commit Atomicity

When the GTM receives a commit operation for a global transaction, it must ensure that every subsequences belonging to the global transaction is locally committed. If this is not possible for some reason, the entire global transaction must be aborted. For that reason, the GTM of an MDBS should implement a protocol, denoted *commit protocol*, for ensuring the atomicity of commit operations.

In order to ensure commit atomicity for multidatabase transactions, we propose the following strategy. The last operation of every subsequence of a particular global transaction G_i must be a *local commit operation*. We denote such an operation with $lc_{i,k}$, where k represents the LDBS in which the subsequence should be committed. A *local commit operation* within a subsequence means that the effect of the subsequence should be reflected in the local database. Besides this, locks held by the subsequence are released.

When the GTM (see Figure 1), more precisely the GS, receives a local commit operation ($lc_{i,k}$) from a global transaction G_i to commit a given subsequence $SUB_{i,k}$, it submits this operation to the appropriate server, which in turn converts the $lc_{i,k}$ operation into an ordinary commit operation to be executed by the LDBMS.

If the subsequence $SUB_{i,k}$ is successfully committed, the server sends a message to the GS reporting this fact. The server then generates a log record for $SUB_{i,k}$, where the fields LRT and LRC contain 'ST' and 'locally-committed', respectively. After that, the server saves this record in the server log file. When the GS receives the message sent by the server, the GS forwards this information to the GRM. In turn, the GRM forces to the global log file a log record for $SUB_{i,k}$. For this log record, the contents of LRT and LRC are 'ST' and 'locally-committed', respectively. In such a situation, we say that the global transaction G_i is *locally-committed* at that particular site ($LDBS_k$).

However, due to local autonomy, local DBMSs can unilaterally abort subsequences, even after the server has submitted commit operations for subsequences. In this case, the server forces to the server log a record with the information that the subsequence was aborted. After that, it sends a message to the GS indicating that the subsequence was aborted. The GS forwards this information to the GRM which, in turn, generates a log record containing the information that the subsequence was aborted. In the meantime, the server executes some recovery actions in order to avoid that the entire global transaction has to be aborted. These actions correspond to the recovery strategy we propose for subsequence failures. This strategy is described in the next section.

When the GS receives a commit operation (c) from a global transaction G_i, it sends a commit request to the GRM. The GRM verifies in the global log whether all subsequences belonging to G_i have been locally committed. With the knowledge that every subsequence of G_i has already been committed, the global transaction can be *globally committed*.

It is important to notice that, hereafter, we will consider that a global transaction contains the following types of operations:

- Database operation. It represents an action which should be executed on a database object. A database operation can be read or write operation;

- Control operation. It represents an action which is not executed on database objects. Control operations can be classified as initiation or termination operations. A global initiation operation, denoted BOT, represents the fact that a new transaction is beginning its execution. A local initiation operation, denoted $BOS_{i,k}$, indicates that a new subsequence $SUB_{i,k}$ belonging to the global transaction G_i is beginning its execution. The counterpart of initiation operations are termination operations. Local termination operations are used to represent a local commit (for short, lc) or an abort (la) to be executed for a particular subsequence. Global termination operations reflect the fact that the execution of a given global transaction is either successfully terminated, represented by a global commit operation (c), or abnormally interrupted, indicated by a global abort operation (a).

A local commit operation executed by a subsequence $SUB_{i,k}$ reflects the fact that operations of the global transaction G_i can be locally committed by local system $LDBS_k$. Moreover, locks held by G_i at each site can be released.

6 ReMT - A Recovery Strategy for MDBSs

As already mentioned, reliability in MDBSs requires the design of two different types of protocols: commit and recovery protocols. In Section 5, we have described a commit protocol which enforces commit atomicity of global transactions. In this section, we will present a strategy, called ReMT, for recovering multidatabase consistency after failures, without human intervention. In MDBSs, recovering multidatabase consistency has a twofold meaning. First, for global transaction aborts, recovering multidatabase consistency means to undo the effects of locally committed subsequences belonging to the aborted global transactions *from a semantic point of view*. In addition, the effects of transactions which have accessed objects updated by aborted global transactions should be preserved (recall that, after the last operation of a subsequence, all locks held by the subsequence are released). For the other types of failures, recovering multidatabase consistency means to restore the most recent global transaction-consistent state. We say that a multidatabase is in a global transaction-consistent state, if all local DBMSs reflect the effects of locally-committed subsequences.

The ReMT strategy consists of a collection of recovery protocols which are distributed among the components of an MDBS. Hence, some of them are performed by the GRM, some by the servers and some are provided by the LDBMSs. We assume that every participating LDBMS provides its own recovery mechanism. Local recovery mechanisms should be able to restore the most recent transaction-consistent state [15] of local databases after local failures.

For each type of failure described in Section 3, we propose a specific recovery scheme.

6.1 Transaction Failures

As seen before, we identify different kinds of transaction failures which may occur in a multidatabase environment. Each of them can be dealt with in a different manner. First, a particular global transaction may fail. This can be caused by a decision of the GTM or can be requested by the transaction itself. Second, a given subsequence of a global transaction may fail.

In the following, we will propose recovery procedures to cope with failures of global transactions and subsequences.

6.1.1 Global Transaction Failures

A global transaction failure may occur for two reasons. The abort can be requested by the transaction or it occurs on behalf of the MDBS. The GTM can identify the reason which has caused the abort. This is because the GTM receives an abort operation from the transaction, whenever the abort is required by the transaction.

We have observed that the recovery protocol for global transaction failures can be optimized if the following design decision is used: specific recovery actions should be defined for each situation in which a global transaction abort occurs. Therefore, we have designed recovery actions which should be triggered when the global transaction requires the abort, and recovery actions for coping with aborts which occur on behalf of the MDBS.

Aborts Required by Transactions

Since we assume that updates of a global transaction G_i may be viewed by other transactions, we can not restore the database state which existed before the execution G_i, if G_i aborts. This implies that the standard *transaction undo* [15] action can not be used in such a situation. However, the effects of a global transaction must be somehow removed from the database, if it aborts.

For that reason, we need a more adequate recovery paradigm for such an abort scenario. This new recovery paradigm should primarily focus on the fact that the effects of transactions which have accessed the objects updated by an aborted global transaction G_i and database consistency should be preserved, when removing the effects of G_i from the database. The key to this new recovery paradigm is the notion of *compensating transactions* [10, 17, 18].

A compensating transaction CT "undoes" the effect of a particular transaction T from a semantic point of view. That means, CT does not restore the physical database state which existed before the execution of the transaction T. The compensation guarantees that a consistent (in the sense that all integrity constraints are preserved) database state is established based on semantic information, which is application-specific.

By definition, a compensating transaction CT_i should be associated with a transaction T_i and may only be executed within the context of T_i. That means that the existence of CT_i depends on T_i. In other words, CT_i may only be executed, if T_i has been executed before. Hence, CT_i must be serialized after T_i. We will assume that persistence of compensation is guaranteed, that is, once the compensating action has been started, it is completed successfully.

For our purpose the concept of compensation is realized as follows. For a given transaction G_i consisting of subsequences $SUB_{i,1}$, $SUB_{i,2}$, ..., $SUB_{i,n}$, a global compensating transaction CT_i is defined, which in turn consists of a collection of local compensating transactions $CT_{i,k}$, $0 < k \leq n$. Each local compensating transaction $CT_{i,k}$ is associated to the corresponding subsequence $SUB_{i,k}$ of transaction G_i. Of course, $CT_{i,k}$ must be performed at the same local site as does $SUB_{i,k}$ and must be serialized after $SUB_{i,k}$.

Now, we are in a position to describe the recovery strategy for aborts required by transactions. When the GS receives an abort request from a global transaction G_i, the GS forwards this operation to the GRM. The GRM reads the global log in order to identify which subsequences of G_i are still active. For each active subsequence, the GRM sends a local abort operation to the servers responsible for the execution of the subsequence. The GRM then waits for an acknowledgment from these servers confirming that the subsequences were aborted. After that, the GRM triggers the corresponding local compensating transactions for every subsequence which has already been locally committed. This information can be retrieved from the global log file. Operations of the compensating transactions are scheduled by the GS.

Therefore, the execution of local compensating transactions will undo the effect of committed subsequences from a semantic point of view.

Since we have assumed that the LDBMSs implement 2PL to enforce local serializability, the compensation mechanism described above satisfies the following requirement. A particular transaction T (subsequence or local transaction) running at a local system either views a database state reflecting the effects of an updating subsequence $SUB_{i,k}$ or it accesses a state produced by the compensating transaction of $SUB_{i,k}$, namely $CT_{i,k}$. In other words, T can not access objects updated by $SUB_{i,k}$ and by $CT_{i,k}$. Such a constraint is required for preserving local database consistency.

Thus far, we have assumed that the effect of any transaction can be removed from the database by means of a compensating transaction. However, not all transactions are compensatable. There are some actions, classified by Gray as *real actions* [12], which present the following property: once they are done, they can not be undone anymore. For some of these actions, the user does not know how they can be compensated, that is, the semantic of such compensating transactions is unknow. For instance, the action *firing a missile* can not be undone. Moreover, the semantic of a compensating transaction for this action can not be defined. For that reason, we say that transactions involving such real actions are not compensatable.

In order to overcome this problem, we propose the following mechanism. The execution of local commit operations for non-compensatable subsequences should be delayed until the GTM receives a commit for the global transaction containing the non-compensatable subsequences. This mechanism requires that the following two conditions are satisfied.

First, the user should specify which subsequences of a global transaction are non-compensatable[3].

[3]When it is not specified that a subsequence is non-compensatable, it is assumed that the subsequence is compensatable.

This is a reasonable requirement, since our recovery strategy relies on a compensation mechanism. This latter mechanism presumes that the user defines compensation transactions, when he or she is designing transactions. Hence, the user can identify at this point, which subsequences of a global transaction may not be compensatable.

Second, the information identifying which subsequences are non-compensatable should be made available to the GTM. For instance, the GTM can be designed to receive this information as an input parameter of subsequences.

The procedure of delaying the execution of local commit operations for non-compensatable subsequences can be realized according to the following protocol:

1. When the GTM receives the first operation of a particular subsequence, it must identify whether the subsequence is compensatable. If the subsequence is non-compensatable, the GTM saves this information in the log record of the subsequence. The log record should be stored in the global log file.

2. If the GTM receives a local commit operation for a non-compensatable subsequence, it marks the log record of the subsequence stored in the global log with a flag. This flag captures the information that the local commit operation for the subsequence can be processed when the global transaction is to be committed.

3. Whenever the GTM receives a commit operation for a given global transaction G_i, it verifies in the global log if there are local commit operations to be processed for subsequences of G_i. This can be realized by reading the log records of all subsequences belonging to G_i.

Following this protocol, we ensure that the effects of non-compensatable subsequences are reflected in the local databases only when the global transaction is to be committed. This eliminates the possibility of undoing the effect of such subsequences.

Unfortunately, this mechanism has the following disadvantage. Locks held by non-compensatable subsequences can only be released when the global transaction completes its execution.

Another drawback of the compensation approach is the specification of compensating transactions for interactive transactions as, for instance, design activities. As a solution for overcoming such a problem, we propose the following strategy.

When an interactive global transaction G has to be aborted and G has some locally committed subsequences, the GTM reads the global log file in order to identify which subsequences of G were already locally committed. After that, the GTM notifies the user that the effects of some subsequences of G must be "manually" undone. The GRM informs which subsequences should be undone and what operations these subsequences have executed. Moreover, the GRM informs the user on which objects these operations have been performed.

The user then starts another transaction in order to undo the effect of such subsequences. Objects updated by these subsequences may have been viewed by other global transactions. For that reason, the user must know which global transactions have read these objects. With this knowledge the user can notify other designers that the values of the objects x,y,z they have read (the GRM has provided this information) are invalid.

Aborts on Behalf of the MDBS

Usually, such aborts occur when global transactions are involved in deadlocks. Deadlocks are provoked by transactions trying to access the same objects with conflicting locks. Committed subsequences have already released their locks. Besides this, they are not competing for locks anymore. Hence, operations of such subsequences can neither provoke nor be involved in deadlocks.

This observation has an important impact in designing recovery actions to cope with transaction aborts required by the MDBS. It is not necessary to abort entire global transactions to resolve deadlock situations. Aborting active subsequences is sufficient. However, we need to replay the execution of the aborted subsequences in order to ensure commit atomicity. This implies that new results may be produced by the resubmission of the subsequences. In such a situation, the user must be notified that the subsequences were aborted and, for that reason, they must be replayed, which may produce different results from those he/she has already received. With this knowledge, the user can decide to accept the new results or to abort the entire global transaction.

Observe that, if the original values read by the failed subsequences were not communicated to other subsequences (those reads may be invalid), the resubmission of the aborted subsequences will produce no inconsistency in the execution of entire global transaction. Such a requirement is reasonable in a multidatabase environment.

Based on these observations, we propose the following strategy for dealing with global transaction aborts which occur on behalf of the MDBS.

When the GTM (or another component of the MDBS) decides to abort a transaction G_i, the GRM must be informed that G_i has to be aborted. When the GRM receives this signal, it verifies in the global log which subsequences of G_i are still active. For each active subsequence, the GRM sends a local abort operation to the servers (through the GS, of course) responsible for the submission of these subsequences to the local systems. In the meantime, the GRM waits for an acknowledgment from the servers confirming the local aborts of the subsequences. Furthermore, the GRM sends to the user responsible for the execution of G_i the notification informing that some subsequences of G_i have to be aborted and they will be replayed. The GRM is able to inform the user which operations have to be re-executed. The user can then decide to wait for the resubmission of the aborted subsequences or to abort the entire global transaction. If the user decides to abort the entire global transaction, the process of replaying the subsequences is cancelled and the recovery protocol for *global transaction failure requested by the transaction* is triggered. Otherwise, the recovery protocol for global transaction aborts which occur on behalf of the MDBS goes on as described below.

When the GRM has received the acknowledgments that the subsequences were aborted in the local DBMSs, the GRM starts to replay the execution of each aborted subsequence $SUB_{i,k}$. For that purpose, the GRM must read from the global log file the log record which contains information about the *installation point* of each subsequence to be replayed. This record can be identified by the fields SUBID and LRT. Observe that LRT must have the value ‘IP’.

6.1.2 Subsequence Failures

As mentioned before, a subsequence of a particular global transaction may abort for many reasons. However, there are two situations of subsequence aborts which should be handled in a different manner. The first situation is when the subsequence is aborted on behalf of the local DBMS. The second situation is when the subsequence decides to abort its execution. In this section, we describe a recovery method to deal with these two subsequence abort situations.

Aborts on Behalf of the Local DBMS

Typically, DBMSs decide to abort subsequences, when such subsequences are involved in local deadlocks. After such aborts, the effect of failed subsequences are undone by the LDBMSs. Locks held by the aborted subsequences are released. As soon as the server recognizes that a particular subsequence has been aborted by the local DBMS, the server reads the server log file and retrieves the log records of the aborted subsequence. The server stores a new log record for the subsequence with LRT=‘ST’ in the server log file. Moreover, the server sends a message to the GTM reporting that the subsequence has been aborted by the LDBMS. The GRM forces a record log of the failed subsequence to the global log file. By doing this, the new state of the subsequence is stored in the global log file as well.

After that, the server forces a log record with the new state of the subsequence to the server log and starts the resubmission of the aborted subsequence. As already seen, new results may be produced by such a resubmission. However, we propose a notification mechanism which gives the user the necessary support to decide for accepting the new results or for aborting the entire global transaction.

It is important to notice here that a given subsequence $SUB_{i,k}$ belonging to a global transaction G_i

may have more than one log record with LRT='ST' (in each log file) during the execution of G_i. For such a subsequence, only the last record with LRT='ST' should be considered.

Aborts Required by Subsequences

When the subsequence identifies some internal error condition (e.g., violation of some integrity constraints or bad input), it aborts its execution. Sometimes the resubmission of the subsequence is sufficient to overcome the error situation. However, we can not guarantee that the subsequence will be committed after being resubmitted a certain number of times. This is because the abort is caused when some internal error condition occurs (e.g. division by 0). Hence, it is impossible to predict whether or not the same problem will occur in a repeated execution of the subsequence. In this case, the solution is to abort the complete global transaction. The user or the GTM should be able to make such a decision.

Observe that, when an internal error occurs, it is necessary that the subsequence reads new values (new input) and produces new results in order to overcome the internal error condition. Based on this observation, we propose the following actions for dealing with aborts required by subsequences.

When the subsequence decides to abort its execution, an explicit abort operation is submitted to the GS, which in turn sends this operation to the GRM. The GRM then writes a new log record with LRT='ST' for the subsequence in order to reflect its new state. Thereafter, the GRM forwards the abort operation to the server. In turn, the server forces a log record with the new state of the subsequence to the server log and submits the abort operation to the LDBMS. After the subsequence is aborted by the LDBMS, the GTM resubmits the aborted subsequence to the LDBMS.

6.2 Local System Failures

Local DBSs reside in heterogeneous and autonomous computer systems (sites). When a system failure occurs at a particular site, we assume that the LDBMS is able to perform recovery actions in order to restore the most recent transaction-consistent state. These actions are executed outside the control of the MDBS. After an LDBMS completes the recovery actions, the interface server assumes the control of the recovery processing. While the server is executing its recovery actions, no local transaction can be submitted to the restarted DBMS.

Before describing the strategy for recovering from local system failures, we need to define the states of a subsequence in a given server. A subsequence may present four different states in a server. A subsequence is said to be *active*, when no termination operation for the subsequence has been submitted to the local DBMS by the server. When the server submits a commit operation, the subsequence enters the *to-be-committed* state. If the commit operation submitted by the server has been successfully executed by the local DBMS, the subsequences enters the *locally-committed* state. When the subsequence aborts, it enters the *locally-aborted* state. Figure 2 illustrates the states of a subsequence in servers.

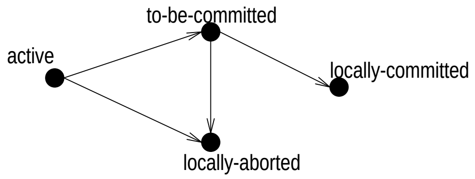

Figure 2: States of a particular subsequence in a server.

After the server recognizes that the local system has failed, it waits for an acknowledgment of the local system reporting that the local recovery actions were completed. The server then starts to execute its recovery actions. These actions consist of an *analysis pass* and a *redo pass* over the server log file. The analysis pass reads the server log sequentially in order to identify subsequences which were *active* or *to-be-committed* before the occurrence of the failure. For each subsequence which was *to-be-committed*, the server verifies if this subsequence was considered a loser transaction by the local DBMS (recall that we are assuming that the server can obtain this information). The effects of such subsequences were undone during the execution of the local recovery actions. The analysis pass builds a table, denoted AST (active subsequence table) which contains SUBID and LSN of the first log record of subsequences whose effects were undone by the local recovery actions. At the end of the analysis pass, the server knows for which subsequences it has to rebuild the effects in the local database.

The redo pass starts from the minimum of the LSN in the table constructed during the analysis pass. The server scans the server log. The write operations are resubmitted to the local DBMS whenever the log scan encounters a write-operation log record for a subsequence in the table AST. By doing this, the effects of such subsequences are deterministically rebuilt in the local database. This is because no local transactions are executed until the server has completed its recovery actions.

When the server completes the execution of the redo pass, it sends a message to the local DBMS reporting this fact. The local DBMS can then resume the execution of local transactions.

Note that, when a local system fails, the global lock manager does not release the locks held by subsequences executed in the failed system.

6.3 GTM Failures

Such failures can be caused by a bug in the GTM code, a hardware failure or an operating system fault at the site where the GTM is running. A failed and restarting GTM should execute recovery actions to guarantee that LDBMSs reflect the effects of committed subsequences, although the respective global transactions have not yet been committed. This implies that the GRM must avoid to undo global transactions which were not committed when the failure has occured.

A particular subsequence at the GTM's site may present one of the following states: *active*, *to-be-committed*, *committed* or *aborted*. It is important to notice here that a global transaction presents only the *active*, *committed* or *aborted* states. In order to avoid to undo global transactions after GTM failures, the GRM must manage information about global-transaction and subsequence states as follows. A subsequence is said to be *active*, when the GRM receives a message from the GS reporting that a local initiation operation (BOS) for that subsequence has arrived. This information is then stored in the global log by the GRM. When the GS receives a local commit operation, it forwards this operation to the GRM. In turn, the GRM saves an information in the global log to reflect the new state (*to-be-committed*) of the subsequence. If the GRM receives an acknowledgment indicating that the local commit operation has been executed successfully by the local DBMS, it updates the information about the state of the subsequence. This is necessary, because the subsequence has passed to the *committed* state. When the subsequence aborts, it enters the *aborted* state and, thereby, the global log should be updated.

Observe that a particular subsequence may present different states at the GTM's site and in the server. To illustrate this fact, consider that the GS has received an lc operation for a subsequence $SUB_{i,k}$. The GS forwards this operation to the GRM, which in turn saves the information that $SUB_{i,k}$ has passed to the *to-be-committed* state. However, at this moment, $SUB_{i,k}$ is active with regard to the server.

When an initiation operation for a given transaction G_i arrives at the GS, it informs the GRM that an initiation operation for G_i has arrived. The GRM, in turn, records the information that G_i is *active* in the global log. When all subsequences of G_i have reached the committed state and a global commit operation for G_i has arrived, G_i enters the *committed* state. When a global abort operation is received, the GRM saves this fact by storing the information that the transaction has entered the *aborted* state.

Now, we are in a position to describe recovery actions for GTM failures. We will consider that global applications run at clients which interact with the GTM. Thus, global applications and the MDBS run in a client-server environment. We will assume that, when a GTM failure occurs, the execution of global applications is interrupted at the clients, but not aborted.

GTM restart consists of an analysis pass and and a redo pass over the global log file. During the analysis pass, the GRM scans all log records until the end of the log and builds a table, denoted AST (active subsequence table), which has the same structure as the table used for recovering from local system failures (see Section 6.2).

The redo pass consists of the following actions. The GRM sends to the GS all operations of subsequences in the table AST. By doing this, the execution scenario (for the *active* subsequences) which existed before the failure is recreated. For instance, the global lock table is recreated. This is possible, because the global log file represents the execution history of global transactions. After the redo pass is completed, the execution of global applications (which has been interrupted at the clients) can be resumed.

6.4 Server Failures

Servers act as interfaces between an MDBS and the LDBSs. From a transaction processing standpoint, their primary function is to control the submission of sequences of operations belonging to global transactions to the local DBMS. Thus, when a server associated to a local system $LDBS_k$ fails, subsequences of global transactions submitted to $LDBS_k$ can not be executed, although $LDBS_k$ is still operational. We assume that the GTM can identify when a given server has failed.

The protocol for handling server failures is the following:

1. When the GTM recognizes that a server has failed, it aborts the execution of all *active* subsequences which were being executed in the failed server. Log records (with LRT='ST') for the aborted subsequences are forced to the global log file in order to store the information that these subsequences have passed from the active to the aborted state. Moreover, the GTM stops submitting operations to that server. In order to decide what kind of recovery actions should be performed for *to-be-committed* subsequences, the GTM must wait until the server has been restarted, since the GTM must know whether the subsequence was successfully committed by the local DBMS.

2. After the server is restarted, it should trigger the following recovery procedures:

   (a) The server log is sequentially read. For each subsequence which was *active* immediately before the occurrence of the failure, the server sends an abort operation to the local DBMS. If the subsequence was *to-be-committed*, the server may query the external interface of the local DBMS in order to know whether or not the subsequence was successfully committed by the local DBMS. The server then forwards this information to the GTM.

   (b) The server log must be updated. For instance, if a particular *to-be-committed* subsequence was aborted by the local DBMS before the occurrence of the failure, the server writes a record in the server log file in order to capture this information.

   (c) After the server log is read and updated, the server sends a message to the GTM informing that it is in operation.

3. When the GRM receives a message from the server reporting that it is operational, the GRM replays the aborted subsequences. After that, the recovery procedure for server failure is completed.

6.5 Communication Failures

The components of an MDBS are interconnected via communication links. Typically, communication failures break the communication among some of the components of an MDBS. According to Figure 2, there may be two types of communication links in MDBSs. One type of link, which we call Server-LDBS link, connects servers to local systems. If the interface servers are not integrated with the GTM, that is, each server resides at a different site from the GTM site, the other type of link connects the GTM to servers. Such a communication link is denoted GTM-Server link. We propose different recovery strategies for handling failures in each type of communication link.

In order to enable MDBSs to cope with communication failures, the following requirement must be satisfied. Each server in an MDBS must know the *timeout period* of the local DBMS with which the server is associated. We also assume that each server has its own timeout period and this timeout period is larger than the timeout period of the respective local DBMS.

Failures in Server-LDBS links

In such failures, the link between a particular local system and a server is broken. The local system and the server will continue to work correctly. Such a situation can lead the local system to abort the execution of some subsequences (which are being executed at the local system) by timeout.

For coping with communication failures between a server and a local system, we propose the following strategy.

If the communication link is reestablished before the timeout period of the local DBMS is reached, no recovery action is necessary. This is because no subsequence was aborted by timeout.

In the case that the communication link is reestablished after the timeout period of the local DBMS is reached, but before the timeout period of the server, the following recovery actions should be performed by the server:

1. The server scans the server log file. During the scan process, the following recovery actions should be performed.
   (a) For each subsequence which was *active* before the occurrence of the failure, the server executes recovery actions, since such subsequences were aborted by the LDBMS by timeout. These recovery actions are the same as those which should be performed for recovering from subsequence failures required by LDBMSs (see Section 6.1).
   (b) If the subsequence was *to-be-committed*, the server may query the external interface of the LDBMS in order to know whether the subsequence was successfully committed. In this case, the server performs actions to confirm the fact that the subsequence was committed (for instance, log records with LRT=‘ST’ must be forced to the server log and global log files). Otherwise, it considers the subsequence as locally aborted and performs actions for recovering from subsequence failures required by LDBMSs.

If the timeout period of the server is reached before the communication link is reestablished, the server sends a message to the GTM reporting that it can not process subsequences anymore. After that, the GTM aborts the execution of all subsequences which were being executed in the failed server. Log records for the aborted subsequences are stored in the global log file with their new state (aborted). The GTM stops submitting operations to that server. If the communication link is reestablished before the timeout period of the GTM is reached, recovery actions for recovering from server failures are executed.

If the timeout period of the GTM is reached before the Server-LDBS link is reestablished, the global log file is sequentially read. For each global transaction which has submitted a subsequence to the server whose Server-DBMS link is broken, the subsequence's log record with LRT=‘ST’ is read. If the subsequence is *active* or *to-be-committed*, the GRM aborts the global transaction. In this case, recovery actions for global transaction recovery should be triggered.

Observe that a subsequence which was submitted to the server with a broken Server-LDBS link and has a *to-be-committed* state in the global log may have been committed by the local DBMS. In this case, after the link is reestablished, the server must be able to query the external interface of the LDBMS to know whether or not the subsequence was successfully committed. If the subsequence was committed, a compensating transaction for such a subsequence should be executed.

Failures in GTM-Server links

Of course, such a failure has only to be considered, if it is assumed that the interface servers reside at different sites from the GTM's site.

When a failure in the GTM-Server link occurs, the link between the GTM and a server is broken. In order to enable MDBSs to cope with failures in GTM-Server links, we propose the strategy described below.

Without loss of generality, consider that the link between the GTM and the server $Server_k$ is broken. $Server_k$ is associated with local system $LDBS_k$. If the communication link is reestablished after the timeout period of $LDBS_k$ is reached, but before the timeout period of the server, the following actions are performed:

- The server log is sequentially read.

  1. For each subsequence which was *active* before the failure, the server executes recovery actions for subsequence failures required by local DBMSs, since such transactions were aborted by the local DBMS (timeout).
  2. If the subsequence was *to-be-committed*, the server may query the external interface of the local DBMS in order to know whether the subsequence was committed. In this case, the server performs the actions to reflect the fact that the subsequence was locally committed. Otherwise, it performs actions for recovering from subsequence failures required by local DBMSs.

If the link is reestablished after the timeout period of $Server_k$, but before the timeout period of the GTM is reached, actions for recovering from server failures are started.

If the timeout period of the GTM is reached before the link is reestablished, the GRM reads the global log in order to identify active global transactions which have submitted a subsequence to $Server_k$ whose link with the GTM is broken. For each global transaction satisfying this condition, the GRM verifies the state of the subsequence submitted to $Server_k$. If the subsequence was *active* or *to-be-committed*, the GRM aborts the global transaction. Recovery actions for global transaction recovery should be triggered. A subsequence which has a *to-be-committed* state in the global log may have been committed by the local DBMS. In this case, after the communication link is reestablished, the server must be able to query the external interface of the LDBMS in order to know whether or not the subsequence was successfully committed. If the subsequence was committed, a compensating transaction for such a subsequence should be executed.

7 Conclusions

In this paper, we have presented strategies to guarantee reliability in the transaction processing in MDBSs. In order to describe these strategies, we have first categorized and analyzed the critical failures which may occur in a multidatabase environment.

We have then proposed an extended model for an MDBS architecture which provides the necessary support for the execution of recovery actions after failures. A key feature of this architecture is that it makes viable the distribution of recovery actions among the diverse components of an MDBS. By doing this, we optimize the recovery process, since each component can perform recovery actions without interfering in the processing of components which were not involved in failures.

Furthermore, we have proposed a commit protocol. The main goal of this protocol is to ensure that, when a global transaction ends its execution, it presents the same state (committed or aborted) at every site it has run.

Finally, we have described recovery actions to cope with failures we have classified in this work. The recovery strategy, denoted ReMT, consists of a collection of recovery protocols, which are distributed among the components of an MDBS.

The key advantages of the ReMT strategy are:

- It can reduce the frequency of global transaction undo after the occurrence of failures, and
- it is able to deal with several types of failures which may occur in a multidatabase environment.

Although it seems to be obvious that a recovery mechanism for an MDBS should have these properties, most of the existing MDBSs (prototypes or commercial products) are not capable to achieve them. For instance, MYRIAD (University of Minnesota), MERMAID (Unisys Corporation), Pegasus (Hewlett-Packard) do not provide these properties. In [4], the reader finds a detailed discussion about existing MDBSs.

References

[1] Özsu, M. T. and Valduriez, P. Distributed Database Systems: Where Are We Now? *IEEE Computer*, 24(8):68–78, August 1991.

[2] Alonso, R., Garcia-Molina, H., Salem, K. Concurrency control and recovery for global procedures in federated database systems. *A quartely bulletin of the Computer Society of the IEEE technical comittee on Data Engineering*, 10(3), 1987.

[3] Bernstein, P. A., Hadzilacos, V. and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[4] Bouguettaya, A., Benatallah, B. and Elmagarmid, A. *Interconnecting Heterogeneous Information Systems.* Kluwer Academic Publishers, 1998.

[5] Brayner, A. *Transaction Management in Multidatabase Systems.* PhD thesis, University of Kaiserslautern, Germany, 1999.

[6] Brayner, A., Härder, T. and Ritter, N. Semantic Serializability: A Correctness Criterion for Processing Transactions in Advanced Database Applications. *Data & Knowledge Engineering*, 31(1):1–24, 1999.

[7] Breitbart, Y., Garcia-Molina, H., Silberschatz, A. Overview of multidatabase transaction management. *The VLDB Journal*, (2):181–239, 1992.

[8] Elmagarmid, A. K., Jing, J., Kim, W. and Bukhres, O. Global Committability in Multidatabase Systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):816–824, October 1996.

[9] Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[10] Garcia-Molina, H. and Salem, K. . SAGAS. In *Proceedings of the ACM SIGMOD Conference*, pages 249–259, 1987.

[11] Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann, 1993.

[12] Gray, J.N. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on VLDB*, pages 144–154, 1981.

[13] Hadzilacos, V. A Theory of Reliability in Database Systems. *Journal of the ACM*, 35:121–145, 1988.

[14] Härder, T. and Rahm, E. *Database Systems: Implementation Concepts and Techniques (In German).* Springer-Verlag, 1999.

[15] Härder, T. and Reuter, A. Principles of Transaction-Oriented Database Recovery. *Computing Surveys*, 15(4):287–317, 1983. Reprinted in: Readings in Database Systems, Third Edition M. Stonebraker, J. Hellerstein (eds.), Morgan Kaufmann Publishers, March 1998.

[16] Härder, T. and Rothermel, K. Concepts for Transaction Recovery in Nested Transactions. In *Proceedings of the ACM SIGMOD Conference*, pages 239–248, 1987.

[17] Korth, H. F., Levy, E. and Silberschatz, A. A Formal Approach to Recovery by Compensating Transactions. In *Proceedings of the 16th International Conference on VLDB*, pages 95–106, 1990.

[18] Levy, E., Korth, H. F. and Silberschatz, A. An Optmistic Commit Protocol for Distributed Transaction Management. In *Proceedings of the 1991 ACM SIGMOD Conference*, pages 88–97, May 1991.

[19] Litwin, W., Mark, L. and Roussopoulos, N. Interoperability of Multiple Autonomous Databases. *Computing Surveys*, 22(3):267–293, 1990.

[20] Lomet, D. and Weikum, G. Efficient Transparent Application Recovery In Client-Server Systems. In *Proceedings of the 1998 ACM SIGMOD Conference*, 1998.

[21] Schek, H. J., Weikum, G. and Ye, H. Towards a Unified Theory of Concurrency Control and Recovery. In *Proceedings of the ACM Symposium on PODS*, pages 300–311, 1993.

[22] Sheth, A. P. and Larson, J. A. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *Computing Surveys*, 22(3):183–236, 1990.