SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(1), 77–98 (JANUARY 1998)

Semantic ACID Properties in Multidatabases Using Remote Procedure Calls and Update Propagations

lars frank[1] and torben u. zahle[2]
[1]*Institute of Informatics and Management Accounting, Copenhagen Business School, Howitzvej 60, DK-2000 Frederiksberg, Denmark*
(e-mail: frank@CBS.DK)
[2]*Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark*
(e-mail: zahle@diku.dk)

SUMMARY

Global ACID properties (Atomicity, Consistency, Isolation and Durability) may be implemented by using a DDBMS (Distributed Data Base Management System.) However, in this solution data availability is low. Further, data may be blocked, i.e. if some data are locked from a remote location, the data cannot always be unlocked when the connection to the data fails. This is not a problem when client/server technology is used because client/server technology only uses local locks, a reason why multidatabases and client/server technology are widely used in real-life distributed systems. However, the trouble with such systems is that they have no inherent global ACID properties. The objective of this paper is to illustrate how global semantic ACID properties, enforced by the transactions themselves, may be implemented on top of client/server technology. This is done to preserve high data availability. The global atomicity property is implemented by using retriable and compensatable subtransactions. The global consistency property must be implemented by the transactions themselves. The global isolation property is implemented by using countermeasures to isolation anomalies. The global durability property is implemented by using the durability property of the local DBMSs. The largest bank in Denmark, Den Danske Bank, has implemented all its applications using methods described in this paper. © 1998 by John Wiley & Sons, Ltd.

key words: multidatabases; concurrency control; atomicity; isolation levels; availability

INTRODUCTION

A *multidatabase system* (MDBS) is a facility that allows users to access data located in multiple autonomous Data Base Management Systems (DBMS.) A *multidatabase* is a union of local databases. *Global transactions* access data located in more than one local database. A *local DB-transaction* is executed under the control of a single DBMS with local ACID properties.[1] Global transactions can be managed by a *Global*

CCC 0038–0644/98/010077–22 $17.50 *Received 2 January 1996*
© 1998 by John Wiley & Sons, Ltd. *Revised 10 December 1996 and 13 August 1997*

Transaction Manager (GTM)[2] belonging to a Distributed Data Base Management System (DDBMS) *or* a MDBS product. In this case, global ACID properties are implemented by the GTM in cooperation with the local DMBSs. Client/server systems, on the other hand, have no GTM, and therefore the local DB-transactions may communicate in an uncoordinated way. This gives no global ACID properties.

In this paper we describe how global ACID properties may be implemented without using a GTM. In practice, *semantic ACID properties* will do, i.e. we only implement as much of the ACID properties as the applications require. (Normally, it is the isolation property that we can dispense with from an application point of view.)

To our knowledge, there is at present no commercial MDBS product that ensures both high availability of data, high performance and semantic ACID properties of transactions. Therefore, many companies have ‘invented’ their own solutions using the client/server architecture. The primary objective of this paper is to help such companies to implement global semantic ACID properties. By using high level tools like Remote Procedure Calls (RPC) and Update Propagations (UP) it is possible to reduce the implementation cost. When transactions are executed without the isolation property, the following isolation anomalies may occur:[1,3]

(a) The lost update anomaly.
(b) The dirty read anomaly.
(c) The non-repeatable read anomaly.
(d) The phantom anomaly.

As we do not use a GTM, the isolation anomalies have to be dealt with by using transaction managed countermeasures. (That is, not measures to obtain full isolation, but countermeasures to isolation anomalies.) In the following, we will therefore call our transaction model *The Countermeasure Transaction Model.*

Since there is a conflict between high availability of data and isolation, it is up to the transaction designers to select as little protection against isolation anomalies as feasible in order to get as high an availability as possible. Normally, all updating transactions must use countermeasures against the lost update anomaly. But for many applications it is enough to select countermeasures that reduce the risk of using ‘dirty’ data or ‘non-repeatable’ data. Of course, it is necessary to know the semantics of the applications to select the proper countermeasures for each transaction type. It is, indeed, possible to solve all the isolation anomalies by using countermeasures over a broad front. However, this cannot be recommended, because if all isolation anomalies are solved, the availability of the data will decrease, and it would be easier to use normal concurrency control. (If some transactions need all the global ACID properties, it is possible to execute these transactions with a DDBMS and intersperse them with transactions that have the global semantic ACID properties described in this paper.)

The paper is organized as follows: Section 2 will describe The Countermeasure Transaction Model, i.e. we will give an overview of how the global semantic ACID properties can be implemented. In section 3, we illustrate by examples how the isolation anomalies often may be reduced or eliminated by using countermeasures. In section 4 we describe how the global atomicity property may be implemented and integrated with countermeasures. In section 5, we describe different methods for implementing Update Propagations, (because in standard systems this high level tool is often implemented in such a way that it cannot be used in The Countermeasure Transaction Model.) Concluding remarks and suggestions for further research are presented in sections 6 and 7.

Related Work

Atomicity by using compensating transactions was first introduced by Garcia-Molina and Salem in 1987.[4] Transaction managed atomicity by using the enforcement method was to our knowledge first described by Frank in 1985[5] and 1988.[6] The two atomicity methods were then integrated in the pivot transaction models by Mehrotra in 1992[7] and Zhang in 1994.[8] The systematic analysis of countermeasures, described in this paper, was not possible until the isolation property was decomposed into disjunctive isolation anomalies by Gray and Reuter in 1993,[1] and Berenson *et al*. in 1995.[3]

THE COUNTERMEASURE TRANSACTION MODEL

In this model a global transaction consists of a *root transaction* (client transaction) and several single site *subtransactions* (server transactions) which execute *stored procedures* or *stored programs*. The subtransactions themselves can be nested transactions, i.e. a subtransaction may be a *parent transaction* for other subtransactions. In The Countermeasure Transaction Model, all communication with the user is managed from the root transaction, and all data is accessed through subtransactions. Data stored in the location of the root transaction is also accessed through subtransactions, as dictated by the client/server architecture. A subtransaction may consist of several DB-transactions separated from each other by local commit orders. Normally a subtransaction only contains a single DB-transaction, and in this situation we normallly do not distinguish between a subtransaction and its contained DB-transaction.

All subtransactions are accessed through the following two types of high-level tools:

(a) **Remote Procedure Call (RPC)**
There is not yet a standard for RPCs, but from a programmer's point of view, they function like normal procedure calls, except that the procedure call and the procedure itself are stored at different sites.

When a parent transaction executes a RPC it waits for the subtransaction to be executed, and when the subtransaction is finished, the control/answer is returned to the parent transaction. If a parent transaction executes more RPCs, the corresponding subtransactions are executed one at a time (contrary to UP-subtransactions.) At the remote site the subtransaction is executed as a transaction with local ACID properties.

Therefore, a RPC can lead to the following problems:

(i) If a parent transaction does not get an answer from a subtransaction, the parent transaction will not know whether the subtransaction has been committed or aborted.

(ii) If a parent transaction fails after the execution of a RPC, the parent transaction is aborted, while the subtransaction is still committed.

In both cases, we have not achieved global atomicity, and therefore the following *UP tool* is needed.

(b) **Update Propagation (UP)**
Update propagation is here used in the general sense of propagation of any update (not just replicas.) The UP tool works in the following way. When a remote stored program is executed using update propagation, the 'call' with parameters is stored on persistent storage (a local DBMS file.) An underlying system (the UP tool) will later read the 'call' and execute the related stored program. If the execution of the stored program fails, the UP tool will restart the stored program until it is committed. We say that the stored program is *enforceable*. The subtransaction will be executed and committed if and only if the parent transaction making the UP has previously been committed.

The parent transaction makes the UP by storing a so-called *transaction record* in persistent storage of the parent location. The parent transaction id, the id of the subtransaction and the parameters of the subtransaction are stored in the transaction record. If the parent transaction fails, the transaction record will be rolled back, and consequently the subtransaction will not be executed. When the parent transaction is committed, the transaction record is secured in persistent storage, and we say that the UP is *initiated*. After the initiation of the UP, the transaction record will be read and sent by the UP tool (which uses a communication protocol) to the location of the corresponding subtransaction. If the subtransaction fails, the transaction record is resubmitted until the subtransaction is committed.

When a parent transaction executes several UPs, the corresponding subtransactions may be executed in parallel.

In the following, we will give an overview of how the global semantic ACID properties are implemented in The Countermeasure Transaction Model by using RPCs and UPs.

The Atomicity Property

An updating transaction is called *atomic* if either all or none of its updatings are executed. In the Countermeasure Transaction Model, the global atomicity property is implemented by using first several compensatable subtransactions, then one pivot subtransaction and finally several retriable subtransactions, as described in the *pivot transaction models*.[7,8]

A subtransaction is *retriable* if its execution is guaranteed to commit locally after a finite number of submissions. The UP tool (described above) is used to resubmit the request for execution until the subtransaction is committed, i.e. the UP tools is used to force the retriable subtransaction to be executed.

Example 2.1
A deposit in a bank account may be implemented as a retriable subtransaction. If there are any restrictions involved in depositing money in the account, (or if the deposit account is closed), the money may be deposited in an error account, where the transaction later will be processed manually. If the retriable subtransaction fails, the request for the execution of the subtransaction will be resubmitted until it is accepted.

A subtransaction is *compensatable* if the effects of its execution can be compensated semantically[9] after the local commitment of the compensatable subtransaction. Compensation is done by executing a compensating subtransaction. The compensating subtransaction must be enforceable, so that the compensating subtransaction can be executed via an UP.

Example 2.2
A withdrawal from a bank account may be implemented as a compensatable subtransaction, because the withdrawal can be compensated by a compensating subtransaction, which deposits the same amount of money that was withdrawn from the account.

A subtransaction that is neither retriable nor compensatable is considered a *pivot* subtransaction. In The Countermeasure Transaction Model, only one pivot subtransaction can be used. The global transaction is committed when the pivot subtransaction is committed. If a global transaction does not by nature have a pivot subtransaction, the last compensatable or the first retriable subtransaction may be chosen as the pivot subtransaction. The compensatable subtransactions are committed locally before commitment of the pivot transaction, and all the retriable subtransactions are committed locally after the commitment of the pivot subtransaction. If this execution order is satisfied, the compensatable subtransactions of the global transaction may be regarded as a *Saga*.[4] If one of the compensatable subtransactions is aborted, compensation of the saga is implemented by executing compensating enforceable subtransactions.

If the pivot subtransaction aborts, all the compensating subtransactions must be executed.

If the pivot subtransaction has been committed, the retriable subtransactions will automatically be executed and committed, provided they are designed as subtransactions invoked by UPs in the pivot subtransaction, i.e. the UPs are initiated when the pivot subtransaction is committed.

Example 2.3
Let us suppose that an amount of money is to be moved from an account in one location to an account in another location. In such a case the global transaction may be designed as a root transaction that calls a compensatable withdrawal subtransaction and a retriable deposit subtransaction. Since there is no inherent pivot subtransaction, the withdrawal subtransaction may be chosen as pivot. In other words, the root transaction executed at the users PC may call a pivot subtransaction executed at the bank of the user, which has an UP that ‘initiates’ the retriable deposit subtransaction.

If the pivot withdrawal is committed, the retriable deposit subtransaction will automatically be executed and committed later. If the pivot subtransaction fails, the pivot subtransaction will be backed out by the local DBMS. In such a situation the retriable deposit subtransaction will not be executed, because the transaction record with the UP-parameters is commmitted/aborted together with the pivot subtransaction.

RPCs are always used to call the compensatable subtransactions and the pivot subtransaction, because for these subtransactions it is important that answers are returned to the root transaction. RPCs are also used, because the local commitment of these subtransactions are not mandatory from a global atomicity point of view.

(If any problems occur before the pivot commit, we can compensate the first part of the global transaction.)

After the commit/abort of the global transaction, all the remaining updatings are mandatory. Therefore, UPs are always used to execute the retriable and compensating subtransactions, which are always executed after the global commit/abort.

The Consistency Property

In The Countermeasure Transaction Model the global consistency property must be managed by the transactions themselves, e.g. local referential integrity may be managed by a local DBMS, while referential integrity between sites must be managed by the global transactions themselves.

The Isolation Property

In The Countermeasure Transaction Model the global isolation property is managed by countermeasures. It is a distinctive feature of our countermeasures that they are managed by the transactions themselves, i.e. by the transaction designers, whereas global concurrency control is managed by a GTM. For most transactions it is not necessary to select countermeasures to solve both the lost update anomaly, the dirty read anomaly, and the non-repeatable read anomaly. (See the examples in the section on ‘Global semantic isolation using countermeasures’.) If all the isolation anomalies have to be solved, we do not recommend using countermeasures, because the programming is as complex as programming your own *two phase locking protocol*.[1]

In designing countermeasures it is possible to use local locking, but all locks are released immediately after a subtransaction is committed/aborted locally. Such locking procedure we call *short duration locks*. *Long duration locks* are locks which are held until the global transaction is committed.[1] In the Countermeasure Transaction Model it is only possible to use long duration locks in the pivot subtransaction, because in compensatable subtransactions locks are obtained and released before the pivot commit, and in retriable subtransactions locks are obtained and released after the pivot commit. If short- and long-duration locks are used in this way it will be impossible to *block* data. (Data is *blocked* if it is locked by a subtransaction which loses connection to the ‘coordinator’ (the pivot subtransaction), which manages the global commit/abort decision.)

The Durability Property

The global durability property will automatically be implemented, as it is ensured by the log-system of the local DBMS systems.[2]

The Global Transaction Manager

The idea of using a GTM to manage the execution of the global transactions’ subtransactions is used in most transaction models, but for the following reasons we do not use a GTM in The Countermeasure Transaction Model:

(a) Our objective is to help transaction designers right now, and as the right type of GTM is not available yet, we focus on the UP tool, because it is relatively easy to make.

(b) Management of the compensatable/compensating subtransactions would be much simpler by using a GTM, but often it is possible (as in Den Danske Bank) to design all the global transactions by using only pivot and retriable subtransactions. In this situation the GTM is not important if one just has the UP tool.
(c) Normally, a GTM is used to manage the distributed concurrency control. In The Countermeasure Transaction Model this is solved by using transaction managed countermeasures.
(d) Client/server architecture consists of a set of tools (RPCs and gateways) and rules that specify how to use the tools. This architecture may allow flexible growth from a centralized database to a complex distributed database without using a GTM. The Countermeasure Transaction Model is a set of tools using the isolation theory and the single pivot transaction model to implement semantic ACID properties within the client/server architecture.

GLOBAL SEMANTIC ISOLATION USING COUNTERMEASURES

In this section we will first define the isolation anomalies and describe the situations where the anomalies may arise in The Countermeasure Transaction Model. After this, we generalize the isolation level concept to include countermeasures against the isolation anomalies. These countermeasures are described in the following subsections.

The Generalized Isolation Level Concept

If the atomicity property is implemented, but there is no global concurrency control, the following isolation anomalies may occur:[1]

(a) *The lost update anomaly*[3] is by definition a situation where a first transaction reads a record for update without using locks. After this, the record is updated by another transaction. Later, the update is overwritten by the first transaction. In the Countermeasure Transaction Model the lost update anomaly may be prevented, if the first transaction reads and updates the record in the same subtransaction using local ACID properties. Unfortunately, the read and the update are sometimes executed in different subtransactions belonging to the same parent transaction. In such a situation it is possible for a second transaction to update the record between the readings and the updatings of the first transaction. In the Countermeasure Transaction Model this may happen in the following situations:

(i) In a dialog a user often wants to see the old data before they are changed. Suppose the data are updated by a compensatable or retriable subtransaction. In order to prevent blocking the data in case of network failure the data must be read in one subtransaction before the dialog, the later updated by another subtransaction. (See example in section ‘The Enforcement Method’.)
(ii) Suppose a user wants to see the data before a pivot subtransaction updates the data. In this situation the data cannot be blocked, because the pivot subtransaction need not ask another location for permission to abort the global transaction. However, it may not be acceptable to

lock the old data between the readings and the updatings, as no one knows how long it takes the user to make the input data. Therefore, even data stored at the pivot location may be read and updated in different subtransactions. (This problem may also arise in central databases, and the solution described in section 'The Reread Countermeasure' using the reread countermeasure may be used in both central and distributed databases. See example in section 'The Enforcement Method'.)

(iii) The lost update anomaly may also occur if the first transaction executes two updatings of the same record, where the updatings are executed in different subtransactions. In the Countermeasure Transaction Model this may occur if the first transaction executes a compensatable subtransaction, which is later compensated by a compensating subtransaction. (See example in the section on 'The Compensation Method'.)

(b) *The dirty read anomaly*[3] is by definition a situation where a first transaction updates a record without committing the update. After this, a second transaction reads the record. Later, the first update is aborted (or committed), i.e. the second transaction may have read a non-existing version of the record. In The Countermeasure Transaction Model this may happen when the first transaction updates a record by using a compensatable subtransaction and later aborts the update by using a compensating subtransaction. If a second transaction reads the record before it is compensated, the data read will be 'dirty'.

(c) *The non-repeatable read anomaly* or *fuzzy read*[3] is by definition a situation where a first transaction reads a record without using locks. This record is later updated and committed by a second transaction before the first transaction is committed or aborted. In other words, we cannot rely on what we have read. In The Countermeasure Transaction Model this may happen when the first transaction reads a record that later is updated by a second transaction, which commits the record locally before the first transaction commits globally.

Gray and Reuter[1] prove that if the isolation anomalies are solved for all transactions, the execution will be serializable and the isolation property implemented.

In this paper (as in the serialization proof in Gray and Reuter[1]), we do not deal with the following anomaly:

(a) *The phantom anomaly*[3] is by definition a situation where a first transaction reads some records by using a search condition. After this, a second transaction updates the database in such a way that the result of the search condition is changed. In other words, the first transaction cannot repeat the search without getting a changed result.

(b) The *dirty write anomaly* is a special case of the lost update anomaly, and therefore the situation will not be dealt with.[3]

For many transactions it is not important from a semantic point of view if some of the anomalies listed above may occur. Therefore, the isolation level concept has been introduced in order to substitute isolation with availability. The *isolation level* of a transaction may be defined by how and to which extent the four anomalies listed above are solved. ANSI in 1992[10] defines four isolation levels, and each transaction type must select one of these isolation levels. The ANSI 1992 isolation levels are arranged in a sequence, where each isolation level has more isolation than the previous one. Berenson *et al*[3] introduce a number of new, practical isolation levels and an improved definition of the isolation level concept, where the isolation level of a transaction is defined by combining locking solutions to some of the isolation anomalies. In this paper, we generalize Berenson's isolation level concept by allowing both locking procedures and countermeasures to be used in protection against the anomalies, i.e. in The Countermeasure Transaction Model the *generalized isolation level* of a transaction is defined by the combination of locking procedures and countermeasures used by the transaction. By using this generalized isolation concept or Berenson's concept the isolation levels may be arranged in a hierarchy, where some isolation levels are incomparable, i.e. the isolation levels have only a partial ordering. However, this does not matter, because, in practice, it is only the transaction's semantic resistance towards the anomalies that counts.

When selecting countermeasures for each transaction type, the objective is not to serialize the global transactions, as it is much easier to serialize by using distributed concurrency control. Countermeasures are only recommended to solve or reduce the problems that arise when the execution is not serializable! Therefore countermeasures are not synchronized by a locking procedure or a time-stamp procedure as used in normal concurrency control.

The rest of this section describes countermeasures against the isolation anomalies listed above. The first four countermeasures may be used to prevent the lost update anomaly, which normally is the most important isolation anomaly.

The Commutative Updatings Countermeasure

Adding and subtracting an amount from an account are examples of commutative updatings. If a subtransaction only has commutative updatings, it may be designed as commutable with subtransactions that only have commutative updatings. This is a very important countermeasure, because both compensatable, compensating, and retriable subtransactions have to be commutative in order to prevent the lost update anomaly. The commutative updatings countermeasure has earlier been described in, e.g. *Sagas*,[4] and *Open Nested Transactions*.[11]

Example 3.1
A deposit may be designed as a retriable commutative subtransaction, where the subtransaction reads the old balance of the account by using a local exclusive lock, adds the deposit to the balance, and rewrites the account record. After this the retriable commutative subtransaction will commit locally.

This deposit subtransaction is commutable with other deposit and withdrawal subtransactions.

The Version File Countermeasure

A *version file* contains all the changes to another file. The records of the version file are the after-images of the updated records. The time-stamps of the updating transactions form the last part of the identification key of the records in the version file.

A version file may be used to design commutative replacement updatings, because the correct field value is always stored in the version with the latest time-stamp.

The idea of a version file may be generalized in such a way that instead of storing the after-images of the updated records, one may store the time-stamp, the id, and the parameters of the updating program. Such a generalized version file may be used to design commutative subtransactions, as it is possible to recalculate the image of the record by using the parameters stored in the generalized version file. Normally, the generalized version file may also be used to design retriable and/or compensatable subtransactions, as it may be possible to recalculate the image of the record by using an old version of the record and the transaction ids and the corresponding parameters stored in the generalized version file. Thus, the usability of the *pivot transaction models* has improved compared to the one described by Mehrotra *et al*.[7]

The Version File and Commutative Updatings Countermeasure

The version file countermeasure and the commutative updatings countermeasure may be combined to improve the design of commutative subtransactions. A subtransaction may have some updatings using the first method and some updatings using the second method. Even within the same file, some fields may be updated using the first method, while other fields may be updated using the second method.

In many large companies such as banks it is required by legislation to file version data. In such companies it would be appropriate to implement versions in the way described above.

Example 3.2

A *hot backup center* is a standby system that is always ready to take over the production of another computer center. A system with a hot backup center may be viewed as a distributed database with two nodes, where each node is ready to take over the production of the other node. The largest bank in Denmark, Den Danske Bank, has implemented a hot backup center using some of the methods described in this paper.

Den Danske Bank has designed all its transactions to be commutative by using a generalized version file. For each transaction they store the valid time of the transaction together with either the after-images or the transaction type and parameters. For example, if a customer address is changed, they store the after-image of the changed record fields, and if an amount is entered on an account, they store the credit or debit amount without updating the old balance of the account. The balances of the accounts are only updated periodically, but may be calculated on request. However, this will change in the future, because most updatings of a balance are commutative, and therefore it is only necessary to recalculate the balance if a non-commutative transaction has updated it.

The bank has two computer centers located in two different cities. The two centers act as hot backup centers for each other. All query transactions are executed as local transactions at the nearest computer center.

All updating transactions are global transactions, where the pivot subtransaction is executed at the nearest computer center. A retriable subtransaction is updating the other computer center by using an UP as described in section ‘SELECT versus

INSERT Update Propagation'. Den Danske Bank only uses local concurrency control. Therefore, it is important that all transactions are designed as commutative transactions. Of course, the two centers are normally inconsistent, but they converge towards the same consistent state (*asymptotic consistency*), and within each center all transactions may be designed by using the local consistency and isolation property.

Due to the temporary inconsistency between the two computer centers there is a small risk of fraud or accounts overdrawn, if withdrawals are accepted at the same time in both computer centers, and if the two withdrawals combined exceed the amount available on the account. The bank has accepted this risk, because the retriable updating subtransactions normally succeed after a short time. Anyway, the risk is small, because you cannot withdraw a large sum of money without warning the branch office at least one day in advance.

If one of the computer centers is destroyed physically, the bank may lose a few transactions updated in the destroyed computer center and not yet propagated to the other center. (There is neither one primary copy nor two synchronous copies, and therefore the system is neither 1-safe nor 2-safe.[1]) However, this is not a problem in a bank, as all transactions may be logged in the branch office server for auditing purposes. Anyway, such an accident is very rare, and only few transactions may be lost.

To have a backup facility as described above gives each center an extra processing time of 8 per cent. The following countermeasures illustrate how it is possible for the bank to develop the system and increase the isolation property where it is needed.

The Reread Countermeasure

The reread countermeasure may primarily be used to prevent the lost update anomaly. Transactions that use this countermeasure read a record twice using short duration locks for each read. If a second transaction has changed the record between the two readings, the transaction aborts itself after the second read.

As described earlier, it is often not acceptable to lock records across a dialog with the user, because one does not know how long it will take the user to make the input data. In this situation the reread countermeasure may be used to protect against the lost update anomaly in the following way. First, the application reads all the records needed using short duration shared locks, which are released after the readings. Next, the user enters data for the updating process assuming that no other transaction will change the records. After the data is entered by the user, the application rereads the records, this time using short duration exclusive locks, and the updatings will only be executed and committed if none of the records have been changed since they were read the first time. (An update time-stamp in all database records could also be used to detect whether data has been changed since it was read the first time.)

Example 3.3

When changing the address of a customer one may first read the old address using short duration locks of the data. Then the corrected address is read from the user's input data, and an updating subtransaction may be 'initiated'. First the updating subtransaction rereads the address data using a local exclusive lock. If the address data has been changed since it was first read, the updating subtransaction is aborted.

(After the abortion the root transaction may be restarted.) If the address data has not been changed, the old address data is corrected and the updating subtransaction is committed locally.

The reread countermeasure can be very useful in situations where primary and secondary copies are used. If data is replicated in this way, the nearest copy of the data may be used the first time the data is read. When the data is reread for updating, a primary copy must be used.

Example 3.4
In an airline company it is convenient that flight departure records contain a field with the number of vacant seats. This file may be replicated to all the company's heavy airline junctions, where it is used for reading purposes. Suppose that the primary copy is stored at the location closest to the airport of departure.

When the field 'vacant number of seats' has to be updated, the primary copy is read using an exclusive lock. If there are enough vacant seats, the primary copy is updated, and UPs updating the replicated copies are 'initiated'.

The reread countermeasure may also sometimes reduce the non-repeatable read anomaly and the dirty read anomaly, because by rereading all the data of a transaction it is possible to abort the transaction, if any of the values have been changed.

The problem by using the reread countermeasure in the Countermeasure Transaction Model is that after rereading a record, the record may be updated by an UP that has been 'initiated' before the rereading transaction was committed globally. Therefore, the reread countermeasure can only reduce the number of globally inconsistent reads.

The Pessimistic View Countermeasure

It is sometimes possible to reduce or eliminate the dirty read anomaly and/or the non-repeatable read anomaly by giving the users a pessimistic view of the situation. The purpose is to eliminate the risk involved in using data where long duration locks should have been used.

A pessimistic view countermeasure may be implemented by using:

(a) Compensatable subtransactions for updatings which limit the options of the users.
(b) Retriable subtransactions for updatings which increase the options of the users.

Example 3.5
When updating stocks, accounts, vacant passenger capacity, etc. it is possible to reduce the risk of reading stock values that are not available ('dirty' or 'non-repeatable' data.) These pessimistic stock values will automatically be obtained if the transactions updating the stocks are designed in such a way that compensatable subtransactions (or the pivot transaction) are used to reduce the stocks, and retriable subtransactions (or the pivot transaction) are used to increase the stocks.

Countermeasures by Value

The selection of transaction-managed countermeasures may depend on record values read by the transaction, and/or the countermeasures may depend on the parameter values of the transaction. In the following, this is called *countermeasures by value*. Countermeasures by value may also be used to mix transactions with global ACID properties managed by a DDBMS with transactions using the semantic ACID properties described in this paper. In other words, it is possible to restrict the use of global ACID properties to transactions where they are absolutely necessary.

Example 3.6
Suppose a bank has accounts replicated on different local databases.

If a 'large' amount of money is withdrawn or deposited, a transaction with global ACID properties is executed managed by a DDBMS.

If a 'small' amount of money is withdrawn or deposited, the nearest copy of the account is updated first, and then the other copy of the account is updated using a retriable subtransaction as described in Den Danske Bank example (Example 3.2).

Suppose there is a group of customers belonging to a risk category, who send transactions comprising large sums between their accounts. If an account manager wants to analyse the total account value of such a group of customers, this may be calculated by using a local transaction, because 'large' amounts of money are executed concurrently and in isolation on both sites, and 'small' amounts of money do not matter and will have no real influence on the result of the evaluation of the group. In other words, by using distributed ACID properties for a few transactions where it is absolutely necessary, the risk of all the isolation anomalies has been reduced to an acceptable level. Using a DDBMS cannot solve the dirty read anomaly and the non-repeatable read anomaly, if the DDBMS-transactions are mixed with transactions using retriable subtransactions. But in the example above the anomalies are reduced, because large sums of money cannot be 'dirty' or 'non-repeatable'.

Countermeasures by value are inspired by the *flexible transaction models*.[12,13,8]

The Semantic Lock Countermeasure

This countermeasure may be used to implement locking concurrency control, but it may also be used as a countermeasure that only reduces the isolation anomalies. Records which are going to be protected by this countermeasure always have to be updated by both a compensatable subtransaction and a retriable subtransaction. First, a compensatable subtransaction updates the record and marks the data (this may function as a lock or a warning depending on the semantics of the application.) When the global transaction is committed, a retriable subtransaction unmarks the data. In normal locking theory records can only be marked with *shared lock* or *exclusive lock*.[1] By using transaction managed marks one may mark the record with the type and parameters of the marking transaction. This will give the concurrent transactions more information, which may be used to decide how they will respect the marking. By combining this countermeasure with the countermeasure by value it is possible to restrict the marking to transactions where it is absolutely necessary.

Transactions that respect semantic locks must also select a solution to the deadlock problem. In the following example it is assumed that the time-out solution is used.

Example 3.7

It is possible to manage the customers in the risk category described in the previous example 3.6 by combining the semantic lock countermeasure and the countermeasure by value. If a 'large' sum is withdrawn or deposited, the account record stored at the remote site is updated by using a compensatable subtransaction that also marks the remote account with the transaction type and the amount of the subtransaction. After this, a pivot subtransaction updates the account stored at the site of the root transaction. If the pivot subtransaction is committed, a retriable subtransaction un-marks the remote account record. If the pivot subtransaction is aborted, a compensating subtransaction compensates and un-marks the remote account record.

If an account manager wants to analyse the total value of a group of customers, the total value of these accounts may be calculated using a local transaction, because 'small' amounts of money do not matter, and 'large' sums are marked if they are not committed. By adding the marked amounts of money we are able to abort the transaction if the uncommitted sums are too large, i.e. the 'dirty' data and the 'non-repeatable' data are under control.

Notice that the availability of the data is greater in this example than in Example 3.6, which deals with the same problem.

In the following example, the semantic lock countermeasure is used only to accumulate information to reduce the dirty read anomaly and the non-repeatable read anomaly. In the example the semantic lock countermeasure does not lock data, and therefore the deadlock situation cannot occur.

Example 3.8

When updating the accounts in Example 3.7, we need not use the semantic lock countermeasure to protect against the lost update anomaly, because the updating subtransactions are commutative. Therefore, instead of marking and demarking an account, we use a counter field to indicate that uncommitted transactions are updating an account record. That is, concurrent transactions may update the same account if the counter field is increased with 'the amount of money' of the 'marking' compensatable subtransaction and decreased with 'the amount of money' of the 'demarking' retriable subtransaction.

If an account manager wants to analyse the total value of a group of customers, the total value of these accounts may be calculated by using a local transaction, as 'small' amounts of money do not matter and 'large' uncommitted sums of money are accumulated in the counter fields. By adding the values of the counter fields for the analysed group of customers we are able to abort the transaction if the uncommitted sums of money are too large, i.e. the 'dirty' data and the 'non-repeatable' data are under control.

Notice that the availability of the data is greater in this example than in Examples 3.6 and 3.7, which deal with the same problem.

Traditional, distributed *two phase commit* and *two phase locking*[1] may be viewed as a special case of The Countermeasure Transaction Model, because two phase locking may be implemented by the semantic lock countermeasure using shared and exclusive locks, and if the pivot subtransaction cannot be committed, the roll back process of two phase commit may be implemented using the afterimages stored by the version file countermeasure.

The three main approaches to locking isolation of replicated data are *write locks all*, *majority locking* and *primary copy locking*.[14] All these methods may also be implemented by the semantic lock countermeasure using shared and exclusive locks. When the semantic lock countermeasure is used as traditional locks, the locks ought to be integrated with countermeasures by value to increase the availability.

The Pivot Lock Countermeasure

By using long duration locks in the pivot subtransaction, it is possible to solve all the isolation anomalies within the pivot subtransaction. Notice that often this countermeasure is not used, since it is 'bad programming' to lock data across a dialog with the user.

Traditional centralized on-line data processing may be viewed as the special case of the Countermeasure Transaction Model, where the pivot subtransaction and the root transaction are integrated in a single transaction, where the pivot lock countermeasure is used for concurrency control, and where the compensatable and retriable subtransactions are missing.

Normal client/server architecture using a central database may also be viewed as the special case of the Countermeasure Transaction Model, where the root is the client and the pivot subtransaction is the server, and where the compensatable and retriable subtransactions are missing.

GLOBAL ATOMICITY USING REMOTE PROCEDURE CALLS AND UPDATE PROPAGATIONS

It is specific for the global atomicity algorithms described in this section that only RPCs and UPs are used to access persistent data. Therefore, these algorithms must be integrated with countermeasures to protect against isolation anomalies.

We assume that a global transaction consists of at least a root transaction and some updating subtransactions (some reading subtransactions may also be used.)

Notice that, in The Countermeasure Transaction Model, the communication with the user (which only takes place from the root transaction) plays an important role, since we normally do not lock data across a dialog with the user (in order to prevent locking the data for a long time.) Therefore, the reading subtransactions are normally separated from the corresponding updating subtransactions.

First, we will describe the *enforcement method*, where the updating subtransactions consist of a single pivot subtransaction that initiates one or more retriable subtransactions. After this, we will describe the *compensation method*, where the updating subtransactions consist of one or more compensatable subtransactions followed a single pivot subtransaction. The enforcement method and the compensation method can be combined in order to implement atomicity for global transactions consisting of several compensatable subtransactions, a single pivot subtransaction, and several retriable subtransactions.

A more detailed description of the methods is presented in the technical report by Frank and Zahle.[15]

The Enforcement Method

This is the most important method for obtaining global atomicity (since it is much simpler for the programmer than the compensation method), and therefore transactions are designed to use the enforcement method if possible.

Using the enforcement method, the global transaction has the following program components:

(a) *In the root transaction*
The root transaction is activated by the user. First some non-updating processing may be executed. Then the RPC for the pivot subtransaction is made.
When control comes back the root transaction may continue with non-updating processing, e.g. telling the user that the global transaction is committed.

(b) *In the pivot subtransaction*
The updatings of the pivot subtransaction are executed and the UPs of the retriable subtransactions are 'initiated'. Finally, all the updatings and initiations at the pivot location are committed.

(c) *In the retriable transactions*
The retriable subtransactions initiated by the pivot subtransaction execute asynchronously in the servers.

The following example illustrates a global updating transaction using the enforcement method:

Example 4.1
As described in Example 2.3, a money transfer between two accounts stored at different locations may be designed as a pivot withdrawal subtransaction and a retriable deposit subtransaction.

(a) *In the root transaction*
The root transaction executes RPCs reading all the data that the user needs for undertaking the money transfer. (If the user has to evaluate the information in the withdrawal account, it is advisable to read the withdrawal account by using short duration locks to prevent locking the account across a dialog with the user.) Data from the deposit account may also be read in this step. (Notice that all the data read in this step may be changed before the updatings are executed.)
If the root transaction and the user can accept the credit card information, etc., the root transaction executes the pivot RPC.
The root transaction receives the answer 'committed' or 'aborted' from the pivot subtransaction and conveys it to the user.

(b) *In the pivot subtransaction*
The pivot subtransaction executes the updating of the withdrawal account. (If the withdrawal account was read in the first step, the reread countermeasure may be used to prevent the lost update anomaly.) After this, the UP of the deposit account is initiated. Provided that the updatings and initiation are successful, the pivot/global transaction is committed.

(c) *In the retriable subtransactions*
Sooner or later, the retriable subtransaction will be executed, i.e. the deposit is updated and committed.

The Compensation Method

In the compensation method, the programmer must make both a number of remote procedures as well as a number of compensating procedures. Using the compensation method the global transaction has the following program components:

(a) *In the root transaction*
The root transaction is activated by the user. In this step some non-updating processing may be executed. The compensatable subtransactions and the pivot subtransaction are called in turn by using RPCs.
When control comes back, the root transaction receives the answer committed or aborted from the subtransactions.
If one of the updating subtransactions aborts (gives a negative answer), the previously executed subtransactions must be compensated. This is done by initiating the relevant compensating procedures. (Notice that the compensating procedures must be enforceable as described in section 'The Enforcement Method' since otherwise the atomicity of the global transaction is endangered.)

(b) *In the subtransactions*
Stored procedures in the servers execute the compensatable subtransactions using short duration locks.

Integration of the Enforcement and Compensation Method

The enforcement method and the compensation method may be integrated to implement atomicity for global transactions consisting of both single pivot subtransaction and one or more retriable and compensatable subtransactions.

IMPLEMENTATION OF UPDATE PROPAGATIONS

Some DBMS products have a UP tool which can only be used to propagate updatings between the venders' own DBMS products. Therefore, it is important to know how to implement the UP tool. We have only found partial descriptions in the scientific literature of how to implement UPs. (*Recoverable request*[16] and *durable request*[1] have similarities with UPs, but UPs do not open a session between the client and the server, because the server is not returning an answer to the parent/user.)

Therefore we will describe in detail how the UP tool may be implemented, and compare the different implementation methods. In the implementations only short duration locks are used, i.e. the implementations must be integrated with countermeasures.

In general, the problem is that a parent transaction at location A wants to transfer the subtransaction id and parameters to a stored program at location B. The subtransaction id, the parameters and the id of the parent transaction are stored in a *transaction record* as described in the section on 'The countermeasure transaction model'. The main steps in transferring the transaction record and executing the corresponding stored program are as follows:

1. The parent transaction writes the transaction record in a DBMS file at location A to ensure that the transaction record has the same atomicity and durability as the other local DBMS updates of the parent transaction. Subsequently, all

location A updatings are committed. (The subtransaction related to the UP will not be executed if the parent transaction fails before commitment of the location A updatings.) When the transaction records are committed, there are two main methods for transferring the transaction records in the transaction file at location A.

2a. At location B one may periodically select transaction records from the location A transaction file. After a transaction record has arrived at location B, the corresponding subtransaction may be executed. In the following this is called a *SELECT Update Propagation*.

2b. At location A one may insert the transaction record from the location A transaction file in a location B transaction file. After the transaction record has been stored in the location B transaction file, the corresponding subtransaction may be executed. In the following, this is called an *INSERT Update Propagation.*

In the following subsections the SELECT and INSERT Update Propagation methods will be described in more detail, and finally the methods will be compared. (The SELECT Update Propagation method has been published by Frank,[5,6] but the following description is more detailed and includes a comparison with other UP methods.)

SELECT Update Propagation

The transaction records stored at location A have a sequence number as a primary key. If the parent transaction at location A fails, the transaction record at the location A transaction file will be backed out, and the sequence number may be reused. This is very straightforward, since each site has the local ACID properties.

In the location B database there is a counter record containing the sequence number of the last transaction record successfully transferred and executed at location B. At location B there is a general transfer program, which is restarted periodically. When the program is restarted, it will begin to read the counter record containing the last successfully executed subtransaction. Then, the program will search for the next transaction record in the transaction file at location A. (A RPC may be used, since it will be restarted periodically as described above, in case it should fail.) When the next transaction record arrives at location B, the counter record will be updated and the corresponding subtransaction will be executed. Following this operation, the location B updatings of the subtransaction and the update of the counter record will be committed.

If the general program or the subtransaction fails before this commitment, the counter record is backed out together with the updatings of the subtransaction. Because the counter record is backed out, the same transaction record will be transferred to location B the next time the general transfer program is started. In other words, if a transaction record in the location A transaction file is committed, the corresponding subtransaction will sooner or later be executed at location B.

The sequence number of the next transaction record may be stored in a ‘sequence number record’ in the location A database. This sequence number record may constitute a bottleneck, as it will be updated for each transaction record that is sent to location B. The problem may be reduced by locking the sequence number record as the last updating of the parent transaction. (The problem may be further reduced by using a time-stamp instead of a sequence number, as the time-stamp does not have to be updated by the DBMS. If a time-stamp is used, it is still necessary to use an exclusive lock while getting the time-stamp, because the transaction file must be updated in an ascending sequence of the primary key, which in this case is the time-stamp.)

Insert Update Propagation

This method does not use a sequence number or time-stamp to control the transfer and execution of subtransactions. Instead, location B has a transaction file to ensure that a subtransaction will only be executed once at location B. The INSERT Update Propagation contains the following steps:

1. The location A updatings, including the transaction record, are committed (as described in the introduction to the section on ‘Implementation of update propagations.)
2. The parent transaction at location A initiates step 3, which sends the transaction record to location B. (The transfer will only be delayed if step 2 fails, as step 3 will be restarted automatically.)
3. This step is a general transfer program, which is restarted periodically with ‘short’ time intervals. This step may also be started from step 2. The program will start sending the transaction records in the location A transaction file to location B. For each transaction record in the location A transaction file the following steps are executed:
4. When the transaction record arrives at location B, a program is checking whether the transaction record already has been inserted into the location B transaction file. If the transaction record already has been stored, the location B updatings have already been executed and committed locally. If the transaction record has not been inserted, the insert and the location B updatings will be executed and committed locally. After the local commitment of the location B updatings, a RPC will be executed and start the following stored procedure at location A.
5. At location A the transaction record corresponding to the location B subtransaction will be deleted from the location A transaction file, i.e. the submission of the transaction record in step 3 cannot be repeated any more. When the delete operation has been committed, the INSERT Update Propagation of a single transaction record is finished.

There exists a version of the INSERT Update Propagation method which is often used in practice, but unfortunately it is not always safe in a heterogeneous environment. The method uses the IBM data transmission protocol LU6.2, which may guarantee the transfer and execution of transactions.[17] If the DBMS is an IBM product, the commit of the DBMS and the LU6.2 transaction transfer can be integrated. If the DBMS is not an IBM product, the local commit of the DBMS updatings in system A will not be the same as the commit of the LU6.2 transfer order. This may be a problem if the application fails after the local DBMS commit, but before the LU6.2 transfer commit. However, in practice, it need not be a problem if the transfer is supervised by a user as illustrated in the next example.

Example 5.1
From the cash desk in a branch office of a bank the user may send money transactions to the central account file. For audit reasons the money transaction must be logged in the local database of the branch office before it is sent to the central database. After the logging the transaction will be sent to the central database by LU6.2. If the user receives an answer from the central database, there are no problems. If the user does not receive an answer, the user must start a program that analyses the situation and tells the user whether the transaction has been sent or not. First, the program must examine the local database. If the money transfer is not in the local database, the user must start a new money transfer. If the money transfer has been recorded in the local database, the program must examine the LU6.2 logfile. If the money transfer record is not in the LU6.2 logfile, the program must start the LU6.2 transfer of the money transaction.

The method described above is used by Unibank (the second largest bank in Denmark) in their on-line accounting system.

SELECT versus INSERT Update Propagation

The SELECT and INSERT Update Propagations have different properties. Therefore, it is often necessary to implement both methods in a distributed system. The most important differences are listed below.

(a) SELECT Update Propagations are batch oriented, as transaction records are only transferred periodically. INSERT Update Propagations are not batch oriented, as a transaction record normally is sent immediately (periodic transfers are only used in cases of error.)
(b) SELECT Update Propagations have a smaller execution cost than INSERT Update Propagations, because a SELECT Update Propagation does not involve deleting the transaction record in the location A transaction file.
(c) The programming cost of SELECT Update Propagations is smaller than the programming cost of INSERT Update Propagations, because the programming of a SELECT Update Propagation does not involve deleting the transaction record in the location A transaction file.
(d) The SELECT Update Propagation of transaction records is single-threaded. That is, the transaction records are transferred, and the corresponding subtransactions are executed in sequence order. The SELECT Update Propagation may be designed to be multithreaded, if there is a counter record for each thread. The INSERT Update Propagation may be designed as multithreaded if step 2 in the transfer algorithm always initiates transfer of the newest transaction record, while the periodic restart of step 3 always transfer the oldest transaction record in the location A transaction file. If the transaction transfer is single-threaded it may be a potential bottleneck. On the other hand, if the transaction transfer is multithreaded, the subtransactions related to the transaction records are not always executed in sequential order, and this may increase the isolation anomalies.
(e) The updatings of the SELECT UP have two potential bottlenecks, since ‘the counter record’ and ‘the sequence number record’ must be updated for all transaction transfers.

CONCLUSIONS

Primarily, the objective of this paper is to present practical knowledge about design of semantic atomicity and isolation in high performance multidatabases.

To solve practical problems we have taken many features and ideas from other transaction models and used them in The Countermeasure Transaction Model. Therefore, many transaction models will appear as special cases of The Countermeasure Transaction Model.

To our knowledge the following concepts/methods/analyses have not been described in the scientific literature:

(a) The atomicity implementation using *only* the Remote Procedure Calls and Update Propagations, i.e. we use client/server architecture and not a global transaction manager.
(b) Transaction managed countermeasures constitute a new method to implement semantic isolation between transactions.
(c) The concepts and methods are applied in the hot backup center used by Den Danske Bank. The bank is very satisfied with the system compared with the 1-safe system they had before. Especially, availability of data and utilization of the two computers have been improved.

SUGGESTIONS FOR FURTHER RESEARCH

There is a need for further research in designing UPs, as there are many versions of the main methods described in this paper. (A number of methods may be found in the literature on atomicity in DDBMS and MDBS products.) From the point of view of performance, a comparison of methods would be interesting. The most successful implementations could be developed as standard software for any DBMS product.

We look at the design of UPs using Internet as a communication network with special interest, because it is a cheap and practical way to design global semantic atomicity. This involves designing standard database interfaces that can receive and send Internet Remote Procedure Calls and Update Propagations.

acknowledgements

Thanks are due to Professor S. Lauesen who gave us feedback on the paper content, Professor N. Jones who corrected our English, and the diligent referees who improved the paper structure and style.

REFERENCES

1. J. Gray and A. Reuter, *Transaction Processing*, Morgan Kaufman, 1993.
2. Y. Breibart, H. Garcia-Molina and A. Silberschatz, ‘Overview of multidatabase transaction management’, *VLDB Journal*, **2**, 181–239 (1992).
3. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil and P. O’Neil, ‘A critique of ANSI SQL isolation levels’, *Proc. ACM SIGMOD Conf.*, 1995, pp. 1–10.
4. H. Garcia-Molina and K. Salem, ‘Sagas’, *ACM SIGMOD Conf.*, 1987, pp. 249–259.
5. L. Frank, *Databaser*, Samfundslitteratur, Copenhagen, 1985.
6. L. Frank, *Database Theory and Practice*, Addison-Wesley, 1988.
7. S. Mehrotra, R. Rastogi, H. Korth and A. Silberschatz, ‘A transaction model for multidatabase systems’, *Proc. International Conference on Distributed Computing Systems*, 1992, pp. 56–63.

8. A. Zhang, M. Nodine, B. Bhargava and O. Bukhres, ‘Ensuring relaxed atomicity for flexible transactions in multidatabase systems’, *Proc. ACM SIGMOD Conf.*, 1994, pp. 67–78.
9. H. Garcia-Molina, ‘Using semantic knowledge for transaction processing in a distributed database’, *ACM TODS*, **8**(2), 186–231 (1983).
10. ANSI X3.135–1992, *American National Standard for Information Systems – Database Language-SQL*, November 1992.
11. G. Weikum and H.-J. Schek, ‘Concepts and applications of multilevel transactions and open nested transactions’, in A. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992, pp. 515–553.
12. A. Elmagarmid, Y. Leu, W. Litwin and M. Rusinkiewicz, ‘A multidatabase transaction model for InterBase’, *Proc. 16th VLDB Conf.*, 1990, pp. 507–581.
13. J. Veijalainen, F. Eliassen and B. Holtkamp, ‘The S-transaction model’, in: A. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992, pp. 467–513.
14. S. Ceri and G. Pelagatti, *Distributed Databases: Principles & Systems*, McGraw-Hill, 1985.
15. L. Frank and T. U. Zahle, ‘Semantic ACID properties in multidatabases using remote procedure calls and update propagations’, *Technical Report*, Institute of Informatics and Management Accounting, Copenhagen Business School, 1997.
16. P. Bernstein, M. Hsu and B. Mann, ‘Implementing recoverable requests using queues’, *ACM SIGMOD Record*, 112–122 (1990).
17. IBM, SNA Logical Unit, *Version 6: Reference: Peer Protocols*, IBM, 1990.
18. R. Cordon and H. Garcia-Molina, ‘The performance of a concurrency control mechanism that exploits semantic knowledge’, *Proc. IEEE 5th International Conference on Distributed Computing Systems*, 1985, pp. 350–358.
19. U. Dayal *et al.*, ‘A transaction model for long-running activities’, *Proc. 17th VLDB Conf.*, 1991, pp. 113–122.
20. A. Deacon, H.-J. Schek and G. Weikum, ‘Semantics-based multilevel transaction management in federated systems’, *Proc. 10th International Conference on Data Engineering*, 1994, pp. 452–461.
21. A. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992.
22. H. Garcia-Molina and B. Kogan, ‘Achieving high availability in distributed databases’, *IEEE Transactions on Software Engineering*, **14**, 886–896 (1988).
23. J. Gray, ‘Notes on databases operating systems’, *Operating Systems: An Advanced Course*, *Lecture Notes in Computer Science* 60, Springer-Verlag, 1978, pp. 393–481.
24. J. Gray, ‘An approach to decentralized computer systems’, *IEEE Transactions on Software Engineering*, **12**, 684–692 (1986).
25. J. Gray and M. Anderton, ‘Distributed computer systems: Four case studies’, *Proceedings of the IEEE*, **75**, 719–726 (1987).
26. H. Korth, E. Levy and A. Silberschartz, ‘A formal approach to recovery by compensating transactions’, *Proc. 16th VLDB Conf.*, 1990, pp. 95–106.
27. W. Litwin, L. Mark and N. Roussopoulos, ‘Interoperability of multiple autonomous databases’, *ACM Computing Surveys*, **22**(3), 267–293 (1990).
28. N. Lynch, ‘Multilevel atomicity – a new correctness criterion for database concurrency control’, *ACM TODS*, **8**(4), 484–502 (1983).
29. J. Moss, *Nested Transactions: An approach to Reliable Distributed Computing*, MIT Press, 1985.
30. C. Polyzois and H. Garcia-Molina, ‘Evaluation of remote backup algorithms for transaction-processing systems’, *ACM TODS*, **19**(3), 423–449 (1994).
31. H. Wachter and A. Reuter, ‘The ConTraact model’, in A. Elmagarmid (ed.), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992, pp. 219–263.