Integrity Problems in Distributed Accounting Systems with Semantic ACID Properties

Lars Frank[1]
Department of Informatics, Copenhagen Business School, Howitzvej 60, DK-2000 Frederiksberg, Denmark

Keywords: Semantic ACID properties, multidatabases, fault tolerance, ERP, distributed accounting.

Abstract:
Many major companies have a physically distributed sales and/or production organization. In such an organization a distributed ERP (Enterprise Resource Planning) system may optimize performance and response time by storing data locally in the locations where they normally are used. If traditional ACID properties (Atomicity, Consistency, Isolation and Durability) are implemented in such a system, the availability of the system will be reduced, because often updating transactions can only be executed if they have access to both local and remote databases. This problem can be reduced by using only semantic ACID properties, i.e. from an application point of view, the system should function as if all the traditional ACID properties had been implemented. This paper illustrates how the integrity problems caused by using semantic ACID properties may be solved in distributed accounting systems. However, some of the techniques described in this paper may also be used to improve performance and availability in centralized accounting systems.

In this paper we describe how the largest bank in Denmark, Den Danske Bank, has distributed all its important database systems by using semantic ACID properties in order to achieve better performance and availability. We have also analyzed how one of the major ERP software companies can design a distributed version of their general management and accounting system by using the methods described in this paper. The Software Company has now started to implement a prototype of their general management and accounting system as described in broad outline in this paper.

[1] This work was supported in part by The Danish Social Science Research Council, Project No. 9701783.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35501-6_14

M. E. van Biene-Hershey et al. (eds.), *Integrity and Internal Control in Information Systems*

1. INTRODUCTION

In the transaction model described in this paper, the global atomicity property is implemented by using retriable, pivot and compensatable subtransactions in that order (Frank, 1999). The application programs may maintain the global consistency property for which purpose we describe tools. The global isolation property is implemented by using countermeasures (Frank and Zahle, 1998) against the missing isolation of the update transactions. The global durability property is implemented by using the durability property of the local DBMS systems (Breibart et al., 1992). Thus, many of the problems of implementing semantic ACID properties have already been solved. However, to our knowledge no technical literature has described the special problems of distributed accounting systems with semantic ACID properties. Therefore, we will focus on how to produce consistent balance sheets in accounting systems that use semantic ACID properties.

The paper is organized as follows: Section 2 will describe an extended transaction model that provides semantic ACID properties. In section 3 we will illustrate by examples the problems of making consistent balance sheets. Concluding remarks are presented in section 4.

Related Research: The transaction model described in section 2 is *The Countermeasure Transaction Model* from Frank and Zahle (1998) and Frank (1999). This model owes many of its properties to e.g. Garcia-Molina and Salem, 1987; Weikum and Schek (1992); Mehrotra, 1992; Zhang, 1994.

2. THE TRANSACTION MODEL

A *multidatabase* is a union of local autonomous databases. *Global transactions* access data located in more than one local database (Gray and Reuter, 1993). In recent years, many transaction models have been designed in order to integrate local databases without using a distributed DBMS. The countermeasure transaction model, Frank and Zahle, 1998, has, among other things, selected and integrated properties from these transaction models in order to reduce the problems of the missing ACID properties in a distributed database not managed by a distributed DBMS. In the countermeasure transaction model a global transaction consists of a *root transaction* (client transaction) and several single site *subtransactions* (server transactions). The subtransactions can be nested transactions; i.e. a subtransaction may be a *parent transaction* for other subtransactions.

All communication with the user is managed from the root transaction, and all data is accessed through subtransactions. A subtransaction is either an execution of a *stored procedure* that automatically returns control to the parent transaction or an execution of a *stored program* that does not return control to the parent transaction.

Each remote subtransaction is accessed through either an RPC or a UP tool.

Remote Procedure Call (RPC).
From a programmer's point of view, an RPC (Birrell and Nelson, 1984) functions like a normal, synchronous procedure call, except that the procedure call and the procedure itself are stored at different sites. RPCs have the following properties, which are important from a performance and an atomicity point of view:

- If a parent transaction executes several RPCs, the corresponding stored procedures are executed one at a time.
- A stored procedure managed from an RPC has only local ACID properties.
- The stored procedure automatically returns control to the parent transaction.

Update Propagation (UP).
Update propagation is here used in the general sense of asynchronous propagation of any update (not just replicas). *Recoverable request* (Bernstein, Hsu and Mann, 1990), and *durable request* (Gray and Reuter, 1993) have a certain similarity with UPs, but UPs do not open a session between the client and the server because the server does not return an answer to the parent/user. The UP tool works in the following way.
The parent transaction makes the UP "call" by storing a so-called *transaction record*[1] in persistent storage at the parent location. 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 has been *initiated.* After the initiation of the UP the transaction record will be read and sent by the UP tool (by means of a communication protocol) to the location of the corresponding subtransaction. If the subtransaction fails, the transaction record will be resubmitted until the subtransaction is committed. UPs may be implemented by using either *push* or *pull* technology. How different types of UPs have to be implemented is described in Frank and Zahle (1998). Some non-heterogeneous versions of these tools have been implemented in e.g. DB2 and Oracle DBMS software. UPs have the following properties, which are important from a performance and an atomicity point of view:

- If a parent transaction initiates several UPs, the corresponding, stored programs may be executed in parallel.
- A stored program initiated from a UP has atomicity together with the parent transaction, i.e. either both are executed or none are.
- The stored program does not automatically return control to the parent transaction.

In the rest of this section we will give a broad outline of how semantic ACID properties are implemented in The Countermeasure Transaction Model. However,

[1] The parent transaction id, the id of the subtransaction and the parameters of the subtransaction are stored in the transaction record.

the countermeasures used in making consistent balance sheets will be described in more detail.

2.1. The Atomicity Property

An update transaction has the *atomicity property* and is called *atomic* if either all or none of its updates are executed. In the countermeasure transaction model the global transaction is partitioned into the following types of subtransactions that normally are executed at different locations:

1. The *pivot* subtransaction that manages the atomicity of the global transaction, i.e. the global transaction is committed when the pivot subtransaction is committed locally. If the pivot subtransaction aborts, all the updates of the other subtransactions have to be compensated or not executed.
2. The *compensatable* subtransactions that all may be compensated. Compensatable subtransactions must always be executed before the pivot subtransaction is executed in order to ensure that they can be compensated if the pivot subtransaction cannot be committed. Compensation is achieved by executing a *compensating* subtransaction.
3. The *retriable* subtransactions that are designed in such a way that the execution is guaranteed to commit locally (sooner or later) if the pivot subtransaction is committed. A UP tool is used to resubmit automatically the request for execution until the subtransaction has been committed locally, i.e. the UP tool is used to force the retriable subtransaction to be executed.

The global atomicity property of all 'single pivot transaction models' is implemented by executing compensatable, pivot and retriable subtransactions in that order. RPCs can be used to call/start the compensatable subtransactions and/or a pivot subtransaction, because the execution of these subtransactions is 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 decision on the global transaction, all the remaining updates are mandatory. Therefore, UPs are always used to execute the retriable subtransactions, which are always executed after the global commitment. If the pivot fails or cannot be executed, the execution of all the compensating subtransactions is mandatory. Therefore, UPs are always used to execute the retriable compensating subtransactions.

Example 2.1 – Transfer of money between accounts.
Let us suppose that an amount of money is to be moved from an account at one location to an account at another location. In such a case, the global transaction may be designed as a root transaction that calls a compensatable withdrawal subtrans-action 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

user's PC may call a pivot subtransaction executed at the bank of the user, which has a 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.

2.2. The Consistency Property

A database is *consistent* if the data in the database obeys the consistency rules of the database. If the database is consistent both when a transaction starts and when it has been completed and committed, the execution has the consistency property. Transaction *Consistency rules* may be implemented as a control program that rejects the commitment of transactions, which do not obey the consistency rules.

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 e.g. referential integrity between sites must be managed by the global transactions themselves.

2.3. The Isolation Property

A transaction is executed in *isolation* if the updatings of the transaction only are seen by other transactions after the updatings of the transaction have been committed. If the atomicity property is implemented, but there is no global concurrency control, the following isolation anomalies may occur: (Gray and Reuter, 1993 and Berenson et al., 1995)

- *The lost update anomaly* 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.
- *The dirty read anomaly* 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 our 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".

- *The non-repeatable read anomaly* or *fuzzy read* 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 our transaction model this may happen when the first transaction reads a record that is updated by a second transaction, which commits the record locally before the first transaction commits globally.

The rest of this section describes countermeasures that are used in the accounting examples of section 3. We will first describe countermeasures against the lost update anomaly, because it is the most important anomaly to guard against.

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 other subtransactions that only have commutative updatings. This is a very important countermeasure, because retriable subtransactions have to be commutative in order to prevent the lost update anomaly.

Example 2.2
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 program identifications and parameters stored in the generalized version file.

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 others may be updated using the second method.

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:

- Compensatable subtransactions for updatings which limit the options of the users.
- Retriable subtransactions for updatings which increase the options of the users.

Example 2.3
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.

2.4. The Durability Property

The execution of a transaction has the durability property, if the updates of a transaction cannot be lost after the transaction has been committed. The updates of transactions are said to be *durable* if they are stored in stable storage and secured by a log recovery system. In case a global transaction has the atomicity property, the global durability property will automatically be implemented, as it is ensured by the log-system of the local DBMS systems (Breibart et al., 1992).

3. INTEGRITY PROBLEMS IN PRODUCING CONSISTENT BALANCE SHEETS

In this section we will describe two new countermeasures which may be used to make consistent local balance sheets that can be integrated into an enterprise wide consistent balance sheet. First, we will describe "*the end of day transaction countermeasure*" which may synchronize the items accumulated in different locations to enable us to make consistent enterprise wide accumulations. This countermeasure prevents the non-repeatable read anomaly in balance sheets, because

all the retriable subtransactions of the day are forced to be executed and committed locally, before the local balance sheets are produced. Next, we will describe "*the semantic lock countermeasure*" which may be used to separate "dirty" data from committed data. Besides these new countermeasures we will also use the countermeasures described in section 2.

3.1. The End of Day Transaction Countermeasure

In a distributed system with retriable subtransactions it is not possible to make a local or enterprise wide consistent balance sheet at the end of the day, because nobody knows when delayed retriable subtransactions are going to be committed locally. Therefore, at the end of the day each local system should send a retriable "end of day transaction" to all the other locations. When a local database has received the "end of day transactions" from all the other locations, it is possible to make the local balance sheet. In the following example, only two locations exchange "end of day transactions". However, the second example in section 3.2 is more complex.

Example 3.1
A *hot backup center* (Frank and Zahle, 1998) 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 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 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, 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. 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 %.

In this example "the end of day transaction" countermeasure is important because the bank makes a daily balance sheet.

3.2. The Semantic Lock Countermeasure

By using this countermeasure we mark "dirty" or uncommitted data in order to separate these data from committed data, i.e. compensatable updatings should be marked as "uncommitted", because the data is "dirty" and may be compensated. Special record fields should be used to accumulate the uncommitted stocks and amounts of money in order to separate these accumulations from the accumulations of committed items. This technique may also be useful in central databases, because by using compensatable updatings it is possible to lock only one record at a time, and, therefore, deadlock cannot occur. Especially in relation to *long-lived* transactions (Gray and Reuter, 1993) this technique may be useful, because it is not acceptable to lock records for a long period of time. Later, when the pivot subtransaction has been committed, retriable subtransactions should un-mark the compensatable updatings in order to make them valid as normal transactions in the balance sheet.

The following example outlines how global invoicing transactions can be designed by using our transaction model. In the exmple, both "the semantic lock countermeasure" and "the end of day transaction countermeasure" are necessary in order to make a consistent local balance sheet that can be integrated into an enterprise wide balance sheet.

Example 3.2
When a salesman wants to make a new order, the salesman first accesses or creates a local copy of the customer record. Next, a compensatable subtransaction in the location of the salesman creates an order record with relationship to the customer record. Now, the salesman can make order-lines. For each new order-line the root transaction receives, it starts a compensatable subtransaction that creates an order-line. For each order-line a compensatable subtransaction updates the stock level of the product ordered in the order-line. If the local stock lot cannot fulfill the quantity ordered in the order-line, another stock lot can be accessed by using another compensatable subtransaction. If an order-line cannot be fulfilled, a compensatable subtransaction must update the field 'quantity-delivered' in the order-line. When the order form has been completed and the order-lines are confirmed by the servers of the stores, the pivot subtransaction is executed. The pivot subtransaction marks the order as "committed", updates the account of the customer and initiates retriable subtransactions that un-mark the compensatable updatings. That is, the compensatable updatings of the stores are committed globally. At the end of the day some of the retriable transactions may have been delayed. Therefore, "the end of day transaction countermeasure" is important too.

It is possible to integrate the application from the example above into an ERP (Enterprise Resource Planning) system such as SAP R/3, Baan, PeopleSoft, Concorde Axapta, Navision, etc. We have analyzed how one of these major software companies can design a distributed version of their financial management and accounting system by using our transaction model. The Software Company has started to implement a prototype of the enterprise-wide financial management and accounting system by using our transaction model.

4. CONCLUSION

Distributed updating applications must use semantic ACID properties in order to achieve high performance and availability. In this paper we have described countermeasures against the missing distributed isolation property, These countermeasures are used to illustrate how it is possible to make consistent local and enterprise wide balance sheet in distributed accounting systems using semantic ACID properties.

Frank and Zahle (1998) have described a large number of countermeasures. To our knowledge "the end of day transaction countermeasure" and "the semantic lock countermeasure" have not been published in the scientific literature. However, the "the end of day transaction countermeasure" has already been implemented in practice in a major bank, and "the semantic lock countermeasure" is under implementation in an ERP prototype system.

5. REFERENCES

1. Berenson, Hal and Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, Patrick O'Neil (1995). A Critique of ANSI SQL Isolation Levels, Proc ACM SIGMOD Conf., pp 1-10.
2. Bernstein, P., Hsu, M. and Mann, B. (1990), 'Implementing Recoverable Requests Using Queues', *ACM SIGMOD Record*, pp 112-122.
3. Birrell, A. and B. Nelson (1984), Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, Vol. 2, pp 39-59.
4. Breibart Y., H. Garcia-Molina and A. Silberschatz (1992), 'Overview of Multidatabase Transaction Management', *The VLDB Journal*, 2, pp 181-239.
5. Elmagarmid, A. (ed.) (1992), *Database Transaction Models for Advanced Applications*, Morgan Kaufmann.
6. Frank, L. and Torben Zahle (1998), Semantic ACID Properties in Multidatabases Using Remote Procedure Calls and Update Propagations, Software - Practice & Experience, Vol.28, pp77-98.
7. Frank, L. (1999), 'Atomicity Implementation in Multidatabases with High Performance and Availability', *Proc of the 2nd International Symposium on Cooperative Database Systems (CODAS'99)*, Springer-Verlag, pp 204-215.
8. Garcia-Molina, H. and K. Salem (1987), ' Sagas', *ACM SIGMOD Conf*, pp 249-259.
9. Gray, Jim and Andreas Reuter (1993). Transaction Processing, Morgan Kaufman.
10. Weikum, G. and H. J. Schek (1992), 'Concepts and Applications of Multilevel Transactions and Open Nested Transactions', in: Elmagarmid (1992), pp 515-553.
11. Wächter, H. and A. Reuter (1992), 'The ConTract Model', in: Elmagarmid (1992), pp 219-263.
12. Zhang, A., M. Nodine, B. Bhargava and O. Bukhres (1994), 'Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems', *Proc ACM SIGMOD Conf*, pp 67-78.