Atomicity Implementation in E-Commerce Systems*

Lars Frank and Uffe Kofod
Department of Informatics,Copenhagen Business School, Howitzvej 60, DK-2000 Frederiksberg, Denmark.
frank@CBS.DK

Abstract

Distributed databases with high performance and availability do not have the traditional ACID properties (Atomicity, Consistency, Isolation and Durability) because long duration locks will reduce the availability and the write performance. The problems of the missing ACID properties may be avoided by using approximated ACID properties, i.e. from an application point of view; the system should function as if all the traditional ACID properties had been implemented. The distributed approximated atomicity property manages the workflow of a transaction in such a way that either all the updates of the global transaction are executed (sooner or later) or all the updates of the global transaction are removed/compensated. In this paper, we will describe a flexible algorithm for implementing distributed approximated atomicity. Frank and Zahle [1] have described how to implement the other global approximated ACID properties.

We will illustrate our algorithm with E-commerce examples. If one of the partaking subsystems fails in a system for E-commerce, the approximated atomicity property will ensure that when an order is accepted, the payment and stock levels are managed automatically in the locations of the partaking banks and product stocks. Even logistics and/or production may be managed by using approximated atomicity.

We have cooperated with one of the major ERP (Enterprise Resource Planning) software companies in designing a distributed version of the ERP system with local autonomous databases in the different sales and stock locations.

Keywords: ACID properties, approximated atomicity, distributed systems, electronic commerce, ERP systems.

1. Introduction

In the transaction model described in this paper, the approximated atomicity property is implemented by using retriable, pivot and compensatable subtransactions. The global consistency property does not exist in our transaction model. However, the concept *asymptotic consistency* can be used to create a consistent database state for e.g. datawarehousing [10]. The approximated isolation property is implemented by using countermeasures [1] to the isolation anomalies that occur when transactions are executed without isolation. The global durability property is implemented by using the durability property of the local DBMS systems.

Our algorithm for approximated atomicity implementation is a transaction pattern with all the necessary types of database accesses, but without the application logic. This transaction pattern must be used by all distributed transactions. By using our transaction pattern, the development costs for new applications may be reduced.

By means of examples, we will illustrate how to implement the approximated atomicity property in E-commerce systems. There are many different workflow architectures for E-commerce systems [12] and our transaction pattern cannot cover them all. However, our transaction pattern can manage the distributed workflow needed in most E-commerce systems [13].

The paper is organized as follows:

Section 2 will describe an extended transaction model that provides approximated ACID properties. Section 3 describes a general transaction pattern (algorithm) for implementing the approximated atomicity property. In this section, we will also illustrate by examples how to implement the approximated atomicity property in practice. Concluding remarks are presented in section 4.

Related Research: The transaction model described in section 2 is *The Countermeasure Transaction Model* [1]. This model owes many of its properties to e.g. Garcia-Molina and Salem [2], Mehrotra [3], Weikum and Schek [4] and Zhang [5]. Frank and Zahle [1] describe in detail the countermeasures used against the isolation anomalies in the E-commerce examples of section 3.

An early version of the pattern described in this paper has been developed for atomicity implementation in CSCW systems [14].

2. The Transaction Model

A *multidatabase* is a union of local autonomous databases. *Global transactions* [7] access data located in more than one local database. In recent years, many transaction models have been designed in order to integrate local databases without using a distributed DBMS. The countermeasure transaction model [1], 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

* This project was in part supported from 'Udviklingscenter for e-business' – UCEB.

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.

All remote subtransactions are accessed through one of the following types of tools:

Remote Call (RC)

From a programmer's point of view, a RC functions like a remote procedure call or submission of a SQL query. RCs have the following properties, which are important from a performance and an atomicity point of view:

- If a parent transaction executes several RCs, the corresponding subtransactions are executed one at a time.
- A stored procedure or SQL submission has only local ACID properties.
- The stored procedure or SQL submission can automatically return control to the parent transaction.

Update Propagation (UP)

In this context, UP is used in the general sense of propagation of any update (not just replicas). The UP tool works in the following way:

The parent transaction makes the UP "call" by storing a so-called *transaction record* in persistent storage at the parent location. The following information must be stored in the transaction record:

> The parent transaction id, the id of the subtransaction, the id of the location where the subtransaction should be executed, the id of a stored procedure (or the SQL code) and the parameters of the subtransaction.

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 to the location where the subtransaction should be executed. UPs may be implemented by using either push or pull technology as described in Frank and Zahle [1]. 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 subtransactions may be executed in parallel.
- A subtransaction initiated from a UP has atomicity together with the parent transaction, i.e. either both are executed or none are.
- A subtransaction does not automatically return control to the parent transaction.

In the following, we will give a broad outline of how approximated ACID properties are implemented in the Countermeasure Transaction Model.

2.1 The Atomicity Property

An updating 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 are executed in different locations:

- 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 must be compensated or not be executed.
- The *compensatable* subtransactions that all may be compensated. Compensatable subtransactions must always be executed before the pivot subtransaction is executed to make it possible to compensate them if the pivot subtransaction cannot be committed. Compensation is achieved by executing a *compensating* subtransaction.
- 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 automatically resubmit 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 of the pivot transaction models is implemented by executing compensatable, pivot and retriable subtransactions in that order.

RCs 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 of 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 are mandatory. Therefore, UPs are always used to execute the retriable compensating subtransactions.

Example 2.1

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 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.

In our transaction model, subtransactions may be nested, i.e. a subtransaction may call another subtransaction, etc. In *The Open Nested Transaction Model* [4], the subtransactions of a compensatable subtransaction must be compensatable. (Otherwise, the parent transaction cannot be compensatable). This idea has been generalized and integrated into our transaction model in the following way:

- Subtransactions of a compensatable subtransaction must be compensatable. Please notice that sometimes the subtransactions of a compensatable subtransaction may also be designed as retriable, which simplifies the application program and reduces the response time.
- Subtransactions of a retriable subtransaction must also be retriable. (Otherwise, the parent transaction cannot be retriable).
- Subtransactions of a pivot subtransaction must either be compensatable or retriable. Compensatable subtransactions must be executed before the commit of the pivot subtransaction and retriable subtransactions must be executed after the commit of the pivot subtransaction.

Furthermore, a non-committed transaction should be subject to changes. This implies that a retriable subtransaction, which more or less compensates a compensatable subtransaction, may be executed before the commit of the pivot subtransaction.

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.

The definition above of the consistency property is not useful in multidatabases with approximated ACID properties because such a database is normally always inconsistent. However, a distributed database with approximated ACID properties should have *asymptotic consistency*, i.e. the database should converge towards a consistent state when all active transactions have been committed/compensated. Therefore, in distributed databases with approximated ACID properties, we want the following property:

If the database is asymptotically consistent when a transaction starts and also when the transaction is committed, the execution has the *approximated consistency property*.

Frank [10] has described how to make a consistent database state for datawarehousing on top of a distributed database with approximated ACID properties.

2.3 The Isolation Property

A transaction is executed in *isolation* if the updates of the transaction only are seen by other transactions after the updates 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 [7] [8]:

- *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 often executed in different subtransactions, as we do not recommend locking a record across a dialog with the user. Therefore, it is possible for a second transaction to update the record between the read and the update 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 *long duration locks* [7]. 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 phantom anomaly* is not relevant in this paper.

In the following, we will only describe the countermeasures that are used in the e-commerce examples of section 3. We will first describe a countermeasure against the lost update anomaly, because it is the most important anomaly to guard against.

The Commutative Updates Countermeasure

Adding and subtracting an amount from an account are examples of commutative updates. If a subtransaction only has commutative updates, it may be designed as commutable with other subtransactions that only have commutative updates. 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 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 updates which limit the options of the users.
- Retriable subtransactions for updates 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 (or approximated atomicity), the global durability property (or approximated durability property) will automatically be implemented, as it is ensured by the log-system of the local DBMS systems [9].

3. A Transaction Pattern for Atomicity Implementation

In this section, we will describe a general transaction pattern that can simplify the atomicity implementation. In order to implement the atomicity property, the transaction pattern must follow the rules of our nested transaction model described in section 2. After the presentation of the transaction pattern, examples will illustrate how to use the transaction pattern.

The following figure illustrates a UML statechart diagram for a global transaction. The syntax for a transition has tree parts, all of which are optional: *Event* [*Guard*] / *Action*. In the diagram, all the events are either subtransactions submitted by the user or subtransaction aborts. All the event actions and state activities of the diagram must be designed with “subtransaction atomicity”. In the diagram, we do not deal with subtransaction aborts that do not change the state of the global transaction, because the user without problems can resubmit these subtransactions. All the event guards coming from the same state in the diagram are mutually exclusive, and, therefore, the diagram does not have loose ends.

The description of the event actions and state activities is a pattern without application logic, i.e., only the necessary database access types of the actions/activities are described. It is important to distinguish between two types of locations for each subtransaction:

- The “execution location” is the location where a subtransaction is executed. For example, the “root location” is the execution location of the root transaction, and it is normally the user’s PC. The “pivot location” is the location where the pivot subtransaction is executed.
- A “log location” is the location where the parameters of a subtransaction are stored. If an event corresponding to a subtransaction changes the state of the global transaction, the new state is also stored in the log location.

Often, the log location of a subtransaction is not the same as the execution location, and, in such a situation, it is important to make the updates in the two locations atomic (or approximated atomic). Otherwise, recovery will be much more complex.

If a subtransaction fails and/or the user does not get an answer, it is important for the user to know the state of

the global transaction. Therefore, a State record must first be created in the log location of the root transaction. Later, a new State record is created in the log location of the compensating subtransactions, and finally, a State record is created in the log location of the pivot subtransaction. By reading these State records it is always possible to find the state of the global transaction.

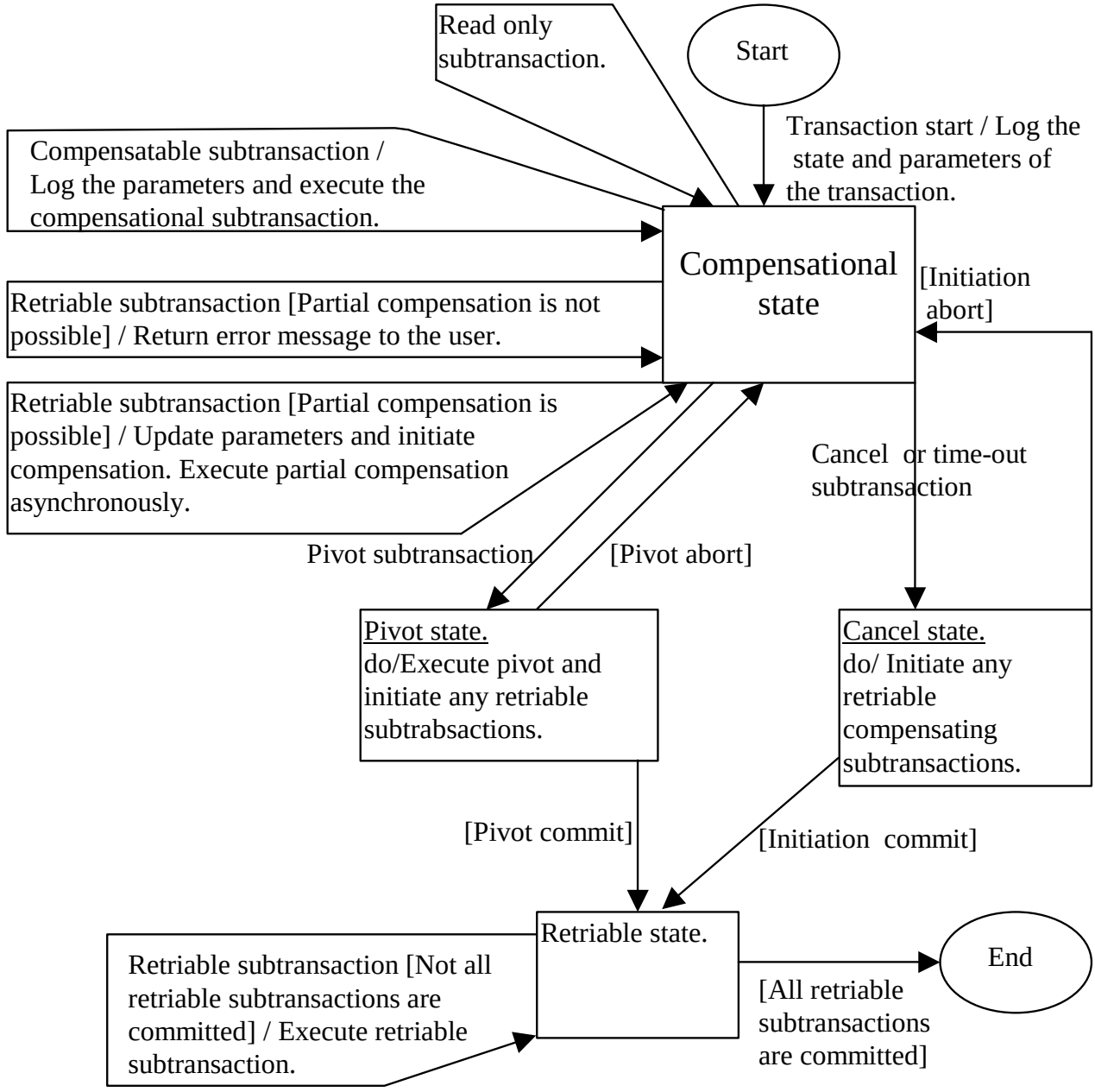

Figure 3.1. Statechart diagram for a general global transaction.

In the following, we will illustrate how to use our nested transaction model in E-commerce systems. The first example outlines how business-to-business E-commerce transactions may be designed by using our transaction model. The second example describes a more complex business-to-consumer E-commerce transaction.

In contrast to the business-to-consumer segment, business-to-business E-commerce may anticipate trust between the customer and seller. Hence, global transactions need not involve a third party, such as the bank of the customer or seller. This simplifies the global transaction in Example 3.1 compared to Example 3.2, which describes the global transaction of a retail customer.

Example 3.1

In this example of business-to-business E-commerce, we will assume that the seller has a customer file with the names, addresses, account balances and credit limits for all his customers. Therefore, the banks of the customers

are not involved in the following description of the order transaction. In this example, we choose the local server of the seller as both pivot location and log location for all the subtransactions. The root location is the user's local PC. Other locations are any remote stock servers of the seller.

At first, the customer reads the offers made by the seller. If the customer wants to make an order, the root transaction in the location of the customer calls a compensatable subtransaction at the location of the seller. This subtransaction creates an order record with relationship to the customer record at the same location. A new State record with the value "Compensatable state" is created for the transaction. Now, the customer can make order-lines. For each new order-line made by the customer, the root transaction starts a compensatable subtransaction, and this subtransaction creates an order-line at the location of the seller. For each order-line, a compensatable sub-subtransaction updates the local (or remote) stock of the product ordered in the order-line. If the first stock cannot fulfill the quantity ordered in the order-line, another stock may be accessed by using another compensatable sub-subtransaction. If an order-line cannot be fulfilled, the field "quantity-delivered" in the order-line is updated. Please notice that only *short duration locks* are used, and, therefore, distributed deadlock cannot occur. When the order form has been completed, the pivot subtransaction is executed at the location of the seller where it updates the account balance of the customer. If the credit limit of the customer is not violated, the pivot subtransaction initiates a retriable sub-subtransaction that is sent to the customer to confirm the deal. The pivot subtransaction also changes the State record to "Retriable state" before all the updates are committed. Alternatively, the global transaction will be rejected or the customer asked to reduce the amount of the balance in order to avoid violating the credit limit.

By executing a retriable subtransaction that reduces the quantity ordered in an order-line, the amount in the order-line may be reduced. For each reduced order-line, a retriable sub-subtransaction increases the local (or remote) stock of the product that is reduced in the modified order-line. Finally, the customer can retry to execute the pivot subtransaction.

Example 3.2

In this example, we will describe the atomicity implementation of a business-to-consumer E-commerce transaction where the global transaction also involves the banks of the customers and the seller. In the example, the bank of the customer is used as the pivot location. The server of the seller is the log location of all the subtransactions. The PC of the user is the root location. When the retail customer wants to make an order, the first compensatable part of the global transaction may be the same as in the previous example, where the order and order-lines were created. However, the seller may not know the customer, and, therefore, a compensatable customer record should also be established. Before the pivot subtransaction is executed, the balance of the customer is updated by a compensatable subtransaction and the State record changed to "Pivot state". The pivot subtransaction is executed at the bank of the customer, where payment of the customer may be accepted/committed and a retriable subtransaction to the seller initiated. When the retriable subtransaction of the pivot is received in the location of the seller, the State record of the global transaction is changed to "Retriable state", and the account of the customer updated. A retriable sub-subtransaction may also be initiated in order to confirm the deal for the customer.

4. Conclusions

Normally, distributed systems do not have the traditional ACID properties because they will reduce availability and write performance. In this paper, we recommend using approximated ACID properties, i.e. from an application point of view the system should function as if all the traditional ACID properties had been implemented.

The distributed approximated atomicity property manages the workflow of a distributed transaction in such a way that either all the updates of the transaction are executed (sooner or later) or all the updates of the transaction are removed/compensated. In this paper, we have described in detail how distributed approximated atomicity may be implemented. That is, we have described a nested transaction pattern (algorithm) designed for updating multidatabases with approximated atomicity. The transaction pattern can be used for all types of distributed updates, and, therefore, it may reduce time for design and programming. Our transaction pattern makes it possible to nest all types of subtransactions to any depth, In addition, the root location (normally the PC of the user), the log location (the location where the recovery information is stored) and the pivot location (the location where the global transaction is committed) may be different locations or grouped in any combination. Another feature is that our transaction pattern allows retriable subtransactions to be executed before the global commit if the retriable subtransactions more or less compensate compensatable subtransactions that have already been committed locally but not globally.

We have illustrated how to use the transaction pattern in E-commerce systems. For example, if one of the partaking subsystems fails in a system for E-commerce, the approximated atomicity property will ensure that when an order is accepted (the global commit), the stock levels, payment etc., are managed automatically in the locations of the partaking product stocks and banks. Even logistics and/or production workflow may be managed by using approximated atomicity.

We have cooperated with one of the major ERP software companies in designing a distributed version of the ERP system with local autonomous databases in the different sales and stock locations.

References

[1] L. Frank and T. Zahle, 'Semantic ACID Properties in Multidatabases Using Remote Procedure Calls and Update Propagations', *Software - Practice & Experience*, 1998, 28, pp77-98.

[2] H. Garcia-Molina and K. Salem, ' Sagas', *ACM SIGMOD Conf*, 1987, pp 249-259.

[3] 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.

[4] G. Weikum and H. J. Schek, 'Concepts and Applications of Multilevel Transactions and Open Nested Transactions', *A. Elmagarmid (ed.): Database Transaction Models for Advanced Applications*, Morgan Kaufmann, 1992, pp 515-553.

[5] 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.

[6] L. Frank, 'Integration of Different Commit/Isolation Protocols in CSCW Systems with Shared Data', *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference*, PSI'99, Springer-Verlag, pp 341-351.

[7] J. Gray and A. Reuter, *Transaction Processing*, Morgan Kaufman, 1993.

[8] H. Berenson and 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.

[9] Y. Breibart, H. Garcia-Molina and A. Silberschatz, 'Overview of Multidatabase Transaction Management', *VLDB Journal*, 2, 1992 pp 181-239.

[10] L. Frank, 'Integrity Problems in Distributed Accounting Systems with Semantic ACID Properties', *Proc. Third Annual IFIP TC-11 WG 11.5 Working Conference: Integrity and Internal Control in INF Systems*, Amsterdam, The Netherlands, November 18-19, 1999.

[11] L. Frank, 'Evaluation of the Basic Remote Backup and Replication Methods for High Availability Databases', *Software - Practice & Experience*, Vol. 29, issue 15, 1999, pp 1339-1353.

[12] Wil van der Aalst, Process-Oriented Architectures for Electronic Commerce and Interorganizational Workflow, *Information Systems*, Vol. 24, No. 8, pp. 6399-671, 1999.

[13] V. Zwass, Electronic commerce: structures and issues, *International Journal of Electronic Commerce*, Vol. 1, No. 1, pp. 3-23, 1996.

[14] L. Frank, 'Atomicity Implementation in Cooperative Distributed Systems With High Performance and Availability', *Proc of Collaborative Technology Symposium 2002*, January 2002.