Transaction Compensation in Web Services

Thomas Strandenæs*, Randi Karlsen**

*Norwegian Centre for Telemedicine, University Hospital of North Norway, N-9037 Tromsø Thomas.Strandenaes@telemed.no

**Department of Computer Science, University of Tromsø, N-9037 Tromsø randi@cs.uit.no

Abstract: In this paper we describe a technique for implementing compensating transactions, based on the active database concept of triggers. This technique enables specification and enforcement of compensation logic in a manner that facilitates consistent and semi-automatic compensation. A web service, with its loosely-coupled nature and autonomy requirements, represents an environment well suited for this compensation mechanism.

1. Introduction

Transaction processing techniques play a major role in preserving data consistency in critical areas of computing, from hospital information systems to trading systems. A transaction maps a database from one consistent state to another by executing a sequence of operations atomically. Effects of concurrency are automatically hidden, and failures recovered. Transactions are therefore appropriate building blocks for structuring reliable systems [1].

The reliability provided through transactional guarantees are required in many types of applications, found in for instance workflow systems, mobile systems, and lately also in web services based systems. As new transactional applications have emerged, the limitations of classical transactions have been recognised and are well known [2]. To handle transaction processing within these application domains, a number of alternative or extended transaction models have been suggested [3] [4] [5] [6] .

Web services provide interoperable application-to-application communication, allowing new applications to leverage existing software functions in a platform independent fashion. The transactional behaviour of a function accessed through a web service depends on the underlying implementation of the web service. Often a database system will provide the required local transactional behaviour. However, when an application combines multiple web services in order to complete a given task, coordination of the participating web services is required in order to preserve data consistency. Traditionally two phased commit (2PC) based protocols have been used to achieve such coordination (e.g. X/Open DTP, CORBA OTS). Because of the loosely-coupled nature and autonomy requirements of web services, 2PC-based protocols may however not be appropriate in this environment.

Efforts are currently under way to design and implement appropriate web service transaction models, see for instance the WSTx framework [7]. Similar to earlier work on extended transaction models, this and other models rely on the concept of compensation in order to preserve the autonomy requirements of web services. Here individual web service invocations may commit early without further coordination,

provided that the effect of the invocation can be semantically reversed at some later point by executing a *compensating transaction*. Typically, compensating transactions are not focused in the design of web service transaction models, and implementation of this functionality is left to the application developer, here web service developer. In this paper we focus on providing support for compensating transactions.

We assert that the trigger mechanism, more formally known as Event-Condition-Action rules [8], may be leveraged to implement support for compensating transactions in an efficient and uniform manner. By building transaction compensation on the trigger mechanism, we allow a web service designer to specify compensation rules, which are used to dynamically generate compensating transactions during runtime. This approach supports both easy and consistent compensation and semi-automatic compensation as seen from the service user. It should be noted that the mechanism proposed in this paper is not restricted for use within web services only. Other environments, relying on compensation for transaction recovery, may also benefit from this approach.

This paper is organized as follows. Section 2 motivates for our work in some more detail. Section 3 describes both the trigger-based mechanism for transaction compensation and a prototype implementation. Fundamentals of triggers are also reviewed in section 3. In Section 4 we conclude.

2. Web services and compensating transactions

A web service is either a stand-alone service or a composite service relying on other web services to perform its task. Service composition allows combination of a number of component services in order to implement the required application logic. Today's web services typically expose their interfaces using an interface description language like WDSL (Web services Description Language [9]), and may be located by querying a catalogue service like UDDI (Universal Description, Discovery and Integration [10]). Services are then accessed using SOAP [11], the XML based messaging protocol, typically through transport protocols like HTTP and TSL.

Interaction between web services, are typically handled through *conversational transactions* [12], that involve participation from several web services. The unit of business at each web service represents a subtransaction, also called component transaction. The transactional behaviour of a single subtransaction is typically provided locally by an underlying database system. Additionally, transactional behaviour of the conversational transaction must be guaranteed through coordination and management of the set of subtransactions.

Unlike traditional distributed transactions, conversational transactions do not assume the classical ACID properties. Because of the loosely-coupled nature and autonomy requirements of web services, traditional Two phase commit (2PC) based protocols are for instance not appropriate. If one or more subtransactions abort, the conversational transaction may or may not need to be cancelled depending on the business logic of the service. It is totally up to the web service starting the conversation to decide if all-or-nothing semantics should be enforced.

To preserve the autonomy requirements of web services, proposed transaction models such as the WSTx framework allow component web services to complete unilaterally (as opposed to blocking while waiting for a global commit decision in 2PC),

externalising partial results of the conversational transaction. This is called early commit of a subtransaction, and happens before a decision is made on the outcome of the overall task (the conversational transaction) performed by the composite service.

Other transactions that read and possibly update partial results of conversational transactions are called *dependent transactions*. Within classical transaction processing, dependent transactions would have to be aborted if the dependent-upon transaction aborted. Resorting to cascading rollback of dependent transactions is however not acceptable in web services, both since dependent transactions may themselves be committed, and also since rollback may be disallowed by autonomous web services. Autonomous web services typically consider the results of a committed subtransaction as final and durable.

For recovery of committed subtransactions, several researchers have proposed using *compensating transactions* [13], [14], [4], [15], which semantically undo the results of the early committed subtransactions. A compensating transaction preserves database integrity without aborting other transactions. The concept of a compensating transaction was first proposed in by Garcia-Molina [4], and has later been included in a number of transaction models. A compensating transaction is defined as follows:

Definition: a **compensating transaction** semantically undoes the partial effects of a transaction *T* without performing cascading abort of dependent transactions, restoring the system to a consistent state.

A general rule for designing compensating transactions is to first perform a logical undo of the compensated-for transaction operations, and then restore consistency or establish other desirable properties [16]. For instance, after removing an inserted data element (undo), some stored statistics may be wrong and need to be updated to restore consistency in the database.

Designing compensating transactions can be viewed as a hard problem in general. However, many of the scenarios handled by traditional transaction systems today have complementing compensating events that are considered as a part of normal processing. Today the logic for undoing the results of a transaction is often implemented as an integrated part of the application system. For instance, the removal of a resource reservation (e.g. trip planning application) is executed as a normal forward transaction, initiated by the application user.

Some activities cannot be compensated for. This is a situation that must be handled by traditional transaction processing systems as well, for example when dispensing money from an automated teller machine (ATM). Such actions are often called real [2]. The traditional approach here is to delay the real action until the end of the transaction, when other operations that may result in an abortion of the transaction have already completed, and the risk for abortion is relatively low. This approach can be applied in an environment where compensating transactions are used as well, by reordering subtransactions so that the real action comes at the end of the transaction. Some real actions may have natural compensating steps as well. For instance, sending a letter to inform the account owner of the account overdraft and arrange for payback may be considered as a compensating step for a real event.

The business rules or logic used in normal applications is a source of information for designing compensating transactions. In a web service environment, it is however not acceptable to let the service user implement the logic required for compensation. Designing compensating transactions must be considered on a per web service basis.

In this paper we propose a mechanism for specifying and executing compensating transactions by using triggers, the fundamental concept found in active databases. This mechanism is used at the web service level, and enables specification and enforcement of compensation logic in a manner that facilitates easy and consistent compensation. The web services designer is responsible for determining the compensation rules, which are used to dynamically generate compensating transactions during runtime.

3. Trigger-based compensation

Trigger-based compensation is an approach for specifying and executing compensating transactions based on the active database concept called triggers, more formally known as Event-Condition-Action-rules (ECA-rules). To apply this approach within the web services domain, the web service must be implemented on top of a database system that supports triggers. Support for classical transactions (ACID-transactions) must be provided by the database system as well. Both triggers and classical transactions are however available in all major database systems today, and required standards support is provided in SQL:1999 [17].

3.1 Introduction to triggers

Support for triggers is the distinguishing feature of an active database. An active database system is a database system that monitors situations of interest, and when they occur, triggers an appropriate response in a timely manner [19]. Triggers have the following general form:

```
on Event
if Condition
do Action
```

Perform Actions upon certain Events and under the given Conditions. Event sources include data modification events (i.e. `insert`, `update` and `delete` operations on tables in SQL), transaction operation events (i.e. `begin, commit and abort transaction`) and others.

Support for triggers extend the functionality of the database system in a number of ways. In a traditional passive database system, external applications must poll the database to evaluate conditions, and the actions must be implemented in the applications. With triggers, rules may be monitored continually inside the database system, and actions are implemented within the database system itself.

3.2 Specifying and executing compensating transactions using triggers

Compensating transactions are determined through three steps. First, the rules for generating compensating operations are specified by the web service designer based on the application logic. Secondly, the database system generates compensating operations, according to the rules, during the execution of the subtransaction. Finally when the subtransaction commits, compensating operations for the subtransaction are

combined into a compensating transaction when the subtransaction commits. The generated operations of the compensating transaction are stored in the database for later execution according to the rules of the recovery protocol.

Compensation rules are defined by using triggers in the database for all subtransaction operation types that change the state of the database (such as `Update`, `Delete` and `Insert` operations) and hence require compensation. The triggers are specified by the database schema designer or web service developer using the trigger language of the DBMS. During execution of the subtransaction each operation generates an event, firing one of the defined triggers if the operation updates the database state. In the Action-part of the trigger the context of the event is then analysed, typically by comparing old and new database states, operation type and parameters. If it is determined that the subtransaction operation cannot be compensated for, the operation may either be rejected or the subtransaction is not allowed to commit. Otherwise information required to perform a compensating operation is extracted from the event context and stored in the database. For instance, when a subtransaction inserts a new data element the 'Insert' event is generated. Simplified code for the insert-trigger may look like this:

```
on Insert
do storeCompensatingOperation ('Delete', newElement)
```

Using this rule, every `insert` operation in the subtransaction will result in the storage of a compensating operation that deletes the inserted data element when executed. The example function `storeCompensatingOperation` is specified by the trigger designer, and here requires two input parameters. The first parameter 'Delete' indicates the type of compensating operation. The second parameter identifies the data element, and is extracted from the event context as the special variable `newElement` that references the element inserted by the subtransaction operation. When `newElement` is used as the input parameter, this function will therefore generate compensating operations for all insert operations. The rule is visualised in Figure 1.
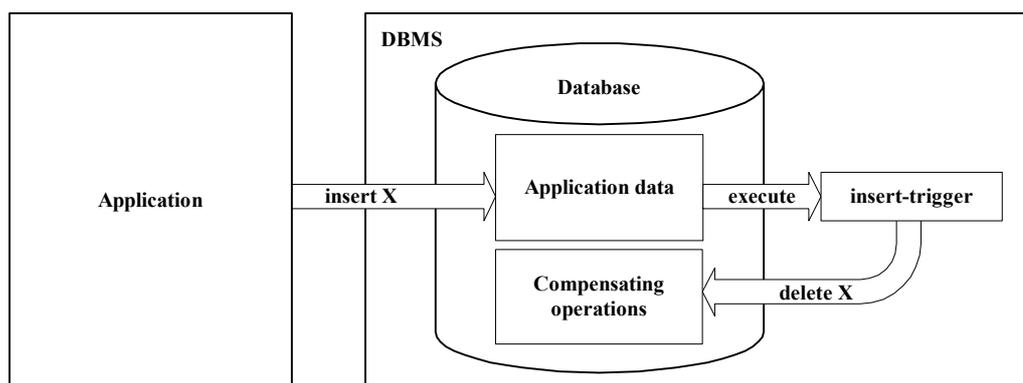

Figure 1: Generating and storing compensating operations

Details of storing compensating operations are abstracted in the example `storeCompensatingOperation` function. Here it suffices to notice that the compensating operations are stored as regular data elements in the database.

The simplified `storeCompensatingOperation` function from the example above may be substituted by functions that embed the business logic for compensation in the specific database system. These functions can be of arbitrary complexity, using the full range of constructs allowed by the DBMS rule language. The event context is used to extract context related information required to perform the compensating operation(s), as well as in conditional statements to determine whether and how compensation can be performed. Compensating operations range from operations that resemble physical undo (like in the example above) to entirely different operations, invocation of external functions etc. The triggers for generating compensating operations have the following general format:

```
on Event
do Generate Compensating Operations(Event Context)
    If compensation cannot be performed Then reject the operation Or
    enforce commit-dependency between subtransaction and top-level
    transaction
    Else Store Compensating Operations
```

The condition clause (`if` clause) of the trigger is left out, as the Action-part of the rule should execute unconditionally.

The triggers for generating compensating operations execute within the transaction context of the subtransaction, and are executed immediately after the event is signalled. This is called immediate coupling mode [20]. Executing within the transaction context of the subtransaction is important to ensure that only committed subtransactions are compensated for. Should the subtransaction abort during execution, all compensating operations that have been generated during the execution of the subtransaction will be aborted as well. The generated compensating operations will be stored permanently in the database only when the subtransaction commits.

When all the operations in the subtransaction have been processed by triggers as described above and the subtransaction has committed, the stored compensating operations constitute the compensating transaction. The DBMS generates a transaction operation event when the subtransaction commits (typically when the application issues a `commit` operation). This event is utilized to group the generated compensating operations into a compensating transaction. When the next subtransaction executes, the generated compensating operations will belong to a new compensating transaction.

The process described above is repeated for each subtransaction in the top-level transaction. Should the top-level transaction require recovery, the stored compensating transactions will be executed according to the rules of the recovery protocol.

3.3 Architecture

Having described the fundamental concepts of trigger-based compensation, we will now summarize the overall architecture of this approach. The trigger-based approach utilizes triggers defined on a number of events to enable specification of compensating transactions. In addition to the data modification and transaction operation events (see section 3.1), triggers defined on system events such as the 'system restart' event, in order to initialising recovery. User-defined events may be invoked to get an identifier

for the currently executing subtransaction. This identifier may later be used by an application in order to initiate recovery for a specific subtransaction.

In Figure 2 the architecture of ECA-based compensation is summarized, showing the event types involved. Also, storage structures related to compensating transactions are depicted.
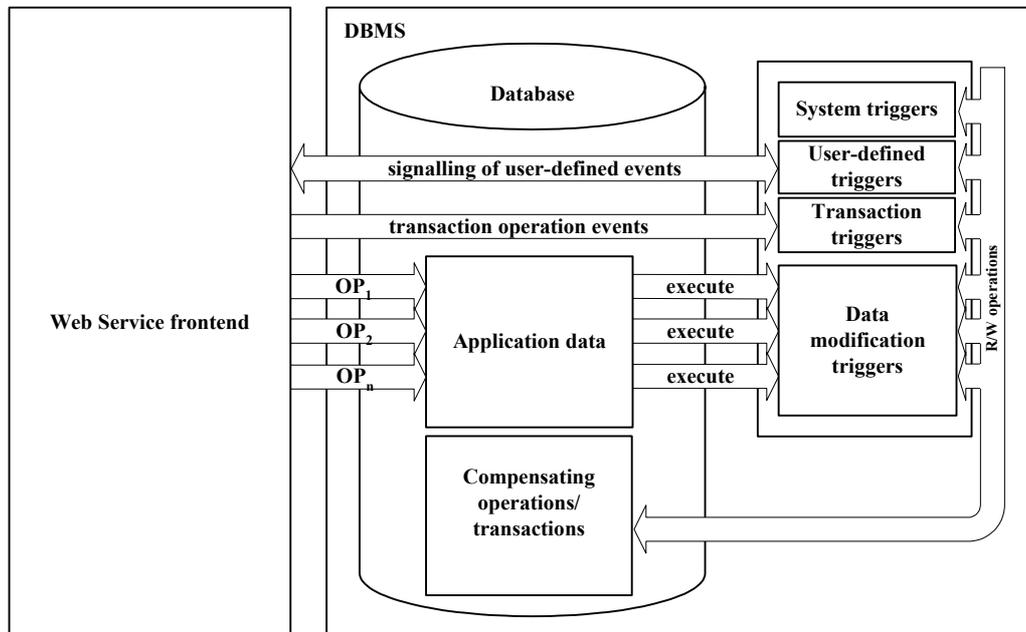

Figure 2: architecture of trigger-based compensation

The functionality of the built-in DBMS transaction manager at a local web service is not affected in this approach. Subtransactions and compensating transactions are executed as classical transactions, and are handled by the DBMS as normal transactions.

3.4 Recovery Protocol

Recovery of a conversational transaction will generally involve both rollback of uncommitted subtransactions and compensation of committed subtransactions. The composed service can initiate abort of the conversational transaction unilaterally at any time during execution. This is performed by invoking the user-defined event 'abort top-level transaction'. The system can decide to abort the conversational transaction as well, for instance during system restart.

The DBMS transaction manager at a component web service is assumed to handle recovery of classical transactions automatically. Since both subtransactions and compensating transactions are executed as classical transactions, recovery of executing subtransactions and executing compensating transactions is handled automatically by the DBMS using traditional undo/redo-based approaches. The following algorithm is used to abort a committed subtransaction T:

Abort[T]:

1. Abort the currently executing subtransaction using the DBMS rollback facility.
2. Set the execution status of subtransaction T to '*aborting*'.
3. Get all stored compensating operations for the compensating transaction that have not been executed. Perform step a-d below to execute the compensating transaction.
    a. Start a new transaction. Execute the compensating operations in the order specified in storage
    b. Set the execution status of the compensating transaction to '*executed*'.
    c. Commit the compensating transaction
    d. If the transaction fails in step a-c above, perform rollback and retry step a-c n times. If the subtransaction still fails, set the execution status of the top-level transaction to '*failed recovery*' and inform system administrator of the failed compensating transaction and go directly to step 6 without executing remaining compensating transactions.
4. Acknowledge completed abort

Idempotency of compensating transactions is ensured in this algorithm. Should the compensating transaction fail, e.g. if a system crash occurs before step 3c is completed, the results of the compensating operations and updated execution status will be rolled back by the DBMS during recovery. Notice that abort completes even if a compensating transaction fails after being retried. The system administrator is then responsible for handling recovery for this top-level transaction. The system administrator may inspect the operations of the compensating transaction that failed and possibly re-invoke the abort command after the problem has been resolved.

3.5 Advantages of the trigger-based approach to compensation

The trigger-based approach provides central specification of compensation policies. In an autonomous web service environment, a central authority with knowledge of the business logic (for instance a web service designer), can now be responsible for determining uniform and consistent compensation policies. Since compensation is an integral part of database (or service) consistency, this responsibility should not be given to external users of the web service. With central enforcement of triggers, there is no way of bypassing the execution of the triggers. This means that the compensation policy implemented by the triggers is enforced at all times (as long as the triggers are not disabled).

This approach facilitates semi-automatic generation of compensating transactions based on the compensation rules. The triggers may generate different compensating transactions depending on the event states seen during execution. Conditions and rules for compensation are expressed in the triggers, similar to other centrally specified consistency constraints like referencing integrity.

Compensation is transparent seen from the service user's point of view. The native language of the database system, e.g. SQL in an SQL-based system, is used for updating the database, and there is no need to specify additional information for use during compensation. This gives the service an impression of automatic recovery. The complexity of writing applications that use service composition (and possibly

compensation) is reduced since the specification of compensating transactions is abstracted from the design of the application program.

The trigger-based approach also provides a modular way of expressing compensation policies. In a relational DBMS for instance, policies may be specified on a per table and operation type basis. Furthermore, support for trigger-based compensation can be added at any time in an existing database and affect only the parts of the database where compensation should be used. Use of triggers for compensation does not use up/deplete the triggers as a resource either, i.e. it will still be possible to use triggers in the database for other purposes in addition to compensation.

3.6 Implementation

We have implemented a proof-of-concept demonstrator for trigger-based compensation using Oracle8 RDBMS. Oracle8 provides a relatively feature-rich trigger mechanism that conforms fairly well to SQL:1999. Oracle also provides a high-level procedural extension called PL/SQL, which was found adequate for implementing trigger logic. It is important to notice however, that any DBMS supporting triggers to a certain extent could have been used here instead of Oracle8.

Central to this prototype implementation is the compensation service, depicted in the bottom right of the architecture in Figure 3 below. The compensation service is implemented as an Oracle PL/SQL package, called DBMS_COMPENSATE.

To activate the compensation service for a given schema, the schema designer must invoke the subprogram DBMS_COMPENSATE.installService. This will install triggers for the system restart event in the current schema. These triggers implement parts of the recovery protocol. Tables required for maintaining compensating transactions and other internal structures of the service are created as well.
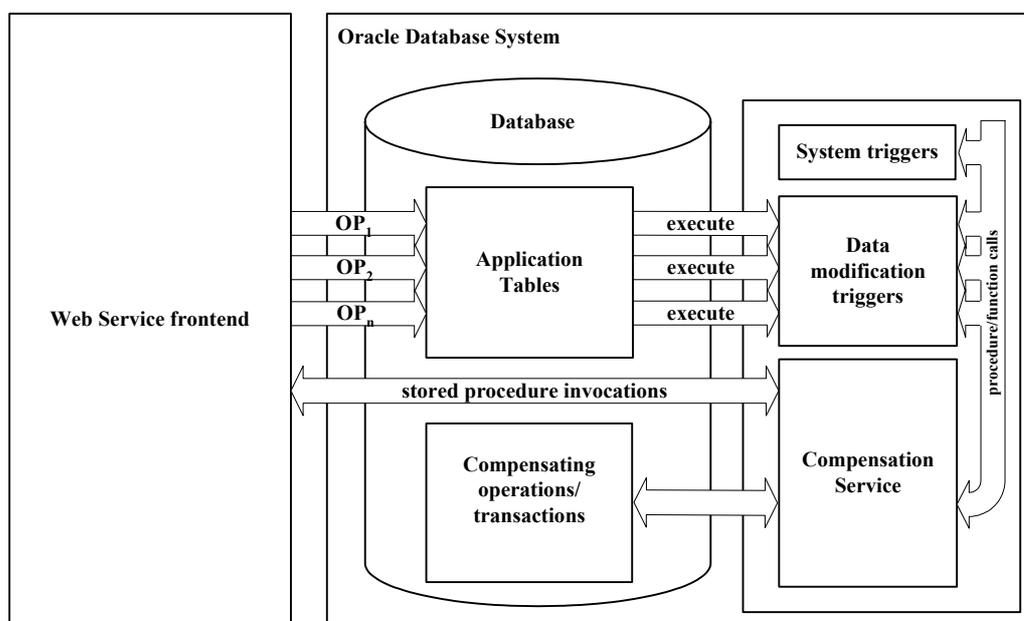

Figure 3: architecture of compensation service

The schema designer specifies compensating transactions by handcrafting triggers that make use of the compensation service. Triggers are defined for data manipulation events at both row and statement level on relevant tables. These triggers examine the new and old database state to extract information used for compensation, or reject the operation if compensation cannot be performed. The compensation service has two types of interfaces, an application interface and a schema designer interface.

Application interface
The application interface describes the methods available for the web service front-end to start, end and restart execution of conversational transactions, and to set persistent save points. This interface is implemented using stored procedures. Figure 4 summarizes the methods available in this interface:

Method	Description
`beginConvTransaction`	Begins a new conversational transaction
`commitConvTransaction`	Commit conversational transaction
`abortConvTransaction`	Aborts the conversational transaction
`continueConvTransaction`	Continues the execution of a conversational transaction after some failure
`getConvTransactionStatus`	Gets the execution status of the conversational transaction

Figure 4: summary of application interface methods

The application may initiate recovery of the conversational transaction by using the `DBMS_COMPENSATE.abortConvTransaction` procedure. If the aborting conversational transaction has committed one or more subtransactions, compensating transactions are executed to undo the results of the subtransactions. Recovery of committed subtransactions is executed according to the protocol described in section 3.4.

Schema designer interface
The main function of the schema designer interface is to hide the complexity of storing compensating transactions. Methods in this interface is used by the schema designer for installing triggers holding the rules for generating compensating operations, and for storing compensating operations and associating these operations to a compensating transaction. Figure 5 summarizes the methods available in this interface:

Comments to the implementation
The implementation handles different error situations. Compensating transactions may for instance be rejected when deadlocks are detected. This is handled by catching the exception that is raised by Oracle when a deadlock occurs. The compensating transaction is then re-executed. Other error situations are dealt with in a similar manner.

If the compensating transaction is aborted N number of times (N can be configured in the compensation service, but is initially set to 4) because of some error during the execution of the compensating transaction, this is considered a recovery failure. The

failure is then logged, and the execution status of the top-level transaction is set to '*failed_recovery*'. The system administrator is required to resolve this issue. Email notification of the recovery failure is sent using SMTP.

Method	Description
isConvTransactionValid	Verifies that a valid and active conversational transaction exists in this session
addOperation	Stores a compensating operation and associates the operation with a compensating transaction
installTriggers	Utility function for installing default triggers for a specified table, used for generating compensating operations

Figure 5: summary of schema designer interface methods

In the current implementation of the system, each compensating operation relates to one operation in the subtransaction. This is due to the immediate coupling mode used by the Oracle8 DBMS, in which the trigger is executed immediately after the event is signalled. Other active DBMSs support different coupling modes, such as the deferred coupling mode. Here the trigger is executed at the end of the triggering transaction, but still in the same transaction context as the triggering transaction. With deferred coupling mode, compensating operations may have been determined in a more flexible manner, taking the more than one subtransaction operation into consideration.

Through this implementation we have demonstrated how trigger-based compensation may be used inside a widely used DBMS, using only facilities provided by the DBMS itself. For more details on the implementation, see [18].

4. Conclusion

Web service environments, with their loosely-coupled nature and autonomy requirements, need to base transaction execution on extended transaction models. In particular, these environments need the concept of compensating transactions in order to preserve autonomy of participating component transactions. In this paper we have proposed a trigger-based mechanism and described a prototype implementation for specifying and executing compensating transactions. This approach has several advantages. It provides central specification and enforcement of compensation policies, and is a modular and uniform way of expressing compensation policies. Also it provides a semi-automatic mechanism for recovery using compensation for the applications using the mechanism, as the mechanism is implemented inside the DBMS.

References

1. Ramamritham, K. and Chrysanthis, P.K. *Executive briefing: advances in concurrency control and transaction processing*. IEEE Computer Society Press, Los Alamitos, California, 1997.
2. Gray, J., The Transaction Concept: Virtues and Limitations. in *VLDB*, (Cannes, France, 1981), 144-154.

3.   Moss, J.E.B. Nested Transactions: An Approach to Reliable Computing, MIT, Cambridge, MA, 1981.

4.   Garcia-Molina, H. and Salem, K. SAGAS. in Stonebraker, M. ed. *Readings in database systems*, San Francisco, California, 1987, 290-300.

5.   Pu, C., Kaiser, G. and Hutchinson, N., Split transactions for Open-Ended activities. in *VLDB*, (Los Angeles, California, 1988), 26-37.

6.   Reuter, A., ConTracts: A Means for Extending Control Beyond Transaction Boundaries. in *International Workshop on High Performance Transaction Systems*, (Asilomar, California, 1989).

7.   Mikalsen, T., Tai, S. and Rouvellou, I., Transactional Attitudes: Reliable Composition of Autonomous Web Services. in *Dependable Systems and Networks Conference*, (Maryland, USA, 2002).

8.   Dittrich, K., Gatziu, S. and Geppert, A. The Active Database Management System Manifesto: A Rulebase for ADBMS Features. *ACM Sigmod Record*, *25* (3). 40-49.

9.   Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S., Web Services Description Language (WDSL) 1.1, 2001, URL: http://www.w3.org/TR/wsdl

10.   ARIBA/IBM/Microsoft, UDDI Technical White Paper, 2000, URL: http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.PDF

11.   Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S. and Winer, D., Simple Object Access Protocol (SOAP), 2000, URL: http://www.w3.org/TR/SOAP/

12.   Dan, A. and Parr, F., An Object implementation of network centric business service application (NBCSAs): conversational service transactions, service monitor, and an application style. in *OOPSLA '97, Business object Workshop III*, (Atlanta, Georgia, 1997), ACM Sigplan.

13.   Garcia-Molina, H. Using Semantic Knowledge for Transaction Processing in Distributed Database. *ACM Transactions on Database Systems*, *8* (2). 186-213.

14.   Farrag, A.A. and Özsu, M.T. Using semantic knowledge of transactions to increase concurrency. *ACM Transactions on Database Systems*, *14* (4). 503-525.

15.   Korth, H.F., Levy, E. and Silberschatz, A., A Formal Approach to Recovery by Compensating Transactions. in *Very Large Data Bases - 16th international conferece on Very Large Data Bases*, (Brisbane, Australia, 1990), 95-106.

16.   Levy, E. Semantics-Based Recovery in Transaction Management Systems *Department of Computer Sciences*, University of Texas at Austin, Austin, Texas, 1991.

17.   ISO/ANSI/IEC. Information Technology - Database Language - SQL *ANSI/ISO/IEC 9075-[1-5]-1999*, 1999.

18.   Strandenæs, T. ECA-Based Transaction Compensation *Department of Computer Science*, University of Tromsø, Tromsø, 2002, 200.

19.   Dayal, U., Hanson, E.N. and Widom, J. Active Database Systems. in Won, K. ed. *Modern Database Systems: The Object Model, Interoperability, and Beyond*, ACM Press and Addison-Wesley, Reading, Massachusets, 1995, 434-456.

20.   Paton, N.W. and Díaz, O. Active Database Systems. *ACM Computing Surveys*, *31* (1).