An Architecture for Nested Transaction Support on Standard Database Systems*

Erik M. Boertjes, Paul W.P.J. Grefen, Jochem Vonk, Peter M.G. Apers

Center for Telematics and Information Technology
University of Twente
{boertjes,grefen,vonk,apers}@cs.utwente.nl

Abstract. Many applications dealing with complex processes require database support for nested transactions. Current commercial database systems lack this kind of support, offering flat, non-nested transactions only. This paper presents a three-layer architecture for implementing nested transaction support on a commercial multi-database environment. The architecture is directed at high portability and flexibility. The modular approach and the simple, event driven interfaces between the layers of the architecture enable the nested transaction support to be adapted to various applications, nested transaction models and database management systems. The architecture has been implemented to support a prototype of a commercial next-generation workflow management system.

1 Introduction

There is an increasing use of database management systems (DBMS's) in advanced applications that deal with complex processes. In the WIDE Esprit project [Cas96a, Cas96b, Cer97, Gre97], a multi-level process model is adopted, allowing hierarchical decomposition of such complex processes (e.g. workflow processes). The process semantics of the higher levels of the process hierarchy are usually different from those in the lower levels of the hierarchy. To comply with these process semantics, extended transaction support is needed that goes beyond the standard transaction support offered by standard, commercial DBMS's.

Higher level processes are in general long running processes with cooperative characteristics. These characteristics demand that these processes should not be executed in strict isolation. Strict isolation would prevent cooperation between processes as they would not be able to share information. Strict atomicity of long running processes is not desirable either. A process failure (e.g. a system crash) would cause the loss of all work done so far by the process. Processes lower in the hierarchy are relatively short living processes requiring strict transactional semantics. The separation between the higher and lower levels is formed by the notion of business transaction in the workflow application, being a subprocess that is composed of subprocesses that together form an atomic unit of work from an application point of view. This atomicity is not completely strict however, because of the notion of *critical* and *non-critical* subprocesses. In general there will be subprocesses whose successful completion is not a necessary condition for the successful completion of their parent. We call such sub-

* The work presented in this paper is supported by the European Commission in the WIDE Project (ESPRIT No. 20280). Partners in WIDE are Sema Group sae and Hospital General de Manresa in Spain, Politectnico di Milano in Italy, ING Bank and University of Twente in the Netherlands.

G. Quirchmayr et al. (Eds.): DEXA'98, LNCS 1460, pp. 448–459, 1998.

processes non-critical. Critical subprocesses are processes that have to be completed successfully in order to let their parent complete successfully. Business transactions need to be strictly isolated from each other as intermediate results of business transactions have no semantics. Since subprocesses within business transactions in general share data, isolation within business transactions is relaxed.

Traditional flat transaction models fail to model the process semantics of both levels described above because they do not allow transactions to be nested neither do they support relaxed atomicity nor relaxed isolation. Therefore, extended transaction support is desired. As current applied research shows [Cosa, Reu95, Sar96], there is a commercial and scientific interest in extended transaction support. Meanwhile, [Alo96, Bar95, Chr94] state that actual implementations of such support are hard to find. In WIDE, a combination of modified existing transaction models is used to cope with the transactional requirements posed by complex processes [Gre97]. The higher level processes are mapped onto a Saga-like transaction model providing relaxed transactional semantics [Gre98]. The lower level processes, i.e. business transactions and the processes below them, are mapped onto the nested transaction model, providing atomicity of business transactions, full isolation between business transactions and relaxed isolation within them.

This paper focuses on transaction support for the lower level processes. We propose to build nested transaction support on top of existing standard commercial DBMS's. Taking commercial DBMS's as a basis will facilitate introduction of the nested transaction support in business environments. Main focus of this paper is a highly portable and flexible architecture for the implementation of a nested transaction support server. To obtain portability and flexibility, the architecture is composed of three independent layers. The upper layer of the architecture translates process events from the client application to nested transaction primitives. The middle layer maps these nested transaction primitives to abstract, DBMS-independent flat transaction primitives. The lower layer maps the abstract flat transaction primitives on DBMS-specific transaction primitives in a multi-DBMS environment. Because of the simple event based communication between the layers a highly modular architecture is obtained resulting in high portability and flexibility. The lower layer, for instance, can be adapted individually to various DBMS's. Various nested transaction models can be handled by adapting the middle layer. The architecture has been realized in a next generation workflow system, as part of the WIDE Esprit project.

In section 2 we give an overview of related work in the area of nested transaction support. Section 3 describes the adopted nested transaction model as well as the mapping process from nested transactions to flat transactions. Section 4 gives an architecture for nested transaction support, an implementation of which is discussed in section 5. Conclusions and directions for further work are given in section 6.

2 Related Work

A model for nested transactions is first introduced in [Mos85]. It is proposed to overcome the limitations of the traditional flat transaction model in dealing with complex processes. Different variations on the model can be found in [Elm92]. We aim at the implementation of nested transaction support rather than introducing another nested

transaction model. The work in [Chr94] describes how ACTA is used as a tool for the synthesis of extended transaction models. Although the framework could be used to show the correctness of a particular implementation, it does not discuss the implementation itself nor does it give an architecture for it. In the approach given in [Kan95], each leaf transaction of a nested transaction tree is mapped on a separate flat transaction. All changes made by a subtransaction become visible to the parent transaction and its children upon the subtransaction's commit. Not before then, the updates are made durable. One sibling can use the results of the other sibling only after the latter has committed. This approach prevents data sharing between concurrent subtransactions. In our view however, data sharing between concurrent subtransactions within a business transaction is required, as stated in the previous section. [Day92] gives another approach to map long running activities onto transactions. They also break down the process in transactions, comparable to business transactions. Main difference is their use of rules and triggers to express and manage control flows. An extended nested transaction model is used to govern the execution of these rules rather than giving nested transaction support to complex processes, as we do.

In [Alo96] it is shown how several transaction models can be implemented using a Workflow Management System (WFMS). They propose to implement extended transaction support as part of the workflow application. In WIDE however, we aim at extended transaction support orthogonal to workflow management functionality by building a separate module that provides extended transaction support. An example of a commercial WFMS is COSA [Cosa]. In COSA the concept of transaction is included, but contrary to our approach, only flat traditional transactions are supported. Thus COSA does not offer extended transaction semantics.

In the Reflexive Transaction Framework [Bar95] transaction support is built on top of a Transaction Processing (TP) monitor. In our approach, the transaction support is built directly on top of a commercial DBMS. This seems to be more attractive for practical use because it does not require a TP monitor to be configured. Furthermore, only a small part of our architecture is DBMS-specific. This makes it more easily portable to other DBMS environments than architectures conforming to the Reflexive Transaction Framework.

3 Nested Transaction Support

In section 1 we have given the requirements posed on transaction management to cope with nesting in complex processes. This section specifies the nested transaction support that fulfills the requirements of section 1. Section 3.1 gives a description of our nested transaction model. Section 3.2 describes how complex processes are mapped onto nested transactions. Section 3.3 discusses the mapping from nested transactions to flat, DBMS-independent transactions. Section 3.4 presents the last mapping step, describing the mapping from DBMS-independent flat transactions to DBMS-specific flat transactions.

3.1 Nested Transaction Model

The nested transaction model has been proposed in [Mos85] to overcome the limitations of the flat transaction model in dealing with complex processes. Nested transac-

tions allow for safe concurrency within transactions and offer a modular approach to realize both intra-transaction and inter-transaction parallelism. Furthermore, nested transactions allow for a finer grained recovery mechanism than flat transactions, localizing the effects of failures and faults (see [Elm92]).

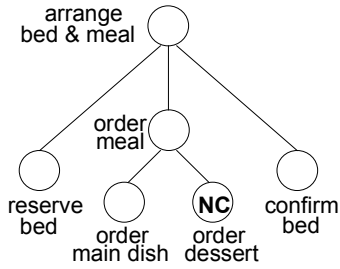

Figure 1: Example nested transaction

Our nested transaction model is based on the nested transaction model given in [Mos85] and allows for hierarchical decomposition of transactions, flexible notion of atomicity and flexible notion of isolation. We use the concepts of *critical* and *non-critical* transactions (called *vital* and *non-vital* in [Elm92]) to localize the effect of failures. Nested transactions are strictly isolated from each other. The isolation mechanisms we adopt in our model to provide safe concurrency *within* a nested transaction are simpler than those proposed in [Mos85]. We allow all subtransactions within a nested transaction to share data concurrently, while models described in [Mos85, Elm92, Kan95] allow datasharing between siblings only. For reasons of brevity, we will not discuss details of the isolation aspects of our model.

Figure 1 gives an example of a nested transaction that is part of a real-life workflow application in a hospital. The top level transaction ‘arrange bed & meal’ is decomposed into subtransactions ‘order meal’, ‘reserve bed’ and ‘confirm bed’. Subtransaction ‘order meal’ is in turn decomposed into subtransactions ‘order main dish’ and ‘order dessert’. The leaves of the tree (e.g. ‘order main dish’) represent transactions that are no further decomposed. Leaf transactions are strictly atomic.

When a subtransaction completes successfully, it is said to have committed, even though it is not a top-level transaction. Such commitment is relative: any updates become permanent only if all subtransactions containing the committed subtransaction also commit, and the enclosing top-level transaction completes. Thus, top-level transactions are special: they are the only irrevocable transactions.

Flexible atomicity is acquired by allowing a transaction to commit although not all its subtransactions have completed successfully. This flexible atomicity is specified by labeling each subtransaction as being either critical or non-critical. A subtransaction is labeled critical when its successful ending is a necessary (but not sufficient) condition for the successful ending of its parent transaction. A subtransaction is labeled non-critical when its success does not affect the success of its parent. A transaction may commit even when its non-critical children have not ended successfully. Thus, the atomicity of the transaction is relaxed.

In Figure 1, all subtransactions are critical except those marked ‘NC’ (non-critical). The subtransaction ‘order meal’ cannot commit when its subtransaction ‘order main dish’, which is critical, has not ended successfully. But ‘order meal’ can commit even when its subtransaction ‘order dessert’, which is non-critical, has ended unsuccessfully. This makes ‘order meal’ non-atomic.

3.2 Mapping Processes onto Nested Transactions

The mapping step from processes onto nested transactions is rather straightforward. Because of their process characteristics, business transactions are mapped onto top level nested transactions. This yields the required strict isolation between business transactions. Processes below the business transaction level are mapped onto subtransactions. This gives the required relaxed isolation between subprocesses of the same business transaction. Critical subprocesses are mapped onto critical subtransactions and non-critical subprocesses are mapped onto non-critical subtransactions.

Each time a new business transaction is started, a new top level transaction is created. Likewise, each time a subprocess of a business transaction starts, a new subtransaction is started. This subtransaction is part of the top level transaction on which the business transaction is mapped. The nested transaction (i.e. the top level transaction and all subtransactions) is committed only when the complete business transaction ends successfully.

3.3 Mapping from Nested Transaction Model to Flat Transaction Model

As stated in section 3.1, we adopt strict *inter*-nested transaction isolation while having relaxed *intra*-nested transaction isolation. The mapping from nested transactions to flat transactions has to preserve these isolation characteristics. To preserve inter-nested transaction isolation, different nested transactions are mapped onto different flat transactions. Accesses in different top-level nested transactions are mapped onto different flat ACID transactions. To preserve intra-nested transaction isolation, an entire nested transaction is mapped onto the same flat transaction. This means that each database access of a nested transaction is done within the same flat ACID transaction, allowing data sharing. A result of this is that accesses in concurrent subtransactions within the same nested transaction are executed in an interleaved fashion in the corresponding flat transaction. The flat transaction is committed if and only if the corresponding top level nested transaction is committed.

Relaxed atomicity is realized by using savepoints in the flat transaction model. Rolling back a subtransaction in the nested transaction model corresponds to rolling back to a savepoint in the flat transaction model. Because of the above mentioned interleaving of accesses in the flat transaction, rolling back one subtransaction in the nested transaction model might require other subtransactions to be rolled back as well. An algorithm has been developed that determines the minimum amount of subtransactions to be rolled back.

Any time a non-critical subtransaction starts, a savepoint in the flat transaction is set. When a failure occurs in a *non-critical* subtransaction, the flat transaction is at least rolled back to that savepoint. The flat transaction has to be rolled back even further when other subtransactions performed accesses concurrently with the failed subtransaction. When a failure occurs in a *critical* subtransaction, its parent should be rolled back. If the parent is critical as well, the parent's parent should be rolled back in turn. As a result of the failure, either a non-critical ancestor of the failing transaction is rolled back or the whole nested transaction is rolled back. In the flat transaction model this corresponds with either aborting the whole flat transaction or rolling back to the

savepoint coinciding with the start of the non-critical ancestor. Again, concurrency aspects have to be taken into account.

Figure 2 shows the progress in time of the nested transaction of Figure 1 up to the point where subtransaction 'order dessert' fails. Figure 2 also shows the mapping of the nested transaction onto a flat transaction. Subtransactions 'reserve bed' and 'order main dish' have ended. Subtransaction 'confirm bed' is still active and is concurrent with 'order dessert'. Because 'order dessert' is non-critical, a savepoint has been set at its start. At failure of 'order dessert' the flat transaction is rolled back to this savepoint. Subtransaction 'confirm bed' however, has done database accesses that are interleaved with those of 'order dessert'. Therefore 'confirm bed' should be aborted as well. Because 'confirm bed' is critical, its parent 'order meal' is aborted as well. Subtransaction 'order meal' is a critical child of the top level transaction 'arrange bed & meal'. Thus, the whole nested transaction is canceled and the flat transaction is aborted.

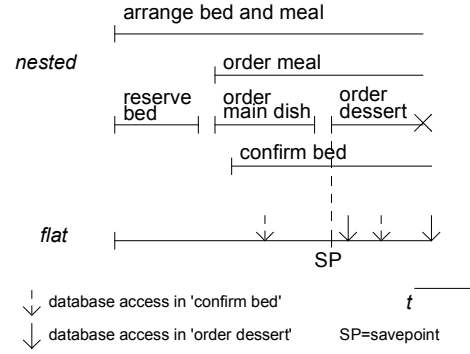

Figure 2: Example of mapping a nested transaction onto a flat transaction

3.4 Mapping from Flat Transaction Model to Database Specific Transactions

The final step in the mapping process involves mapping of abstract, database independent transaction operations to database specific operations. Operations like starting a new flat transaction, setting savepoints, and rolling back to savepoints are mapped onto the functions that the DBMS offers. Table 1 gives examples of DBMS-specific operations, in this case those of the Oracle DBMS. The left column contains abstract flat transaction operations that are mapped on the operations of the Oracle OCI library [Ora96] listed in the right column. Note that this mapping is not a one-to-one mapping. There are no OCI primitives for setting savepoints or for rolling back to savepoints. In those cases SQL-statements are used, parsed and executed by the OCI primitives OPARSE and OEXEC.

Abstract operations	Oracle OCI functions
start transaction	OOPEN
commit	OCOM, OCLOSE
abort	OROL
rollback to savepoint	OPARSE, OEXEC
set savepoint	OPARSE, OEXEC

Table 1: Abstract and Oracle specific flat transaction operations

4 Architecture

This section describes the architecture of a software component that offers the nested transaction support as specified in section 3. It maps nested processes via a nested transaction model onto DBMS-specific flat transactions. The Nested Transaction Support component acts like a server and can handle multiple clients simultaneously, and per client multiple transactions. Figure 3 shows the communication between the Nested Transaction Support (NTS) server and its client, a Process Engine (PE). The PE notifies the NTS of process events like the start of a business transaction or the canceling of a subprocess. The NTS provides the PE with status information on the nested transaction (e.g. the list of subtransactions that have been rolled back due to the canceling of a subprocess). In case a new business transaction or subprocess is started, the NTS returns the identifier of the transaction on which it has mapped the new business transaction or subprocess. This identifier is used by the PE when performing database accesses. These accesses are performed outside the NTS. When the PE wants to access the database during a subprocess, it requests a physical database context from the NTS, providing the NTS with the transaction identifier on which the subprocess is mapped. The NTS returns the physical database context in which the PE should perform the database access.

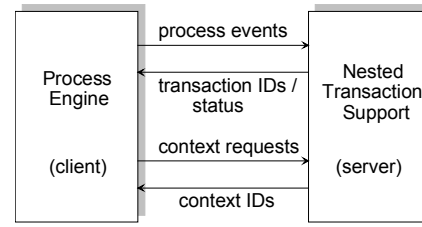

Figure 3: Communication between NTS server and its client

The architecture of the NTS consists of three layers, each performing one of the mapping steps. The interfaces of the layers are kept simple by using event based communication, resulting in a modular architecture of independent layers, which makes the architecture portable and flexible. Portability allows the architecture to be implemented in different DBMS environments with little adaptation, while flexibility allows it to be easily adapted to various nested transaction models. Modularity of the architecture allows the layers to be used separately in various applications. The lower layer, for instance, can be used independently in environments with flat transactions only. Another example is the use of the lower two layers separately from the top layer in situations where there is at most one business transaction active at a time.

Figure 4 gives a global overview of the architecture. The sections below each describe one of the architecture's layers.

4.1 Nested Transaction Manager

The Nested Transaction Manager (NTM) performs the first mapping step as described in section 3.2. It acts as a nested transaction support server and can handle multiple clients simultaneously. It receives notifications of process events (like 'start subprocess' and 'cancel subprocess') from clients and maps these events to nested transaction operations (like 'start subtransaction' and 'cancel subtransaction').

The NTM takes care of passing each nested transaction operation to the transaction object that represents the corresponding nested transaction in the Nested Transaction

Object layer. The NTM manages those objects: it creates and removes top-level transaction objects when needed by a process (see section 4.2).

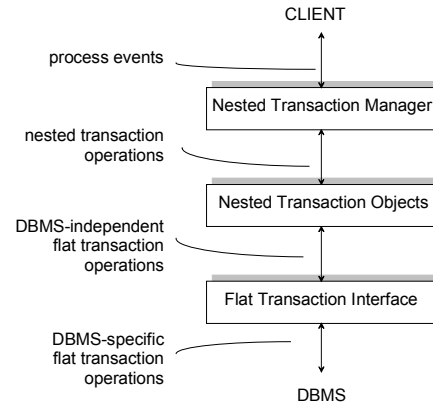

Figure 4: Internal architecture of NTS

4.2 Nested Transaction Objects

The middle layer of the architecture consists of Nested Transaction Objects (NTO's). There are two kinds of objects: each Top-level Transaction (TT) object represents a top-level transaction, while each Subtransaction Object (ST) represents a subtransaction. Figure 5 shows the internal architecture of this layer, with a hierarchical arrangement of the objects.

Top-level transaction objects are responsible for the mapping from the nested transaction model to the abstract flat transaction model. They use the ST objects to keep track of (1) the hierarchical decomposition of the top-level nested transaction and (2) concurrency between subtransactions within the nested transaction.

There is no centralized administration of this decomposition or concurrency: this knowledge is distributed among the objects. Each object applies the same algorithms on the part of the nested transaction's state that it represents. Each object takes care of creating and deleting its own children. The TT objects are created and deleted by the NTO-layer's client (the Nested Transaction Manager).

A nested transaction operation enters the NTO-layer at a TT object. This object passes the message to each of its children, which in turn pass it further to their children until it reaches the proper object. This object takes care of handling the operation, by performing an action like creating or deleting a child or by changing its status (e.g. to 'ended' or 'canceled'). Objects update their own administration of hierarchical structure and concurrency. Decisions concerning the canceling of subtransactions are taken locally by the objects corresponding to the subtransactions. The notification of the action or decision is sent back up the tree until it reaches the TT object. Based on the notifications it gets from its children, the TT object maps the operation to the proper flat transaction operation.

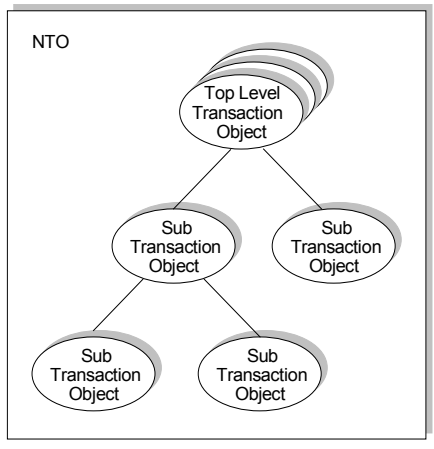

Figure 5: Objects in the NTO layer

As an example we take the nested transaction operation 'abort subtransaction'. This operation is sent by the NTM layer to the TT object containing the subtransaction. The TT object

sends the abort message down until it reaches the ST object representing the subtransaction to be canceled. This ST object changes its status to 'canceled' and informs its parent of the cancellation. The parent decides whether it should be canceled as well, which depends on whether the child is critical or not. The parent, in turn, notifies its own parent of the decision, until the decision reaches the TT object. Other ST objects representing subtransactions that are concurrent with the one aborted, might decide to abort as well. These decisions are also passed up to the TT object. Based on all notifications coming from down the hierarchy, the TT object maps the 'abort subtransaction' operation either on the flat transaction operation 'rollback to savepoint' or on the flat transaction operation 'abort flat transaction'. The latter one is chosen when one of the critical children (if any) of the TT object has decided to abort itself.

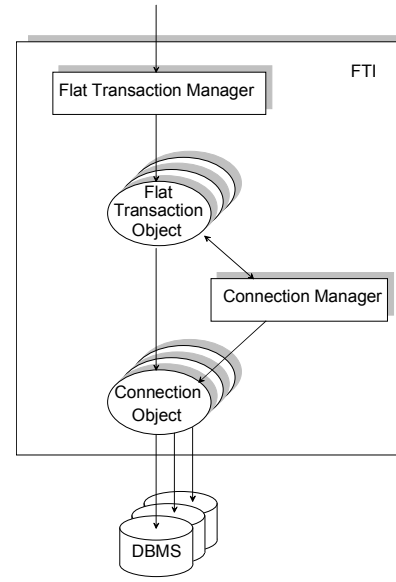

Figure 6: Internal architecture of the FTI

4.3 Flat Transaction Interface

The Flat Transaction Interface (FTI), which forms the lower layer of the architecture, is responsible for the third mapping step (see section 3.4). It translates abstract flat transaction primitives to database specific flat transaction operations. This layer thus forms the interface to the DBMS for the other two, DBMS-independent layers, shielding them from DBMS-specific issues. By adapting the FTI, the nested transaction support can be ported to different DBMS's. Besides the mapping process, the FTI also takes care of handling the client-server connections with the various DBMS's. Figure 6 shows the internal architecture of the Flat Transaction Interface.

The Flat Transaction Manager handles the creation and removal of Flat Transaction Objects. Each time it is notified (by a Nested Transaction Object) of the start of a new flat transaction, it creates a new flat transaction object. Likewise, when it is notified of the end of a flat transaction (either by commit or cancel) it removes the corresponding Flat Transaction object. Besides, the Flat Transaction Manager is responsible for dispatching the abstract flat transaction primitives to the proper Flat Transaction Objects.

A Flat Transaction Object represents one abstract, DBMS-independent, flat transaction. In practical applications, there is sometimes the need to perform accesses on different DBMS's within the same business transaction. As each business transaction is mapped on one flat abstract transaction (via a nested transaction), this abstract flat transaction should be mapped on *multiple* physical transactions. The Flat Transaction Object takes care of this mapping. It ensures that all physical flat transactions belonging to the same abstract flat transaction are committed, aborted or partially rolled back

simultaneously. Flat transactions that are mapped onto more than one physical transaction are pseudo-atomic however. Imagine a flat transaction being mapped onto two different physical transactions T1 and T2. When the flat transaction commits, the Flat Transaction Object signals T1 and T2 to commit. T1 might commit successfully while committing T2 might fail because of a DBMS failure, leaving the flat transaction only partially committed.

The Connection Manager (CM) provides its clients (the Flat Transaction Objects) with connections to the DBMS. When a Flat Transaction Object asks for a new physical transaction, the CM provides a free connection that is not in use by a Flat Transaction object. If there is no free connection, the CM opens a new one by creating a new Connection Object representing a client-server connection with the DBMS. The Connection manager returns a reference to this Connection Object.

The X/A protocol [XOp96] allows multiple concurrent physical transactions to use the same connection, but is not part of standard DBMS functionality. Since our goal is to build transaction support on standard DBMS functionality, we assume that a connection can be used by at most one physical transaction at a time. The actual opening and closing of connections is done by the Connection Objects, as they contain the DBMS-specific knowledge required to do so. Each Connection Object represents a client-server connection to a DBMS. Connection Objects contain DBMS-specific methods for opening and closing connections. They also perform the mapping from DBMS-independent transaction primitives to DBMS-specific transaction operations. Thus, all DBMS-specific knowledge is kept locally in the Connection Objects. For each DBMS there must be defined a Connection Object class, from which the Connection Manager can instantiate Connection Objects. The CM picks the class depending on the DBMS to which the CM's client requires a connection.

5 Implementation

The architecture specified in section 4 has been implemented in the WIDE Esprit project [Cas96a, Cas96b, Cer97, Gre97]. Goal of WIDE (Workflow on Intelligent Distributed database Environment) is to develop extended database technology as the basis for a commercial next-generation workflow management system. In WIDE, extending database technology focuses on extended transaction management and active rule support.

Figure 7 shows how the nested transaction support architecture is embedded in the overall WIDE architecture. In WIDE, the nested transaction support is called

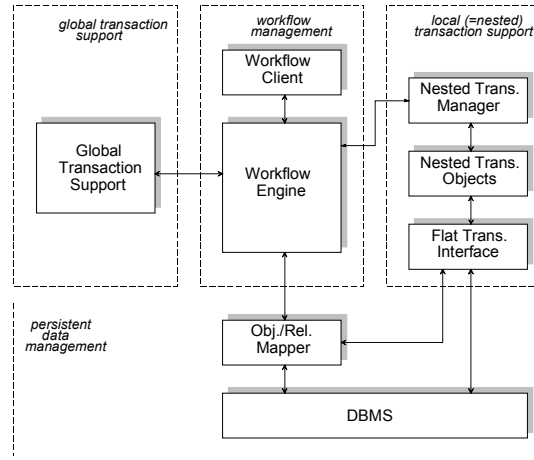

Figure 7: WIDE transaction support architecture

'Local Transaction Support'. The Saga based transaction support for the higher level processes (see section 1) is called Global Transaction Support. The terms Global and Local do not refer to distribution aspects but rather to being below or above the business transaction level. As can be observed from the architecture, the two modules of transaction support are completely orthogonal. For a detailed description of the WIDE architecture, see [Cer97].

The Workflow Engine (WFE) is the core of the Workflow Management System. It takes care of scheduling and assigning processes. It provides the Nested Transaction Manager with the process events of the processes on and below the level of business transactions. The Nested Transaction Support performs the mapping as described in section 3. It provides the WFE with transaction identifiers, that are used by the WFE to access the DBMS. Database accesses go through the Object/Relational Mapper. This Mapper provides an object-oriented database access interface to its clients and maps this to the relational interface of the DBMS. It requests the physical database context in which it performs the accesses from the FTI layer, providing it with the transaction identifier. In case a subprocess is canceled by the Workflow Client, the WFE notifies the Nested Transaction Manager (NTM) accordingly. The NTM provides the WFE with the IDs of the subprocesses that have been canceled by the Nested Transaction Support. The WFE can then adapt its scheduling and can notify its clients. The architecture allows multiple clients per NTM (see section 4.1). For reasons of simplicity, however, there is a separate NTM per WFE in the current implementation. Each NTM is capable of handling multiple transactions simultaneously though.

The WIDE WFMS has been built on top of the Oracle v.7.3 DBMS. Connection Objects in the Flat Transaction Interface take care of mapping abstract flat transaction primitives to Oracle specific function calls from the Oracle Call Interface (OCI) library [Ora96]. In the current implementation the Flat Transaction objects map each abstract flat transaction on one physical transaction only.

6 Conclusions

In this paper, we have presented an architecture for the implementation of nested transaction support. As shown by current research and commercial developments, implementation of nested transaction support is valued important. This is also confirmed by the results of the prototype tests within WIDE.

Although illustrated by workflow management, the use of the architecture presented is not limited to this purpose. Thanks to its modularity, the architecture can be easily used in other areas in which complex processes are important, like CSCW and Process Control. Furthermore, the architecture is easily adaptable to different DBMS environments. The architecture is implemented as part of the next generation workflow management system FORO during the WIDE Esprit project, with Oracle as the underlying commercial DBMS. The end users in the WIDE consortium used and tested the implementation in their applications. They value Nested Transaction Support useful since it provides them with strong transactional primitives which they can use in the modeling of complex processes.

We specified isolation concepts within nested transactions on the level of workflow data objects. Further work will be aimed on implementing this intra-nested transaction

isolation. Other future work will focus on the formalization of the algorithms used in nested transaction support. As some of these algorithms take possibly critical decisions about abortion of processes, formalization is desired to show the correctness of the behavior of the Nested Transaction Support to advanced application designers.

References

[Alo96] G. Alonso, D. Agrawal et al.; *Advanced Transaction Models in Workflow Contexts*; Procs. Int. Conf. on Data Engineering, 1996.

[Bar95] R. Barga & C. Pu; *A Practical and Modular Method to Implement Extended Transaction Models*; Procs. 21st Int. Conf. on Very Large Data Bases; Zürich, Switzerland, 1995.

[Cas96a] F. Casati, S. Ceri, B. Pernici, G. Pozzi; *Deriving Active Rules for Workflow Enactment*; Procs. 7th Int. Conf. on Database and Expert Systems Applications; Zürich, Switzerland, 1996.

[Cas96b] F. Casati, P. Grefen, B. Pernici, G. Pozzi, G. Sánchez; *WIDE: Workflow Model and Architecture*; CTIT Technical Report 96-19; University of Twente, 1996.

[Cer97] S. Ceri, P. Grefen, G. Sánchez; *WIDE - A Distributed Architecture for Workflow Management*; Procs. 7th Int. Workshop on Research Issues in Data Engineering (RIDE); Birmingham, UK, 1997; pp. 76-79.

[Chr94] P.K. Chrysanthis, K. Ramamritham; *Synthesis of Extended Transaction Models using ACTA*; ACM Transactions on Database Systems, vol. 19(3), 1994, pp. 450-491.

[Cosa] COSA Solutions Gmbh; *Cosa Solutions: Turning Workflow into Cashflow;* http://www.cosa.de.

[Day92] U. Dayal, M. Hsu, R. Ladin; *A Transactional Model for Long-Running Activities*; Procs. 17th Int. Conf. on Very Large Databases, 1991.

[Elm92] A. Elmagarmid, ed.; *Database Transaction Models for Advanced Applications*; Morgan Kaufmann; USA, 1992.

[Gre97] P. Grefen, J. Vonk, E. Boertjes, P. Apers; *Two-Layer Transaction Management for Workflow Management Applications*; Procs. 8th Int. Conf. on Database and Expert Systems Applications; Toulouse, France, 1997.

[Gre98] P. Grefen, J. Vonk, E. Boertjes, P. Apers; *Global Transaction Support: Formal Specification and Practical Application*; CTIT Technical Report 98-10; University of Twente, 1998.

[Kan95] I.E. Kang & T.F. Keefe; *Reliable Nested Transaction Processing for Multidatabase Systems*; Integrated Computer-Aided Engineering, vol. 2(1), 1995, pp. 49-67.

[Mos85] J. E. B. Moss; *Nested Transactions: An Approach to Reliable Distributed Computing*; The MIT Press, Cambridge, Massachusetts, 1985.

[Ora96] Oracle Corporation; *Programmer's Guide to the Oracle Call Interface, release 7.3*; Oracle Corporation, 1996.

[Reu95] A. Reuter, F. Schwenkreis; *ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management Systems*; IEEE Data Engineering Bulletin; vol. 18(1), 1995, pp. 4-10.

[Sar96] S.K. Sarin; *Workflow and Data Management in InConcert*; Procs. 12th Int. Conf. on Data Engineering; New Orleans, USA, 1996.

[XOp96] X/Open Company; X/Open Guide: Dis*tributed Transaction Processing: Reference Model, version 3*; X/Open Company, 1996.