ReflecTS; A Flexible Transaction Service Framework

Anna-Brith Arntsen
Computer Science Department
University of Tromsoe
9037 Tromsoe, Norway
annab@cs.uit.no

Randi Karlsen
Computer Science Department
University of Tromsoe
9037 Tromsoe, Norway
randi@cs.uit.no

ABSTRACT

Transactional requirements, from applications and execution environments, are varying and may exceed traditional ACID properties. We believe that transactional middleware platforms must be flexible in order to adapt to these requirements. Present systems are, however, inflexible with respect to such adaptations. `ReflecTS` is a flexible transaction processing platform maintaining an extensible number of concurrently running transaction services. This paper presents the architecture and the first prototype of `ReflecTS`, which focuses on a transaction service selection procedure. The selection procedure is based on XML-specifications of transactional requirements and transaction service descriptions - making the platform adjustable to varying requirements.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; H.2.4 [**Database Management**]: Systems—*transaction processing*

General Terms

Design, management, reliability

Keywords

Flexible transaction processing, middleware, reflection, component technology, requirement specifications

1. INTRODUCTION

Today's application domains and execution environments have requirements to transaction execution that are varying and exceed traditional ACID properties. Applications are also under constant evolution, which mean that new requirements can arise. Varying and beyond-ACID transactional requirements demand for a flexible transaction execution environment. These requirements are not met by current transactional middleware solutions where merely ACID transactions are supported.

Application domains, including workflow, cooperative work, medical information systems and e-commerce, all possess varying requirements. The travel arrangement scenario is a well-known example of a long-running beyond-ACID transaction. A user requests a hotel room (T1), a flight (T2), a car (T3), a restaurant table (T4), and a theater ticket (T5). He may also want to specify alternative hotels and restaurants. Not all reservations are equally important, and a restaurant table may for instance be omitted from the transaction. This transaction must be structured as a beyond-ACID transaction where intermediate results of individual tasks (subtransactions T1-T5) can be committed and revealed when finished. Commit of partial results requires compensating transactions to be specified and executed in case of rollback [11].

Another example is a medical information system where patient journals, radiographs and spoken reports are stored over a number of heterogeneous sites. Medical personnel handling patient information may have either ACID or beyond-ACID requirements to transactions. For instance, when in an emergency, the response time is important. This imposes a time constraint. Further, mobility is a central environmental challenge. Medical personnel on a place of loss are most likely equipped with PDA, sensor- and recording devices wanting to transfer and receive real-time information with quality of service guarantees. These examples show that transactional requirements may vary between applications and between users within the same application.

A number of advanced transaction models [10, 23, 20, 11] have been proposed to meet different requirements. They address specific transactional requirements, such as relaxed atomicity and isolation, offering some flexibility. Many of these models where suggested with specific applications in mind, with fixed semantics and correctness criteria. Consequently, they fail to provide sufficient support for wide areas of applications. It is unrealistic to believe that the "one-size fits all" paradigm will suffice and that a single approach to extended transactions will suit all applications. The counterpart to the "one-size fits all" paradigm, which is the main motivation behind our work, is to some extend recognized within the web-services domain where for instance the WS-transaction [15] specification describes a solution providing two different transaction services.

In this paper we present the architecture and the current prototype of `ReflecTS`, a highly adaptable and flexible

RM '05, November 28- December 2, 2005 Grenoble, France

transaction processing platform. `ReflecTS` serves varying requirements by offering a number of *concurrently running transaction services* where each service provides different transactional guarantees. Hence, the fundamental mechanisms and the main contribution of `ReflecTS` is to adapt to varying requirements by configuring and selecting transaction services according to the needs of the applications.

We build `ReflecTS` using component technology throughout the whole platform. To achieve configurability and reconfigurability of transaction services, our approach is based on the reflective building blocks *OpenCOM components* [8] and *ReMMoC component frameworks* [12]. Further, a transaction service selection procedure is based on XML specifications of transactional requirements and transaction services.

In the remainder of this paper we first, in section 2, give necessary background information on OpenCOM components and ReMMoC component frameworks. Then, the `ReflecTS` architecture and prototype is presented in section 3. Section 4 follows with related work and section 5 draws concluding remarks and gives directions for future work.

2. BACKGROUND

Reflection, components and component frameworks are means to achieve flexible and configurable middleware systems today. In this section, we briefly present the building blocks of `ReflecTS`: the OpenCOM component model and ReMMoC component frameworks.

Reflection and Middleware Construction

Reflection is a technique applied for "opening up" a system to support inspection and adaptation of internal structure and behaviour [24][19]. Meta-interfaces provide operations to inspect the internal details of the platform (introspection) and to change the underlying middleware (adaptation). Two styles of reflection can be considered [4]: *Structural* reflection to inspect and change the underlying structure of the system, and *behavioral* reflection to inspect and change the activity in the underlying system.

Reflective middleware platforms utilize the concepts of open implementation and reflection to get access to the underlying virtual machine. A number of such platforms have been developed, including OpenORB [5] and ReMMoC [12], which are build of reflective OpenCOM components and component frameworks.

OpenCOM Components

According to Szyperski [26], a component can be viewed as: *"a unit of composition with contractually specified interfaces and explicit context dependencies, and in this context, a component can be deployed independently and is subject to third-party composition"*.

Components implement strong interfaces and encapsulate implementation details. Hence, components can be added, removed or replaced, making component-based middleware systems adaptable. One of the many available component models is OpenCOM [8], a lightweight, efficient and reflective component model built upon the core concepts of COM [6]. OpenCOM was designed specifically for the implementation of OpenORB [5] and is considered a good model for this purpose.

The fundamental concepts of OpenCOM are interfaces, receptacle and connections (bindings between interfaces and receptacles). An interface represents a unit of service provision and a receptacle a unit of service requirement. The base interfaces of an OpenCOM component are IMetaInterface, ILifeCylce and IConnections. Reflective features are added to OpenCOM through meta-interfaces for introspection and adaptation: `IMetaInterception`, `IMetaArchitecture` and `IMetaInterface`. A standard runtime substrate maintains a system graph of components and manages the creation and deletion of components.

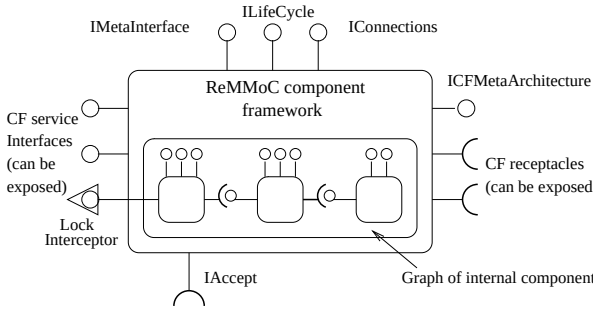

Figure 1: ReMMoC Component Framework

ReMMoC Component Frameworks

Component frameworks (CF) are defined by Szyperski [26] as: *"a collection of rules and interfaces that govern the interaction of a set of components plugged into them"*. A CF enforces architectural principles on the components it supports. This is especially important in reflective architectures with dynamic changes that must be verified.

Both OpenORB and ReMMoC define a component framework model which is constrained by the choice of OpenCOM as the component model. We choose ReMMoC CF as the base CF for ReflecTS as it, contrary to OpenORB CF, has added domain-specific features for maintaining and controlling changes to the component graph. Changes are restricted to equal a valid component configuration. These valid configurations are reachable through the IAccept interface. ReMMoC CF also provides a method for assuring that all changes to existing configurations are made at appropriate times. This is done by implementing a standard readers/writers lock to access the local CF graph. The ReMMoC CF model is pictured in figure 1.

A ReMMoC CF is itself an OpenCOM component maintaining a graph of internal structure. Each ReMMoC CF implements at least the base interfaces of an OpenCOM component (IMetaInterface, ILifeCycle, IConnections). To inspect its internal structure, a CF also implements the IMetaArchitecture interface. Further, to adapt the internal structure of its component configuration, each CF implements the CF-specific interface `ICFMetaArchitecture` [12], with provided operations for inspection and adaptation.

3. REFLECTS

3.1 Introduction

As previously noticed, there is a big gap between offered and required support for varying transactional requirements. `ReflecTS` has been designed to close this gap and offers flexible transaction processing by providing an extensible number transaction services. The main features of `ReflecTS` are as follows:

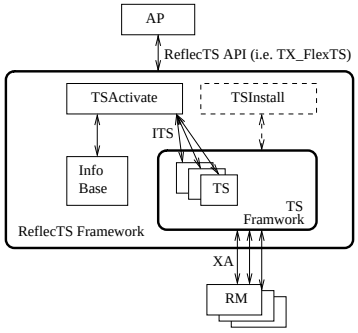

Figure 2: Overview of ReflecTS

1. *Configuration* and *reconfiguration* of transaction services
2. Deployment of an extensible number of *transaction services* supporting dissimilar transactional requirements
3. Selection of transaction services based on formal specifications of requirements and services
4. Management of concurrently running transaction services

The properties of `ReflecTS` has been influenced by the TSenvironment [18, 17], which is a model for an adaptable transactional system. Within TSenvironment, transaction services can be assembled, deployed, modified and concurrently used according to the needs of the applications. `ReflecTS` is designed for managing global transactions in a distributed environment. A global transaction is typically composed of a number of subtransactions, which are executed and controlled by underlying systems.

In the rest of this section, we first outline the architecture of ReflecTS and then present the current prototype of ReflecTS.

3.2 ReflecTS Architecture

`ReflecTS` is hierarchically composed of two component frameworks; `ReflecTS Framework` and `TS Framework`. These frameworks host a number of components and transaction service implementations, expose interfaces and provide policies for constraining changes. An overview of the ReflecTS platform is given in figure 2.

3.2.1 Overview

The `ReflecTS API` defines the interaction between application programs and `ReflecTS`. For transaction execution purposes, an interface contains methods for controlling transactions. An example of such an interface is `TX_FlexTS`, which is presented in the next section. While global transaction management (including global commit/recovery and global concurrency control) is performed by the transaction services within `TSFramework`, local transaction control is performed by transactional mechanisms in underlying resource managers (RM) where the actual operations are carried out.

Transaction services are offered via the `ITS` interface. In figure 2, several arrows between TSActivate and the deployed transactions services illustrates concurrently active services. Likewise, several arrows between ReflecTS and resource managers (RMs) illustrates that numerous RMs can be involved in one or more transactions.

The ReflecTS-RM interaction is determined by the X/Open XA-interface [13] when the RM's are XA-compliant. This interface allows `ReflecTS` to structure the work of RMs into global transactions and coordinate global completion or recovery.

3.2.2 TS Framework

`TS Framework` is configured by plugging in an extensible number of different transaction service (TS) implementations, with possibly different transactional guarantees. Thus, `ReflecTS` provide the transactional properties, $\mathcal{P} = P_1, \ldots, P_n$, where P_i represents the properties of transaction service TS_i.

Transaction services are either active or inactive. An *active service* is currently in use by at least one transaction, while an *inactive service* is not. Two services, TS_i and TS_j, are *incompatible* if the transactional properties of at least one of them cannot be guaranteed. It is the responsibility of `ReflecTS` to manage TS's such that incompatibility do not interfere with TS correctness. Concurrently active and *compatible* services do not interfere.

Configurations and reconfigurations of transaction services conforms to valid services and architectures, controlled by `TS_Framework` and according to the features of ReMMoC component framework.

3.2.3 ReflecTS Framework

The components of `ReflecTS Framework` perform tasks such as transaction service configuration, reconfiguration, selection and activation.

`TSActivate` receives requests for transaction execution, performs transaction service selection and evaluates compatibility. Hence, `TSActivate` enables behavioral reflection. Service selection is based on XML specifications of transactional requirements and deployed transaction services. Prior to completing service selection, *vertical* and *horizontal* compatibility is determined. *Vertical compatibility* is evaluated between the transactional mechanisms (commit, recovery, concurrency) of the elected TS and those of the involved resource managers. Before activating the selected TS, *horizontal compatibility* is determined. Horizontal compatibility is defined between concurrently active services and involves a thoroughly comparing of transactional mechanisms of each service. However, examining horizontal compatibility is only necessary if the transaction services are working towards the same resource manager and the same dataset.

`TSInstall` handles requests for transaction service configuration and reconfiguration, and assures that they conform to valid services. By exploring the features of the component framework model, `TSInstall` enable structural reflection.

The `InfoBase` component stores information about deployed transaction services; transaction service (TS) descriptors and compatibility information (*vertical* and *horizontal*) using XML. TS descriptors are provided `InfoBase` from `TSInstall`. Further, descriptors and compatibility information are encompassed in the transaction service selection and activation procedures carried out by `TSActivate`.

3.3 ReflecTS Prototype

This section describes the first prototype of `ReflecTS` focusing on behavioral reflection. The prototype implements `TSActivate` including transaction service selection and activation, and excludes `TSInstall`. We also omit discussing

vertical and horizontal compatibility. The platform is still configurable and reconfigurable as the feature for changing it - the `ICFMetaArchitecture` interface - is inherent in the component framework model.

To support transaction execution, `ReflecTS` prototype describes an interface and a specification. The interface, `TX_FlexTS`, represents the AP-ReflecTS interaction, and the `TX_RQ` specification represents a formal way to describe transactional requirements by using XML. The transaction service selection procedure performed by `TSActivate` receives transactional requirements via `TX_FlexTS` and gathers XML-specifications of deployed transaction services from `InfoBase`. Further, the selection procedure is executed based on these specifications, making `ReflecTS` adaptable to varying requirements.

3.3.1 Interfaces

`TX_FlexTS`, is the interface by which applications call `ReflecTS` to demarcate global transactions and to control the direction of their completion. The following table presents the IDL-specification of the `TX_FlexTS` interface.

interface IReflecTS:IUnknown
HRESULT Trans_Begin(char* XMLDOC)
HRESULT Trans_Commit()
HRESULT Trans_Rollback()
HRESULT Trans_Info()

The design of `TX_FlexTS` has been influenced by the TX-interface of X/Open DTP [14], but differ from it in several respects to conform to varying transactional requirements. First, the XML-description of transactional requirements is added to the *Trans_Begin()* request. Next, `TX_FlexTS` does not implement routines to open and close resource managers linked with the application. The TX-interface embraces these routines, which is an accomplished task when there is only one transaction service available. Within `ReflecTS` there might be several services available, and compatibility must be evaluated before service selection. The selected service is then responsible for opening and closing resource managers. Further, the TX-interface includes a routine for setting transaction timeout. This is not included in the `TX_FlexTS` interface, but is added as a requirement to the `TX_RQ` specification.

Every transaction service implements `ITS`. Literally, this interface is a projection of the `TX_FlexTS` interface. An exception is that there is no transferal of transactional requirements. Based on our current work, we believe that this interface is sufficient to explore the different properties of the different transaction services.

3.3.2 TX_RQ Specification

The `TX_RQ Specification` formally describes a variety of transactional requirements by using XML. Literally, `TX_RQ` specifies a range of different transaction models, which easily can be adapted to suit an arbitrary transaction model due to the extensibility of XML.

Properties can be refined into specialized properties. The following table lists a suggestion of transactional requirements, with requirements to the left and refinements the right, mostly focusing on the ACID properties. It is important to note that this is a preliminary suggestion that easily can be changed.

Requirement	*Degree of Requirement*
Atomicity	Strict, Relaxed, Alternative, Exception
Consistency	Strict
Isolation	Strict, Relaxed
Durability	Strict
Time	Strict, Repetitive

In this approach, atomicity can possess the following degrees: (1) Strict atomicity causes all subtransactions to complete their work before commit of global transaction. (2) Relaxed atomicity indicates that subtransactions can commit unilaterally before completion of the global transaction. This requires that subtransactions define associated compensating transactions. Parameter tags in the specification defines early commit of subtransactions and compensating transactions. (3) Alternative atomicity is introduced for atomicity specification in case alternative ways of execution are allowed and (4) exception atomicity is defined for situations when there are vital versus non-vital subtransactions.

The level of isolation follows implicitly by the level of atomicity. When atomicity is relaxed and partial results early committed, we would like to reveal those results by relaxing the isolation requirement. Controlling the use of partly committed results is part of the mechanism implemented by the transaction service. The combination of atomicity and isolation follow strict rules: relaxed isolation is only possible when atomicity is relaxed.

Consistency needs to be fully preserved to ensure the correctness of the results of a transaction execution. Correctness might however, be hard to define for a transaction with relaxed atomicity and isolation. With strict isolation, the database defines the correctness criteria. With relaxed isolation, the correctness criteria can be user-defined.

As for consistency, durability must be present to ensure that results of transactions are permanent. One question is when to save the results of committed parts of long-running activities. Saving partial results on the fly require commit-dependency and abort-dependency specifications. In addition, when isolation is relaxed, sharing may occur. For both situations, compensating activities must be specified.

The `TX_RQ Specification` is outlined below.

TX_RQ Specification

```
<Requirements>
   <Requirement>
     <Name>NameOfRequirement</Name>
     <Degree>DegreeOfRequirement</Degree>
     <ParameterList>
       <Parameter>
         <Name>NameOfParameter</Name>
         <Values>
           <Value>ParameterValue</Value>
         </Values>
         </Parameter>
     </ParameterList>
   </Requirement>
</Requirements>
```

The following exemplifies use of this specification. We describe two different transaction models: one traditional ACID transaction and one long-running beyond-ACID transaction as described in the travel arrangement scenario in section 1. This specification could naturally be fulfilled by an

implementation of for instance the Saga transaction model [11] or the DOM transaction model [10]. As an extension follows that, any transaction model can be expressed using this specification.

TX_RQ Specification of an ACID transaction

```
<Requirements>
   <Requirement>
     <Name>Atomicity</Name>
     <Degree>Strict</Degree>
   </Requirement>
</Requirements>
```

TX_RQ Specification of a beyond-ACID transaction

```
<Requirements>
   <Requirement>
     <Name>Atomicity</Name>
     <Degree>Relaxed</Degree>
     <Parameterlist>
       <Parameter>
         <Task>T1</Task>
         <Values>
           <Value>CT1</Value>
          "Compensating trans, CT for T2-T5"
         </Values>
       </Parameter>
     </Parameterlist>
   </Requirement>
   <Requirement>
     <Name>Atomicity</Name>
     <Degree>Alternative</Degree>
     <Parameterlist>
       <Parameter>
         <Task>T1</Task>
         <Values>
           <Value>AT11</Value>
           "More alternative trans, AT"
         </Values>
       </Parameter>
     </Parameterlist>
   </Requirement>
</Requirements>
```

3.3.3 Transaction Service (TS) Descriptors

TS Descriptors contains information about transaction processing mechanisms: commit, recovery and global concurrency control (GCC). Two-phase commit (2PC) or one of its extensions, for instance presumed nothing (PrN), presumed abort (PrA) or presumed commit (PrC) [2], fits as global commit of distributed transactions. The presence of GCC depends on the autonomy of the involved resource managers. The most common approaches to beyond-ACID transactions are for instance extensions of two-phase locking to altruistic locking or relaxation of serializability using semantic information about transactions. With full isolation, serializability is the only correctness criteria and strict locking the selected GCC scheme. The following description captures the essentials of a transaction service.

TS_Description

```
<TS_Descriptor>
   <ServiceID>TS_ID</ServiceID>
   <Properties>TS_Properties</Properties>
   <TaskList>
     <Task>
       <TaskId>TaskId</TaskId>
       <TaskParameters>
         <Parameter>ParameterID</Parameter>
         <Values>
             <Value>ParameterValue</Value>
         </Values>
       </TaskParameters>
     </Task>
   </TaskList>
```

With this approach, the *Properties* tag can possess one of two options: ACID or beyond-ACID. *TaskID* takes the values Commit or GCC, and the following *TaskParameter* tags with corresponding *TaskValues* are descriptions of the specified mechanisms.

3.3.4 TS Selection

The TS Selection procedure is performed by `TSActivate`. When selecting a transaction service, the requirements of the transaction and the specification of deployed transaction services are compared. The mapping of requirements to a service need not be one-to-one, which means that a specific set of requirements can be mapped to different TS's. The selection procedure first compares the requirement specification of TX_RQ with the TS_Properties specification of TS_Description. From our knowledge of the combination of atomicity and isolation follows that when the requirement specifications denotes *strict atomicity*, an atomic commit-protocol as for instance 2PC must be selected. The structure of the specifications indicates that not only a few tags must be extracted in order to select an appropriate service.

4. RELATED WORK

Existing transactional middleware platforms offer key infrastructures for distributed transaction processing, but with limited flexibility. Platforms like Microsoft Transaction Server (MTS) [9], Sun's Java Transaction Server (JTS) [25], and OMG's Object Transaction Service (OTS) [1], provide merely one transaction service with ACID guarantees. An exception from this is given by the CORBA Activity Service Framework [16], where various extended transaction models can be supported (although not concurrently) by providing a general-purpose event signaling mechanism. Another approach is the WS-transactions [15] specification where two transaction services are defined - one serving atomic ACID transactions, and one serving long-running activities.

Flexibility within transactional systems can be found in the works of Barga [3] and Wu [27]. Barga [3] describes a Reflective Transaction Framework implementing extended transaction models on top of TP-monitors. Wu [27] describes the use of Reflective Java to implement a flexible transaction service. Related work on dynamic combination and configuration of transactional and middleware systems can be found in Zarras [28], Ramampiaro [22], Prochazka [21] and Chrysanthis [7]. These works recognizes the diversity of systems and their different transactional requirements, and describes approaches to how these diverse needs can be supported.

`ReflecTS` contrasts previous work in several matters. Within `ReflecTS` transaction services can be added or removed according to the needs of applications and execution environments. The services exhibit different guarantees, may run

concurrently, and can consequently cover different transactional requirements simultaneously. `ReflecTS` adapts to varying requirements by performing a selection among a number of the available transaction services. This selection is based on a formal description of transactional requirements and transaction services.

5. CONCLUSIONS AND FUTURE WORK

We argue that transactional middleware must meet varying transactional requirements from applications and execution environments. Our contribution to meet these requirements is the design of `ReflecTS`, a flexible transaction processing platform maintaining an extensible number of concurrently running transaction services. Transaction service selection is based on XML-specifications of requirements and deployed transaction services, making the platform adjustable to different application needs. `ReflecTS` is built using the reflective building blocks OpenCOM components and ReMMoC component frameworks, making services configurable and reconfigurable.

Our approach currently concentrates on transaction service selection based on XML-specifications. The selection of an appropriate transaction service for transaction execution justifies the intention of our work - to assemble a flexible transaction processing system.

Ongoing and future work includes further development of the `ReflecTS` prototype to also include structural reflection, and an evaluation of the selection procedure. Designing ReflecTS for a web services [15] environment is also a part of current work. Further, we are working on the transaction service management part, assuring correctness for concurrently running transaction services.

6. REFERENCES

[1] Corba services, transaction service specification, v1.1, 1997.

[2] Yousef J. Al-Houmaily and Panos K. Chrysanthis. Atomicity with incompatible presumptions. pages 306–315, 1999.

[3] R. Barga and C. Pu. Reflection on a legacy transaction processing monitor, 1996.

[4] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, London, 1998. Springer-Verlag.

[5] Gordon S. Blair, Geoff Coulson, Anders Andersen, and Lynne Blair et.al. The design and implementation of open orb 2. *DSOnline*, 2(6), 2001.

[6] Don Box. *Essential COM*. Addison-Wesley, 1998.

[7] Panos K. Chrysanthis and Krithi Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[8] Michale Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware*, Heidelberg, Germany, 2001.

[9] Microsoft Corporation. The .net framework, 2000.

[10] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

[11] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987.

[12] Paul Grace. *Overcoming Middleware Heterogeneity in Mobile Computing Applications*. PhD thesis, Lancaster University, 2004.

[13] The Open Group. The x/open cae specification. distributed transaction processing: The xa specification. x/open document number: Xo/ca/91/300, December 1991.

[14] The Open Group. X/open cae distributed transaction processing: The tx specification, December 1995.

[15] W3C Working Group. Web services architecture, working draft, February 2004.

[16] I. Houston, M. C. Little, I. Robinson, S. K. Shrivastava, and S. M. Wheater. The corba activity service framework for supporting extended transactions. *Lecture Notes in Computer Science*, 2218, 2001.

[17] Randi Karlsen. An adaptive transactional system - framework and service synchronization,. In *International Symposium on Distributed Objects and Applications (DOA)*, Catania, Sicily, November 2003.

[18] Randi Karlsen and A. B. A. Jakobsen. Transaction service management an approach towards a reflective transaction service. In *2nd International Workshop on Reflective and Adaptive Middleware*, Rio de Janeiro, Brazil, June 2003.

[19] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, oct 1987.

[20] J. E. Moss. Nested transactions: An approach to reliable distributed computing, 1985.

[21] Marek Prochazka. Advanced transactions in Enterprise Java Beans. *Lecture Notes in Computer Science*, 1999, 2001.

[22] Heri Ramampiaro and M. Nygaard. Cagistrans: Providing adaptable transactional support for cooperative work. In *Proceedings of the 6th INFORMS conference on Information Systems and Technology (CIST2001)*, 2001.

[23] K. Ramamritham and P.K. Chrysanthis. *Executive briefing: Advances in concurrency control and transaction processing*. IEEE Computer Society Press, Los Alamitos, California, 1997.

[24] B.C. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, MIT, MIT Computer Science Technical Report 272, Cambridge, 1982.

[25] Allarmaraju Subhramanyam. Java transaction service, 1999.

[26] Clemens Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[27] Zhixue Wu. Reflective java and a reflective component-based transaction architecture. In *OOPSLA workshop*, 1998.

[28] A. Zarras and V. Issarny. A framework for systematic synthesis of transactional middleware, 1998.