Towards a Reflective Transactional Middleware Platform for Advanced Applications

Anna-Brith A. Jakobsen and Randi Karlsen

Computer science department, University of Tromsø, 9037 Tromsø, Norway

annab@cs.uit.no, randi@cs.uit.no

October 14, 2003

Abstract

Transactional middleware platforms must accommodate an increasingly diverse range of requirements from both the applications and the underlying systems. It is clear that advanced applications have characteristics and requirements that vary a lot, and that transactional middleware must be able to support the potential variety in transaction execution requirements. We describe a configurable and re-configurable reflective middleware platform that meets the needs of applications by supporting concurrently executing transaction services in a consistent way. The platform exposes the ability to deploy and manage transaction services according to the needs of the applications. The ability to perform these tasks is provided through reflection. A described meta-obejct protocol represents the reflective features. The platform is built upon the concepts of component frameworks and reflection as proposed in the OpenORB project.

1 Introduction

In traditional database systems, the transaction concept is related to the flat transaction model and the key to success is the traditional ACID properties. However, new, advanced applications have varying characteristics and transactional requirements, and the traditional transaction model is generally not applicable. New, advanced applications are often of long duration and their requirements to transaction execution are not limited to the traditional ACID properties, but may for instance include requirements for timely constraints or semantic atomicity [1].

An example of applications where dynamically changing transactional requirements is needed is medical information systems. In such systems, information of different types is stored over a number of sites. Users can for instance execute simple transactions involving read or updated of a single patient journal, collect statistical data from widespread, distributed sources, or access patient journals while on the move. This involves handling long-running computations and mobility issues. Other situations may cause the user to bind timely constraints to the transaction. An example is an emergency situation where fast access to information is vital. Transactions may also require real-time access to multimedia information such as radiographs or spoken reports. This may involve quality of service guarantees including timeliness, throughput, media quality, and synchronization.

Application can issue different types of transactions, from simple to complex. Different applications can have different requirements, and requirements can vary within an application. This gives a diverse set of requirements to the transaction execution, which must be flexible and adapt to these needs. In addition, transaction execution may be affected by resource fluctuations in the underlying system, including bandwidth, connectivity, CPU, storage.

We argue that an adaptive transactional middleware platform is required to support the varying needs from advanced applications and from the execution environment. Traditional transaction processing and transactional middleware systems, such as CORBA's Object Transaction Service (OTS)[2] and Enterprise JavaBeans' Java Transaction Service (JTS) [21], supports mainly the traditional flat transaction model. These transaction processing environments can therefore not support transactional requirements demanded by most advanced applications. Over the years, a number of advanced transaction models have been proposed, that addresse specific transactional requirements and offer some flexibility within a limited context [1, 18]. They do, however, not support the required flexibility in a principled way.

Transaction services must accommodate requirements from both applications and underlying system, and they must be capable of deployment-time configurability and run-time reconfigurability. Component technology is the applied approach to achieve these requirements in software systems since components can be added, removed, reused and configured. However, middleware systems like Enterprise JavaBeans and CORBA Component Model are still 'black boxes' where component technology is only applied at the application level. Emerging reflective middleware platform utilize the concepts of open implementation and reflection to get access to the underlying virtual machine. A number of such platform have been developed, including OpenORB [5][6], Dynamic TAO [19], Open CORBA [14], and Flexinet [10]. These platforms offer great flexibility in terms of meeting the needs of application domains. However, they do not provide transactional services.

OpenORB is a component-based reflective middleware platform. OpenORB combines the reflective OpenCom component model [8] with component frameworks (CF) [22]. OpenORB is structured as a set of configurable CF's, and reflection [15, 13] is used to discover the current structure and behaviour, and to enable selected changes at run-time. We will exploit the key technologies, OpenCOM, CFs, and reflection to describe a configurable and reconfigurable transaction service platform, the *FlexTS platform*. The FlexTS transaction service platform aims to meet the varying transaction execution requirements. FlexTS can contain an extensible number of concurrently running transaction services (TS) while consistent transaction execution is preserved. During run-time, TS's can be added or configured according to needs of applications. Their ability to run concurrently depends on the compatibility defined between them. The platform will perform the work guaranteeing consistency, determine compatibility, configure and reconfigure transaction services and issue transaction execution. We will also describe the meta-object protocol supporting the FlexTS platform, which is formed by the interfaces provided by the platform.

This work is a part of the Arctic Beans project[3] , which is funded by The Research Council of Norway. The primary goal of the Arctic Beans project is to provide a more open and flexible enterprise component technology, with support for configurability and re-configurability, based on the principles of reflection.

This paper is organized as follows. Section 2 provides background information on OpenCOM, component frameworks, and OpenORB. Following this, section 3 describes the FlexTS platform. Section 4 presents related work. Finally, the last section gives a summary and directions for future work.

2 Background on OpenCOM components, Component Frameworks and OpenORB

Component technology is the most widely adopted and central technique to construct configurable software systems. According to Szyperski [22], 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. Systems made of components can be configured and reconfigured by adding, removing or replacing their constituent components.

OpenCOM [8] is a reflective component model built atop a subset of Microsoft's COM [9]. The implementation of OpenCOM relies on core aspects of COM, and omit higher-level features such as distribution, persistence, security and transactions. The core aspect's OpenCOM is implemented on is *i)* the binary level interoperability standard, *ii)* Microsoft's Interface Definition Language (IDL), *iii)* COM's globally unique identifiers (GUID's) and *iv)* the IUnknown interface.

The fundamental concepts of OpenCOM are *interfaces*, *receptacles* and *connections* (bindings between interface and receptacles). Every component declare the services they provide through *interfaces*, and the required services through *receptacles*. A runtime substrate is available in every OpenCOM address space that manages a repository of available components, supports the creation and deletion of components. The OpenCOM runtime substrate also keeps a system graph of the components currently in use to support the introspection of the platform structure. The OpenCOM architecture is illustrated in Figure 1.

Each OpenCOM component implements five standard interfaces: two component management interfaces (*ILifeCylce* and *IReceptacle*) and three meta-interfaces (*IMetaInterception*, *IMetaArchitecture* and *IMetaInterface*. Only three interfaces are mandatory (ILifeCycle, IReceptacele and IMetaInterface). The ILifeCycle interface provides operations for creating and destroying a component while IReceptacle offers methods to modify the interfaces connected to a component's receptacles. The IMetaInterception interface enables interceptors to be associated with a given interface. Interceptors are methods that can be invoked before or after every method invocation on the specified interface. IMetaArchitecture allows the programmer to obtain information about the underlying component architecture and IMetaInterface supports inspection of a components interfaces and receptacles.

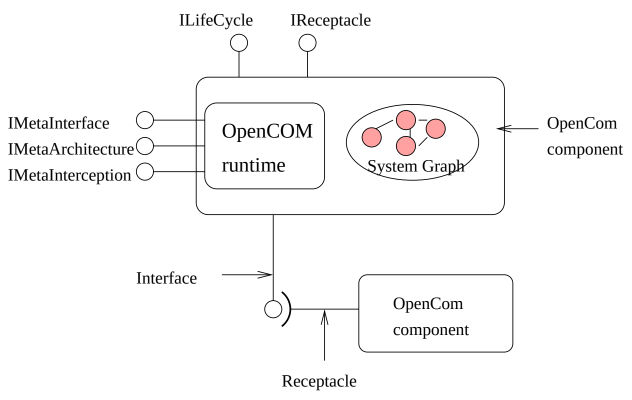

Figure 1: OpenCOM COmponent Model

OpenCOM Component Framework A component framework, CF, is defined in [22] as a collection of rules and interfaces for the components that belongs to the CF. A component framework is based upon the concept of composite components, and is itself a component that consists of other components. Users interact with a CF for services through well-defined application programming interfaces. A CF in OpenCOM is an OpenCOM component that maintain internal structure, component graph, and interfaces. Each OpenCOM CF implements at least the mandatory interfaces of an OpenCOM component. An OpenCOM CF offers also custom interfaces and receptacles representing provided and required services, which also can be exposed interfaces of internal components. To achieve access to internal structure (reflection), an OpenCOM CF may implement the ICFMetaArchitecture and the ICFMetaInterception interfaces.

OpenORB is composed of OpenCOM components and component frameworks, and can be configured and re-configured through the principles of reflection. The architecture of OpenORB is layered. Each CF in a given layer can access intefaces offered by CFs in the same layer or the layer below. A top-level CF maintains the layer-composition of the architecture. The top-level CF manages the lifecycle of all hosted CFs, and resolved the dependencies between them.

OpenORB is currently composed of five CFs and a number of components. The CFs are divided into three layers: the binding layer, the communication layer and the resource layer. CF's representing functional properties, like binding types, can interact with CF's representing non-functional properties, like resource discovery protocols. The hierarchical structure of OpenORB opens up for several possible middleware architectures, and the one described in [6] is one of several possible architecture. Adding more component frameworks can extend the platform.

We propose extending the OpenORB platform with the FlexTS platform. The transactional services offered by the FlexTS platform will be provided to both the underlying system and the application. The application, and possible also the underlying system, will initiate requests for transaction execution and transaction service configuration and re-configuration. Information about resources in the underlying system comes from one of the layers in the architecture. CFs in the resource layer can for instance have monitoring functions, and the transaction execution can adapt based on this information. For this reason, we propose FlexTS to be vertically placed in the OpenORB architecture. This can be seen

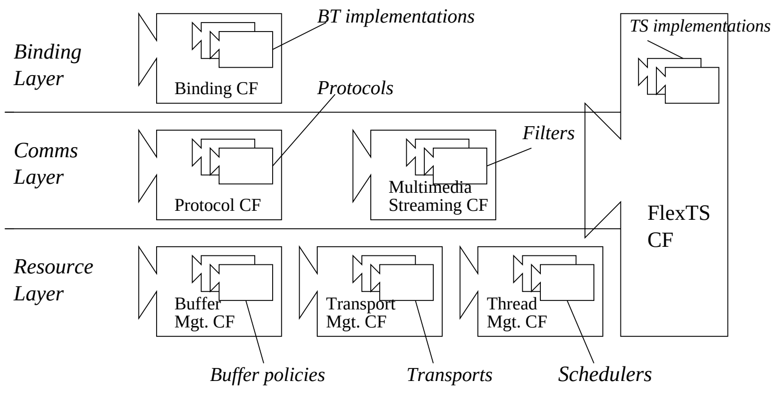

Figure 2: OpenORB with FlexTS extension

in Figure 2. The FlexTS platform is described in the next chapter.

3 A Reflective Transactional Middleware Platform

The main intension of the reflective transactional middleware platform, the FlexTS platform, is to enable flexible transaction processing for advanced applications while meeting different transactional requirements. The FlexTS platform solves this by allowing different transaction services to coexist and be used concurrently within the same application while consistent use of the services is guaranteed. The platform will adjust according to the needs of the applications. These adjustments involve selection of appropriate transaction services, and configuration and re-configuration of services. The FlexTS platform's functionality is based on the description of the TSenvironment in [12, 11], which represents a model for an adaptable transactional system. The model is briefly described below.

Model for an Adaptable Transactional System

An architecture for a reflective transactional system, called *Transaction service execution environment (TSenvironment)* is presented in [12, 11]. In the TSenvironment, transaction services can be deployed and modified, and used concurrently according to the needs of the applications. The TSenvironment includes a *Transaction service manager (TSmanager)* that controls transaction service deployment and modification, and guarantees consistent use of different transaction services.

A *transaction service, TS* is a self-contained software component that is independently developed and delivered to a transactional environment. A TS may be composed of a number of smaller components *(task components)*, which are assembled to form a complete and consistsent TS. Task components represent well-defined tasks within a transaction service, for instance commit, recovery, lock and resource management procedure.

Component deployment: The TSmanager handles deployment of transaction service components. The TSmanager also stores information about the components as provided in accompanying component descriptors, holding relevant information about the component (e.g. transaction management properties, service compositiont, conditions for using the component, and component *compatibility*).

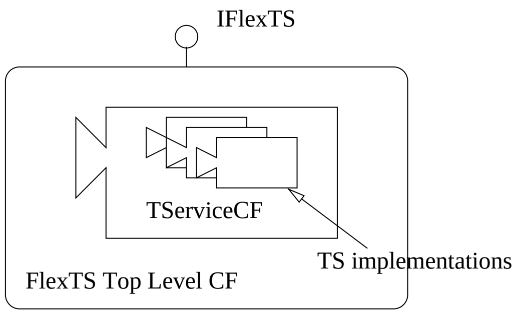

Figure 3: Overview of the FlexTS platform

Service assembly: A component deployed in the TSenvironment can, as described, either represent a complete TS or a task component that must be assembled with other tasks to form a complete service. Assembling of task components are based on information from a component descriptor. A complete TS is registered in the TSenvironment, and the descriptor stored in an information base.

Service activation: When transaction execution is requested, the TSmanager i) determines the proper TS to use, and ii) allows the TS to start execution when this does not violate correctness.

Based on information from user, available system resources and underlying systems, a suitable TS will be determined from the pool of available services. The selected TS may be either active or inactive. An *active service* is currently managing at least one transaction, while an *inactive service* is currently not used by any transaction. *Compatible* transaction services do not interfer with each other and can be active at the same time. Two services are *incompatible* if the service properties of either one of them cannot be guaranteed if they are both active at the same time. The activation and use of TS's are controlled so that incompatible services do not cause inconsistent transaction executions.

Design of FlexTS Platform

This section describes the FlexTS platform, a configurable and re-configurable reflective middleware platform that meets the varying transactional requirements from applications by supporting concurrently running TS's. In FlexTS, only complete transaction services will be deployed at this stage. Deploying and assembling of task components will be a future extension. The FlexTS platform consists of a plugged-in *component framework (CF)* and a number of components, enclosed by a top-level CF. The top-level CF represents the FlexTS platform and provides an interface to the application. The FlexTS platform will also provide interfaces to the underlying system at a later stage. This means that, in this version of the FlexTS platform, selecting an appropriate transaction service depends only on information from the application. The plugged-in CF (named *TServiceCF*) consists of deployed transaction services. The TServiceCF is configured by plugging in different transaction service implementations. Figure 3 gives an overview of the FlexTS platform.

The top-level CF provides an application programming interface, *IFlexTS*, to the user/application through which they can configure, reconfigure and select transaction services, and execute transactions. The plugged-in TServiceCF and

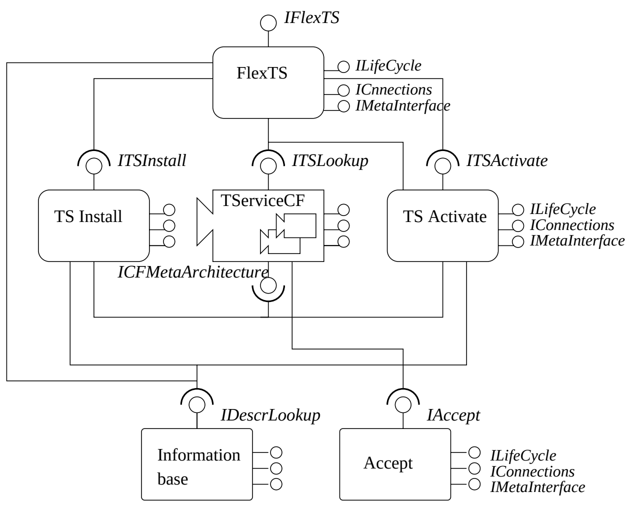

Figure 4: FlexTS platform, Components, Interfaces and Recpetacles

components provides interfaces to the top-level CF. The top-level CF manages its plugged-in CF and components and initiate activity at their provided interfaces. The components in the platform perform specific tasks in the transaction service environment, such as deploying services, selecting a transaction service for a transaction execution, guaranteeing consistent use of the transaction services, maintaining transactional requirements, and adjust according to the needs of the applications. We will in the following describe the plugged-in TServiceCF and components, and theirs provided interfaces. Figure 4 illustrates the FlexTS platform.

FlexTS Component. The FlexTS provides the *IFlexTS* interface, to the applications. The following table lists the methods in the IFlexTS interface.

IFlexTS API	

StartTransaction()
StopTransaction()
ReconfigureTS()
DeployTS()
	DeleteTS()
	LookupDescr()
	ChangeDescr()

FlexTS declares receptacles to required interfaces provided by plugged-in components. When FlexTS receives requests on IFlexTS, FlexTS initiates activity on one of these interfaces. FlexTS declares a receptacle on the ITSActivate interface provided by the TSActivate component, which controls the activation/deactivation of a transaction service. ITSActivate interface is invoked when a StartTransaction() or a StopTransaction() is issued on the IFlexTS interface. A receptacle is declared on the TServiceCF's ITSLookup interface for looking up deployed services, active or inactive. ITSLookup interface is invoked when DeployTS(), ReconfigureTS() or DeleteTS() is issued on the IFlexTS inteface. And, a receptacle is declared on the IDescrLookup interface provided by the InformationBase component, which is invoked for reading or updating of component descriptors. The following table lists the methods in the component's provided interfaces.

ITSLookup i.f.	*ITSActivate i.f.*	*ITSInstall i.f.*	*IDescrLookup i.f.*
GetServices()	StartTrans()	RegisterTS()	LookupDescr()
GetActiveServices()	StopTrans()	UnregisterTS()	StoreDescr()
StartTrans()		ChangeTS()	DeleteDescr()
StopTrans()			UpdateDescr()

TServiceCF hostes transaction services (TSs) and contains information about deployed transaction services, such as whether a transaction service is active or inactive. Only compatible transaction services can be active at the same time.

The TServiceCF provides an *interface ITSLookup*. This interface has methods for looking up TSs and for explicitly giving control over transaction execution to a transaction service. This interface is invoked by the FlexTS top-level component and by the TSActivate component. The FlexTS top-level component invokes the interface when looking up services, and the TSActivate component when looking up services and controlling transaction execution.

TSs can be configured and deployed both initially (at configuration time) or during run-time. To achieve this, the TServiceCF has implemented the *ICFMetaArchitecture* interface for deploying and configuring transaction services. ICFMetaArchitecture has methods for inserting, removing and replacing components. ICFMetaArchitecture has also methods for connecting and disconnecting components. An active TS is a connected composite components, and an inactive TS is a disconnected composite component. See appendix A for a overview of some of the operations provided by the ICFMetaArchitecture interface.

Activity on the ICFMetaArchitecture interface is initiated by the TSInstall component and by the TSActivate component that declares receptacles on this interface. The TSInstall component initiates activities regarding configuration or reconfiguration, such as inserting, removing or replacing TS's. This will invoke the insertComponent(), removeComponent() or the replaceComponent() operation at the ICFMetaArchitecture interface.

TSActivate handles requests for transaction execution. When StartTrans() is issued, an appropriate TS is selected. If the selected TS is active, the transaction is handled over to it by issuing StartTrans() at the ITSLookup interface. If the selected TS is inactive, the TSActivate component first determines its compatibility with other active TSs. If compatibility exists, an operation at ICFMetaArchitecture interface for connecting the TS is invoked. This is the *localBind* operation. If StopTrans() at the ITSActivate interface is invoked, and the transaction to be stopped is the last one active at the particular TS, the TS is turned inactive. This is done by invoking the *breakLocalBind()* operation at the ICFMetaArchitecture interface.

TServiceCF has constraints on the configuration of components to be valid implementations. In the TServiceCF only valid transaction service implementations are allowed to be deployed, and a reconfiguration of a TS must result in a valid TS. To enforce this policy, the TServiceCF declares a receptacle on the *Accept* interface. After configuring or reconfiguring a TS, a call to IAccept is performed. A check is performed on the architecture of the components in the TS, and a positive response is generated if the TS is verified and a valid configuration. Otherwise, and exception is generated. This check can be a comparison against configurations that are defined valid in advance.

TS Activate Component. This component handles requests for transaction execution, and has routines for guaranteeing consistent and correct use of concurrently running transaction services. The component provides an interface, *ITSActivate*, that have methods for starting and stopping transactions. The component declares a receptacle on the ITSLookup interface and the ICFMetaArchitecture interface provided by the TServiceCF, and on the IDescrLookup interface provided by the InformationBase component.

TSActivate invokes the GetServices() and GetActiveServices() methods provided by the ITSLookup interface to determine a proper TS. If the selected TS is active, the transaction is handled over to it by invoking the StartTrans() method at the ITSLookup interface. If the selected TS is not active, its compatibility with other active TS's has to be determined. To investigate this, a call to the LookupDescr() method at the IDescrLookup interface is performed. If the TS is compatible with other active TS's, it can be activated. Otherwise, one has to wait for this event to happen.

To activate a TS, TSActivate issues a call to localBind() operation at the ICFMetaArchitecture interface provided by the TServiceCF. After activation, the transaction is handled over to the TS by invoking StartTrans() at the ITSLookup interface.

TS Install Component. This component implements a protocol for deploying and undeploying of transaction services, and provides an interface *ITSInstall* to the FlexTS platform.

TSInstall component declares a receptacle on the ICFMetaArcthitecture interface and a receptacle on the IDescrLookup interface. The TSInstall component will use the ICFMetaArchitecture operations insertComponent() and localBind() and the IDescrLookup operation StoreDescr() when registrating a TS. Likewise, UnregisterTS() and ChangeTS() will use operations declared at the same interfaces.

InformationBase component: The InformationBase component stores component (TS) descriptors, and provides an interface *IDescrLookup* for reading and updating them. A transaction service descriptor contains information about how to use the TS and information about the transactional guarantees provided by the TS. The InformationBase component also stores information about *compatibility* as a part of the descriptor.

4 Related Work

Reflection is a concept used in different areas to make systems flexible. It was originally introduced in work on flexible programming languages [20], and is now used in distributed systems, operating systems, middleware [5], and also in transactional systems [4, 23]. Our work on a reflective transactional system contrasts previous work on two matters: Firstly, we focus on how to guarantee correctness for the reflective transactional system. The close relationship between different transaction service modules (for instance recovery-, lock- and resource managers), makes it necessary to re-evaluate correctness of the transaction service if the internals of a service is manipulated. Secondly, we allow an application to have a number of concurrently active transaction services, and must consequently guarantee consistent use of possibly incompatible services.

Besides related work on reflective transaction services, our work is also related to research on dynamic combination and configuration of transactional and middleware systems. The work of [7, 16, 17, 24] recognize the diversity of systems and their different transactional requirements, and describe approaches to how these diverse needs can be supported.

5 Summary and Future Work

In this work we argue that transactional middleware must accommodate an increasingly diverse range of requirements from both applications and underlying systems. We have presented a flexible and adaptable transactional middleware platform (FlexTS) that will meet the requirements from advanced applications by allowing configuration and reconfiguration both initially and during run-time. FlexTS can host a number of concurrently running transaction services. Transaction services are chosen automatically based on transaction requirements from the application and possible also the underlying system. We are currently working on how to specify or define the applications transactional requirements.

We are currently developing a synchronization protocol that will guarantee consistent use of concurrent transaction services [11]. And, we have started to work on a prototype implementation of the FlexTS platform. We have described the interfaces provided by the different parts of the platform, forming a meta-object protocol (MOP) for the reflective features. Ongoing work includes an evaluation of the MOP in order to describe a complete MOP.

We have designed the FlexTS platform as a apparently stand-alone unit, but it will have no purpose unless it is integrated in a functional environment. FlexTS platform is suggested to be a part of the reflective, middleware architecture OpenORB.

Appendix A

An OpenCOM CF implements the ICFMetaArchitecture interface to achieve access to internal structure (reflection). The table below lists operations provided by the CFMetaArchitecture interface. The operations represents operations for reconfiguration and inspection.

Operations for Reconfiguration	*Operations for Inspection*
localBind	getInternalComponents
breakLocalBind	getBoundComponents
insertComponent	getInternalBindings
removeComponent	
replaceComponent	
ExposeInterface	
UnExposeInterface	
ExposeReceptacle	
UnExposeReceptacle	
ReplaceConfiguration	
initArchTransaction	
commitArchTransaction	
rollbackArchTransaction	

References

[1] *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

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

[3] A. Andersen, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulø, and W. Yu and. Arctic beans, configurable and re-configurable enterprise component architectures. In *IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, 2001. Middleware.

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

[5] 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.

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

[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 component object model specification, version 0.9. Technical report, October 1995.

[10] R. Hayton. Flexinet open orb framework. Acm technical report 2047.01.00, APM Ltd, Poseidon House, Castle Park, Cambridge, UK, 1997.

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

[12] 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.

[13] Gregor Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[14] T. Ledoux. Implementing proxy objects in a reflective orb, 1997.

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

[16] Marek Prochazka. Advanced transactions in Enterprise Java Beans. *Lecture Notes in Computer Science*, 1999:215–??, 2001.

[17] 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.

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

[19] M. Roman, F. Kon, and R. Campbell. Design and implementation of runtime reflection in communication middleware: the dynamictao case, 1999.

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

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

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

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

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