Abstraction of Transaction Demarcation in Component-Oriented Platforms⋆

Romain Rouvoy and Philippe Merle

INRIA Jacquard Project
Laboratoire d'Informatique Fondamentale de Lille
UPRESA 8022 CNRS – U.F.R. I.E.E.A. – Bâtiment M3
Université des Sciences et Technologies de Lille
59655 Villeneuve d'Ascq Cedex, France
{rouvoy,merle}@lifl.fr

Abstract. Component-oriented middleware becomes the privileged substrate for distributed computing in heterogeneous and open environments. Technically they promote the notion of container as structure to host application components. They transparently take charge of a large set of technical or non-functional services like security or transactions. The transaction service is integrated using a set of transaction demarcation (TD) policies. Nevertheless, they are strongly linked to a specific transactional monitor and they are not often isolated. The main contribution of this paper is to propose a component-based framework to deal with TD policies. Thus, this framework allows one to instantiate several configurations of TD policies with different platforms like EJB, CCM, OSGi, WebServices and several transactional monitors like JTS, OTS, WS-T, BTP, etc. It proposes an extensible abstraction of TD policies. This framework shows that no performance degradation is introduced by the refactoring process.

1 Introduction

Component-oriented middleware such as Enterprise Java Beans (EJB) from Sun Microsystems [1] or the CORBA Component Model (CCM) from the Object Management Group (OMG) [2] becomes the privileged substrate for distributed computing in heterogeneous and open environments. The major reason for this success is that they rationalize the whole process associated to design, development, packaging, assembly, deployment and execution of distributed software. Technically they promote the notion of container as structure to host application components. The containers transparently take charge of a large set of technical or non-functional services like synchronous and asynchronous communication, concurrency, life cycle, activation, persistency, security, transaction, etc. From the transaction point of view, the container has to manage the code related to the transaction management in order to allow the developers to be concentrated on the business code. The delegation of the transaction management is configured via deployment descriptors and the container interprets these descriptors in order to inject the code related to the manipulation of the transaction manager (TM). The different strategies used are identified under the term of transaction demarcation (TD) policies.

Nevertheless, every middleware implementation mentioned before and other implementations like .NET from Microsoft [3] or OSGi from Open Services Gateway Initiative (OSGi) [4] are developing their own code for integrating transactions into containers. Moreover, this integration is not often isolated in the container code. It becomes also difficult to introduce new TD policies. And most of the time, containers are strongly linked to the programming interfaces (API) of a specific transactional monitor. For example, EJB containers work with the Java Transaction Service (JTS) [5] while CCM uses the Object Transaction Service (OTS) [6]. But many other transactional monitor specifications exist such as Web Services Transaction Services (WS-T) [7], Business Transaction Protocol (BTP) [8] or more specific ones like the Java Open Transaction Manager (JOTM) [9].

The main contribution of this paper is to propose a component-based framework to deal with TD policies. This framework is independent both of the transaction manager it uses and of the component-oriented platform which uses it. Thus, this framework allows one to instantiate several configurations of TD policies with different platforms like EJB, CCM, OSGi, WebServices and several transactional monitors like JTS, OTS, WS-T, BTP, etc. This framework proposes an extensible abstraction of TD policies through the application of the *Command* Design Pattern [10]. The six standard policies defined in EJB and CCM are supported, but new ones could be designed and included into the framework also. This framework is implemented in the Java language, and has been experimented with the JOnAS J2EE application server [11] using a JTS manager. This framework shows that no performance degradation is introduced by the refactoring process. This framework has also been successfully integrated into JOnAS and OpenCCM [12] platforms using an OTS manager.

This paper presents the scope of transaction demarcation in Section 2. Section 3 establishes the technical challenges. Next, in Section 4 it introduces the Open Transaction Demarcation Framework (OTDF) and its concepts. Then an example of Java implementation of the OTDF is used for illustrating the capabilities of the framework in Section 5. This implementation allows the experimentation of the framework in various situations validating some of its properties and they are presented in Section 6. From there it establishes the lessons of this work in Section 7 and we conclude on perspectives in Section 8.

2   What Transaction Demarcation Is

Transaction demarcation is a way for guaranteeing the transactional context of an invocation. Basically, the transaction policies evaluate if a method should be invoked under an active transaction or not.

2.1   The Policies

Standards like EJB and CCM define a set of six policies in order to cover most of the situations that could happen. These policies are described in Table 1. The description of each policy is:

Supports

   The *Supports* policy is used when a component's operation is able to take transactions into consideration. If a transaction is activated by the client, it may be used by

Table 1. TD policies defined by CCM and EJB

Demarcation Policy	Client Transaction	Container Transaction
Supports	-	-
	Transaction 1	Transaction 1
Never	-	-
	Transaction 1	RAISES(**NEVER**)
Mandatory	-	RAISES(**MANDATORY**)
	Transaction 1	Transaction 1
Required	-	Transaction 2
	Transaction 1	Transaction 1
Not Supported	-	-
	Transaction 1	-
Requires New	-	Transaction 2
	Transaction 1	Transaction 2

the component's operation. On the other hand, if no transaction is activated, the component's operation will be able to execute the same processing without transactional behavior.

Never

The *Never* policy imposes that the selected operation should not be invoked under a transactional context. If the condition is not verified, an exception –specifying that the *NEVER* clause is not respected– is raised. Otherwise, the component's operation is invoked.

Mandatory

The *Mandatory* policy imposes that an operation should be invoked under a transactional context. If the condition is checked then the operation is invoked else the *MANDATORY* exception is raised by the policy.

Required

The *Required* policy is used if an operation requires transactional features. If a transaction has already been activated, the policy propagates this transaction to the operation, else the policy begins a new transaction which would be committed after the invocation.

Not Supported

The *Not Supported* policy checks if a component's operation needs features which could be conflicting with an activated transaction. If a transaction is active, the policy will suspend it for the execution of the method and resume it after.

Requires New

Finally, the *Requires New* policy presents the need for a local transaction for the method. If a transaction is already active, the policy suspends it before beginning a new one. This new transaction is committed just after the execution of the method and the suspended transaction is resumed.

2.2   The Domains

Looking at the TD policies, another organization of these policies could be defined, providing a more structured representation. The goal is to observe each policy for de-

Demarcation
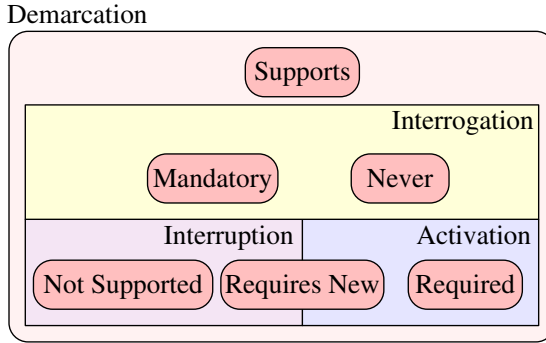
Fig. 1. An Organization of the TD Policies

termining affinities between them. Thanks to this observation, we define the concept of domain. A domain represents the realizable actions of a policy. For example, the *Never* and *Mandatory* policies are only consulting the state of the transaction manager. The *Required* and *Requires New* policies uses the transaction manager in order to activate and validate transactions. Finally, the *Not Supported* and *Requires New* policies need to suspend and resume transactions during their execution. So a repartition of the policies can be obtained and is illustrated in Figure 1.

One can notice that the *Supports* policy is not included inside a domain. The reason is that the *Supports* policy does not interact with any transactional monitor, by consequence no domain could be associated to this policy, while the *Requires New* policy is shared by the activation domain and the interruption domain. The *Requires New* policy requires interruption features for suspending and resuming a possible client transaction and it requires activation features for beginning and validating a container transaction. Each part of this organization defines a domain of interaction with the transaction manager. The domain is both a restriction of the interaction with the transaction manager and a simplification of this interaction specialized in transaction demarcation.

3   Challenges for Transaction Demarcation

This section discusses the main challenges of the integration of transaction demarcation in component-oriented platforms. Figure 2 presents the different challenges encountered when dealing with transaction demarcation. The first challenge is the factorization and the abstraction of the TD policies for addressing more component platforms. The second challenge is the specialization of the TD policies for a given transaction manager. Other technical challenges are more architectural ones and are concerned with organization and extensibility of the TD policies. The following sections detail each of these concerns.

3.1   Transaction Demarcation Abstraction

Most of the platforms define TD policies. The problem is that these policies are implemented inside the scope of a given platform. From a transversal point of view, this code is redundant and needs to be factored between platforms.
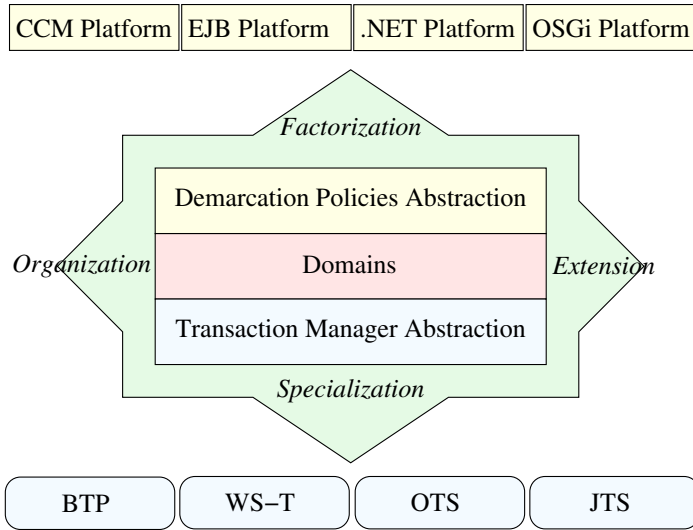
Fig. 2. The Challenges for Transaction Demarcation

But who should address this factorization? It could not be realized by the platform because of the transversality. It could not be realized by the transactional monitor because each policy is designed for interacting with a specific transaction manager which could be different from one platform to another.

So the abstraction of TD policies needs to be addressed by a third party which has to be platform independent as illustrated in Figure 2. This external entity would provide a set of TD policies compliant to the policies defined in the specification of each platform.
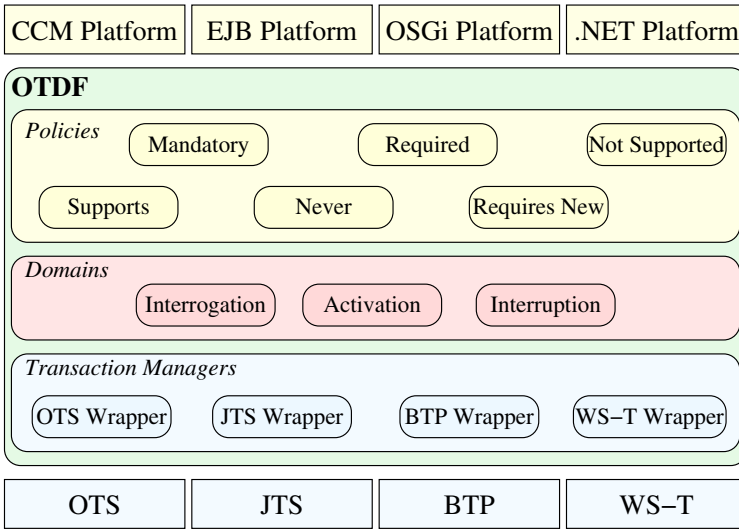
3.2 Transaction Monitor Abstraction

Each platform works with a given transactional monitor but is not able to work with other non-compliant transactional monitor. For example, platforms which are able to interact with both JTS and OTS transactional monitors are not prevalent. EJB platforms are working with JTS while CCM ones are working with OTS. But none are able to address more than one type of transactional monitor.

In order to use transaction policies over multiple transaction managers, an abstraction of the transaction manager must be defined and specialized in component transactions as depicted on Figure 2.

3.3 Organization of TD Policies

TD policies could be linked to one or more domains. The domain ensures the connection between type of policies and transaction managers. But domains also introduce a classification of TD policies as mentioned in Section 2.2. This classification needs to be open as the addition of new TD policies could imply the definition of new domains [13].

Fig. 3. The OTDF Framework Representation

The domain represents an abstraction of the demarcation as we define an abstraction of the transaction manager. This second level of abstraction allows the specialization and the simplification of the functionalities of the transaction manager for the context of transaction demarcation. Then, policies become easier to design because the technical part is managed by the domain.

3.4   Integration of New TD Policies

TD policies introduced by the CCM and EJB specifications are not covering all domains defined by transactional monitors. Indeed domains like sub-transactions or resources management are not introduced by existing specifications. An open structure must be provided in order to allow the integration of new policies [14]. It means mechanisms for adding new policies and associated domains must be provided.

4   Open Transaction Demarcation Framework

Considering the previous technical challenges, we define the *Open Transaction Demarcation Framework* (OTDF). This framework provides a library of configurable TD policies. It addresses the problem of transaction demarcation over any transactional monitor and is usable from any component-oriented platform.

4.1   Overview

Figure 3 is a structured view of OTDF architecture. OTDF is divided into three parts which are the policies, the domains and the transaction manager wrappers. The policies

level considers the types of policies defined in platform specifications (EJB, CCM, etc). The transaction manager wrappers level groups the abstractions for the different models of transactional monitors (OTS, JTS, WS-T, etc). The domains level considers the interactions between a policy and a transaction manager.

The framework is open in order to integrate new TD policies and to cover unexplored domains of transaction demarcation. It provides mechanisms and methods for facilitating this integration.

From a technical point of view, the framework provides an object-oriented and a component-oriented vision of the TD policies. In the meantime the framework is independent of any programming languages as it is designed using the Unified Modeling Language (UML) [15].

4.2   Abstraction of Transactional Monitor

The first technical challenge addressed in this section is the problem of the abstraction of the transactional monitor. Usually, containers strongly adhere to one transactional monitor. Sometimes, the implementations of transaction managers referenced could be changed if they implement the same interfaces. But this variability remains restricted to transaction manager implementations of a same specification (JTS, OTS or WS-T).

In order to provide a better abstraction, the *Wrapper* Design Pattern [10] is used. According to the method, the abstract transaction manager is mapped to a specific implementation of a manager.

The UML diagram of Figure 4 illustrates the application of this design pattern to the transaction manager abstraction.

Our objective is to define a set of transaction manager interfaces with different levels of complexity. Each transaction manager could not implement every potential functionalities addressable by transaction demarcation. For example, JTS monitors are not able to manage nested transactions. So the differences between the transaction manager abstractions need to be identified. By extension, the transaction manager abstraction interfaces would be able to take into account the future evolution of the transaction manager models [16].

The approach for modeling levels of complexity is to define a set of *feature* interfaces. A feature represents a particular property of a transaction manager model. A feature is independent of any other feature. Figure 4 introduces features like configuration, control and interrogation. Because of the fine granularity of features, the interface of a transaction manager could be defined by composing features.

The specialization of the interfaces specifies with a finest granularity the connection between the transaction manager abstraction and a domain. Sometimes, the domain requires more than one feature. But this set of features need to be associated to only one transaction manager. So we define the notion of *scope* for answering this problem. A scope is a composition of features which could be required by a domain. *Scope* interfaces are defined by an extension of the features.

In addition, the combination of features allows also the reproduction of specific models like the Client/Server model of the Java Transaction API as depicted on Figure 4.

When integrating a transaction model, only the elements introducing new functionalities need to be defined using the *Feature* paradigm. If a new domain is introduced with
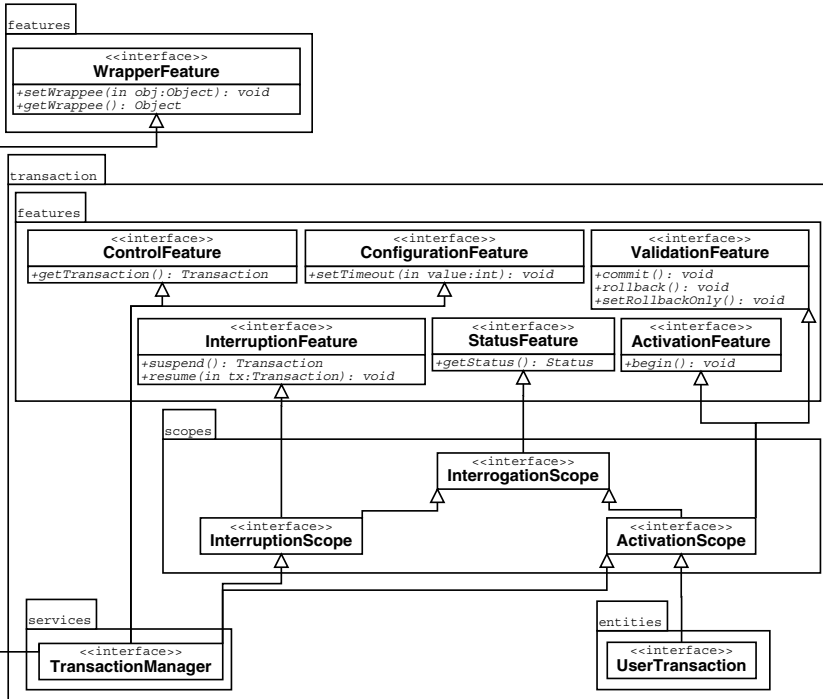
Fig. 4. The Class Diagram of Transaction Manager Abstraction

these features, the associated *Scope* need also to be defined. Other features and scopes are inherited from the framework and the transaction manager abstraction is defined by composing all the features and scopes required for the transaction model.

4.3   Abstraction of Transaction Demarcation

Another technical challenge is the abstraction of the TD policies. In a traditional way, containers implement TD policies using no particular methodology. Most of the time, they check the type of policies they have to activate and the associated treatments are integrated inside the container.

So there are two objectives to reach. Firstly, the extraction of the TD policies from the container. Secondly, we need to burst the block of TD policies into a set of independent TD policies. The first objective is just an extraction of the code managing the transaction demarcation. The other one could be obtained using the *Command* Design Pattern [10].

The root interface *RequestCallController* introduced in Figure 5 is independent of the type of policy implemented. So any TD policy has simply to implement the root interface in order to be compliant with the framework.

The interface *RequestCallContext* offers a generic structure for transmitting parameters to the policies. The lifespan of this interface follows the invocation of a component's method.
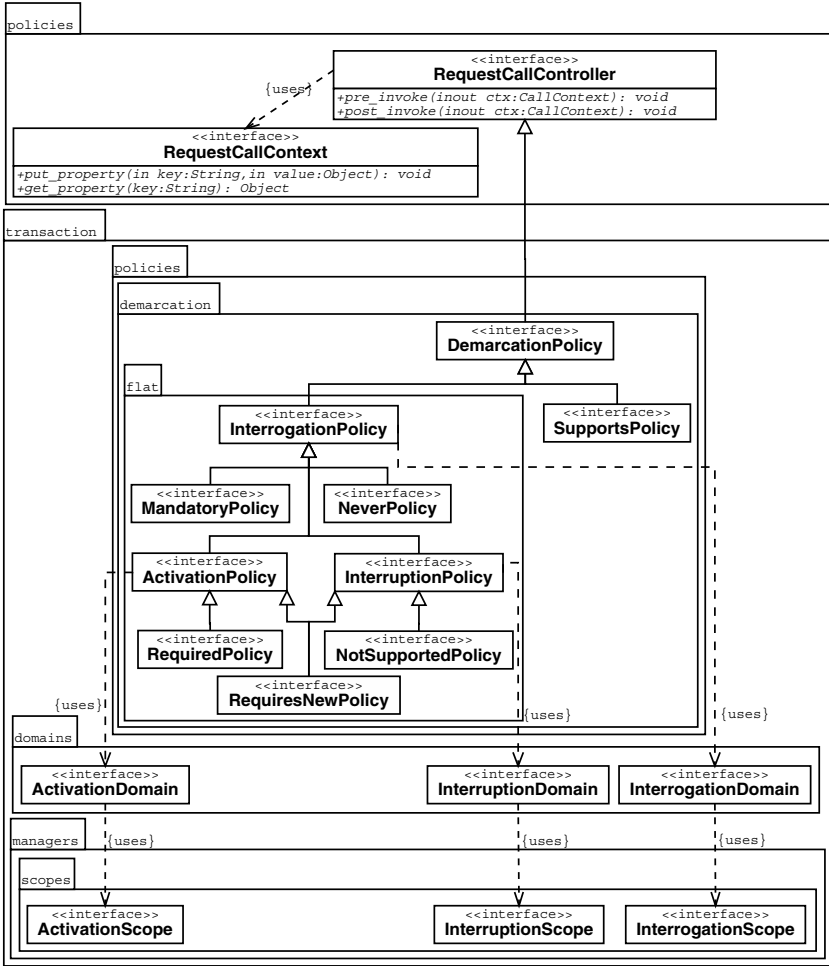
Fig. 5. The Class Diagram of Transaction Demarcation Abstraction

So the information about the associated method and its invocation are not localized with the policy but with the *RequestCallContext*. The consequences of this choice are that this information is defined just before the invocation of the component. And if policies do not contain information about the associated method, they could be shared between the containers.

We define different types of policies which are linked to a specific domain. These types of policies have the same name as the corresponding domains. So the child policies would inherit from the properties defined in the parent interface. The considered properties are the connections to the associated domain.

Thanks to this abstraction, some added values are obtained:

- *identification*: Each TD treatment is associated to a TD policy,
- *adaptability*: By selecting the TD policies to integrate inside the platform,
- *isolation*: By addressing only the TD policy needed, and
- *factorization*: The technical code is delegated to the domain.

4.4   Integration of New TD Policies

The structure of the framework defining independence between platforms and transactional monitors provides a lot of flexibility. A TD policy is composed of a policy and a transaction manager. Thanks to the different abstractions we define (transaction manager and policies), two solutions are possible. They are based on the definition of new transaction manager abstraction and new type of policies. Another solution uses the composition property for defining new policies using existing ones.

Integration of a New Type of Transaction Manager: A solution for defining new TD policies is to integrate –using the abstraction– a new transaction manager abstractions like the BTP abstraction into the framework. As a consequence, all the policies of the framework are associated to the new transaction manager abstraction. A set of new TD policies for the BTP transaction model are obtained using the legacy policies and domains which would be connected to the BTP transaction manager abstraction.

Definition of a New Policy: Another solution consists in defining a new type of policies. An example of this type of policy could be the *Requires New Sub* policy [17]. This type of policy activates a nested transaction inside the scope of an active transaction. The definition of this new policy is based on the abstraction of TD policies. So the new policy has to be associated with the transaction manager which supports nested transactions. The result is a set of new TD policies for a new type of policy using the legacy transaction managers.

Combining Existing TD Policies: Using the *Requires New Sub* policy mentioned before, one could observe that the semantics are incomplete. Indeed the behavior of this policy has not been defined if invoked under no transactional context. How could a new nested transaction be started if no top-level transaction is active? A solution is to say that before beginning the nested transaction, a top-level transaction has to be begun. Another solution is to force the client to invoke the method under a transactional context like it is defined in the *Mandatory* policy. Similar options could be considered also for the case of an invocation realized under a transactional context.

Rather than implementing a policy for each solution which would not be in favor of our process of flexibility, we choose to introduce the notion of TD policy composition. Indeed the composition of the *Required* policy and the *Requires New Sub* policy answer the question mentioned before. The composition of the *Mandatory* policy and the *Requires New Sub* policy provides another solution. Another example is the composition of the *Not Supported* and the *Required* policies. This association simulates the *Requires New* policy.

This combination is realized using a coordinator which organizes the calls to the delegated policies. Next, if delegated policies are themselves a combination of policies, an organization representing a tree of policies could be extracted. So a hierarchy of policies could be defined inside the container. The interesting point is that a hierarchy defines a set of branches and some of them could be preferred to others during the execution.

An evolution of this hierarchy could define "Clever coordinators" in order to switch between the delegated branches or policies depending on a specific clause. The structure of the tree becomes dynamic and the policies executed are different according to the context of the invocation. The framework executes only the policies which are compliant to the execution context.

We are not defining a new organization which could be conflicting with the notion of domain. This hierarchy is only an organization of the policies execution.

5 Java Open Transaction Demarcation Framework

We choose to implement our framework with the Java language [18] essentially for experimental reasons. Many component-oriented platforms are implemented in Java (JOnAS[11], JBoss[19], OpenEJB[20], EJCCM[21], OpenCCM[12], JEFFREE[22]) and as well for transactional monitors (OpenORB TS[23], Tyrex[24], JOnAS TS[11], JOTM[9]).

The multiplicity of implementations allows the experimentation of our framework with various platforms and transactional monitors for the validation of our proposal.

In order to provide several implementations, the framework provides a separated view of interfaces (API) and implementations.

5.1 Transaction Monitor Wrapper

The abstraction of the transaction manager introduces a semantics which is particularly adapted for transaction demarcation. Figure 6 illustrates how a transaction manager could be wrapped and how the semantics introduced by the interfaces is implicitly translated.

From Figure 6, the *UserTransaction* interface inherits from the group of features dealing with activation properties. It also has to implement a configuration feature materialized by the *set_transaction_timeout* method. These features are implemented through the *UserTransaction* interface which defines the features to implement in the scope of a JTS *UserTransaction* interface. So the Java abstraction of the transaction manager needs to be mapped with the transactional monitor used. In this example the wrapped functionalities of the transaction manager are the JTS ones.

The originality of this abstraction comes from the definition of the features which will allow one to extend the abstractions if new transaction models and concepts are defined.

5.2 Transaction Domain

The second level of abstraction is the domain. It models a subset of the functionalities of the transaction manager.

```
public class JTSUserTransactionImpl
  implements UserTransaction {
    protected javax.transaction.UserTransaction tm_ ;

    public void set_transaction_timeout(int seconds) {
        try { tm_.setTransactionTimeout(seconds); } catch(Exception ex) { ... }
    }
    public Status get_status() {
        try { return JTSStatus.jts_to_status(tm_.getStatus());
        } catch (javax.transaction.SystemException ex) { ... }
    }
    public void begin() {
        try { tm_.begin(); } catch (Exception ex)  { ... }
    }
    public void commit() {
        switch(get_status()) {
            case Status.STATUS_ACTIVE :
                try { tm_.commit(); } catch(Exception ex) { ... }
                break ;
            case Status.STATUS_MARKED_ROLLBACK :
                rollback();
                break ;
        }
    }
    public void rollback() {
        try { tm_.rollback(); } catch(Exception ex) { ... }
    }
    public void set_rollback_only() {
        try { tm_.setRollbackOnly(); } catch(Exception ex) { ... }
    }
}
```

Fig. 6. The JTS Transaction Manager Wrapper

```
public class InterruptionDomainImpl
  extends InterrogationDomainImpl
  implements InterruptionDomain {
    protected InterruptionScope is_ ;

    public void suspend(RequestCallContext ctx) {
        try { ctx.put_property("transaction_suspended", is_.suspend());
        } catch(SystemException ex) { ... }
    }
    public void resume(RequestCallContext ctx) {
        Transaction _tx = (Transaction) ctx.get_property("transaction_suspended");
        if (_tx != null)
          try { is_.resume(_tx); } catch(Exception ex) { ... }
    }
}
```

Fig. 7. An Example of Demarcation Domain: The Interruption Domain

Figure 7 illustrates how to define a domain for abstracting a part of transaction demarcation. But the domain provides a simplification of the TD business code. So the domain has a *RequestCallContext* for storing properties related to the transaction manager. Concretely, the *RequestCallContext* is used for storing the instance of the transaction which is suspended during the invocation of the method. In the case of the

```
public class NotSupportedPolicyImpl
  extends AbstractInterruptionPolicy
  implements NotSupportedPolicy {
    public void preinvoke(RequestCallContext ctx) {
        if (id.get_status(ctx) == Status.STATUS_ACTIVE)
            id.suspend(ctx) ;
    }
    public void postinvoke(RequestCallContext ctx) {
        id.resume(ctx) ;
    }
}
```

Fig. 8. An Example of TD Policy: The Not Supported Policy

InterruptionDomain, the domain requires a set of transaction features which are defined in the *InterruptionScope* interface of the transaction manager.

So from the policy, the developer does not need to manage the propagation of the transaction between the *pre_invoke* and the *post_invoke* methods.

Different possibilities of evolution are possible from the definition of domain. We could define new domains based on the scopes defined by a transaction manager abstraction, this domain would contain the use rules of the selected scope applied to transaction demarcation. But we could also modify the behavior of an existing domain by introducing for example interactivity such as "commit/rollback choice" for the container transactions. The behavior of the TD policy is changed without modifying the content of the policy itself.

5.3   Transaction Policy

Until now, the structure of the framework has been defined and introduced without dealing with the type of policies. Figure 8 presents the implementation of the *Not Supported* policy.

Figure 8 promotes that defining a TD policy is easier than before while being both platform and transactional monitor independent. Moreover, the behavior of a policy becomes easier to understand. One has just to read the code of the policy to understand what it is able to do and could notice that only three lines are necessary to define the policy.

5.4   Platform Usage

This section is an illustration of the integration of JOTDF in a platform which requires TD policies. Most of the time, platforms are not designed for working with JOTDF so an adaptation class for converting the formalisms need to be defined for adapting the framework to the platform.

Figure 9 depicts the mechanism of interception which could be used for integrating TD policies in a component platform. The component is defined by the *Account* interface and the *AccountImpl* implementation class. The component platform generates the *AccountInterceptor* class which would delegate the incoming calls realized by the *Account* interface to its implementation. The interceptor calls also the TD policy through the *DemarcationPolicy* interface just before and after the delegation.
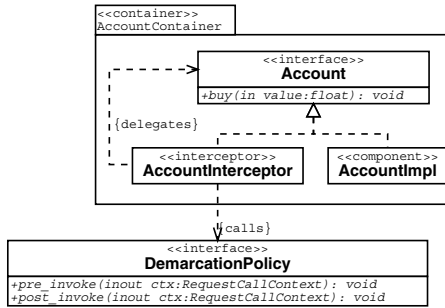
Fig. 9. Interception on a Container hosting Account Components

6   Experimentations

All the design patterns introduced in this paper provide lots of architectural properties. But what about performance? Does this framework introduce a huge overhead to the execution time? Is it really possible to use an EJB platform over an OTS transactional monitor? This section will answer to these questions giving valued results of JOTDF based on various experimentations realized with and without JOTDF.

6.1   Context and Scenario

For evaluating the framework, the computer used is based on an Intel Pentium4 2 GHz with 1024 MB of RAM (DELL Optiflex GX 240). The operating system installed is a Linux Debian based on version 2.4.19-686 of the Kernel. The experimentation is realized on a single computer in order to avoid the interferences generated by the network.

From the software point of view, the Java Development Kit used is the JDK 1.4.1_01 provided by Sun Microsystems. The platform used for the experimentation is an EJB platform working with a JTS transactional monitor. The platform and the transactional monitor are provided by version 2.5.3 of the JOnAS Application Server.

The version of JOTDF used during the experimentation is a basic implementation where we choose to merge the domains and the transaction manager abstraction into one class. This choice illustrates that more or less flexibility could be applied to the framework. A single object is used, which groups the three main domains and the transaction manager abstraction. So each policy would be bound to this object and then will delegate the technical code to the transaction manager.

As an application scenario, a simple example provided with the JOnAS platform is used. This example is a bank account simulation. It uses a container with transaction demarcation features. The bank account is the business component which is tested. A bench component is added and located with the business component on the same component server.

When the bench component is invoked, it activates the business component. Next it produces 10 000 invocations on the bank account component under a transaction, and then it generates 10 000 invocations on the same business component but apart from any transaction.

Table 2. The Class Evaluation of JOTDF

	Transaction Demarcation	
	without JOTDF	with JOTDF
Classes	**X** TD x **Y** TM	**X** TD + **Z** D + **Y** TM
Example	**6** TD x **3** TM	**6** TD + **3** D + **3** TM
(6 policies, 3 TM)	= 18 classes	= 12 classes
New TD	Y classes	1 class
New TM	X classes	1 class

Initially, the standard version of JOnAS is experimented. Next the experiment is started again using a version of JOnAS coupled to the version of JOTDF described in the previous section.

The execution time of the 10 000 calls in each of the four situations is measured. As the business code is negligible, one can consider that the time of crossing of the container and the TD policy is measured. Moreover the difference of time is taken into account more than the absolute values of the measures.

6.2   The Memory Evaluation

Before applying the scenario, a static evaluation of the framework could be realized. It consists with measuring the number of classes and the size of source code used for the management of transaction demarcation. The study is presented in Table 2.

Observing Table 2, much information could be extracted. The "Transaction Demarcation without JOTDF" term considers a basic implementation of transaction demarcation which could be the definition of a TD policy for each type of transaction manager (Y attribute) and each type of policy (X attribute), while the "Transaction Demarcation with JOTDF" term considers the detailed implementation presented in the paper.

Regarding the size of the framework, a basic implementation would provide much more classes than JOTDF. Concerning the extensions of transaction demarcation, the integration of a new TD policy like the *Requires New* policy depends on the number of transaction managers in the basic implementation. In JOTDF, the integration of such a policy results in the definition of only one class. In the same way, the integration of a new transaction manager like the *BTP* transaction manager would introduce as much of the classes as policies in the basic implementation whereas only one class is required in JOTDF. So in JOTDF, the class evolution is linear whereas it would be exponential in a classic implementation.

JOTDF introduces predictability of memory. Indeed JOTDF could share the TD policies between the components of a same component server. So independently of component count, JOTDF uses one instance of the framework. Actually the size of transaction manager depends much more of the number of containers using transaction demarcation features deployed on the server.

6.3   The CPU Evaluation

After the static evaluation, the dynamic evaluation of the framework could be realized. This section presents the results of the scenario execution. The results are presented in Table 3.

Table 3. The Time Evaluation of JOTDF

Policies	JOnAS	JOnAS & JOTDF	Evolution
Supports	17,697 sec	17,569 sec	-0,72 %
Not Supported	18,324 sec	18,302 sec	-0,12 %
Required	31,013 sec	30,963 sec	-0.16 %
Requires New	42,869 sec	42,832 sec	-0,09 %
Never	91,869 sec	90,940 sec	-1,01 %
Mandatory	97,419 sec	97,027 sec	-0,40 %
		Average	-0,42 %

The results presented in Table 3 are explicit, the use of JOTDF does not introduce a higher cost than a handwritten version of TD policies. JOTDF improves the performances of transaction demarcation of the JOnAS platform.

JOTDF also adds better predictability properties. Indeed the invocation does not depend on all TD policies anymore. According to the *Command* Design Pattern, only the cost of TD policy configured influences the cost of invocations. The more policies are used by the platform the better performances can be obtained regarding classical platforms. So the more policy implementation is following our framework the better is the platform performance.

6.4   Example of Heterogeneity in Middleware: EJB over OTS

Another experiment which has been undertaken is an illustration of the new properties obtained thanks to JOTDF. Indeed we try to use an OTS transactional monitor with an EJB platform. These entities are connected through JOTDF.

Thanks to that properties, platforms are no more forced to work with a specific transactional monitor. Going further, heterogenous platforms could execute over a common transaction manager and even in the same transaction.

7   Lessons

This section will discuss several properties about the OTDF framework and its Java implementation JOTDF. Next the advantages and limits of this approach are described. The OTDF framework shows that we obtain:

- **Independence to language:** OTDF is independent of the programming language. It uses no language specific technologies and the architecture could be implemented in any object-oriented programming language.
- **Independence to platforms:** OTDF could be integrated by any platform requiring transaction demarcation. The container calls OTDF where transaction demarcation needs to be applied.
- **Independence to transactional monitors:** OTDF could interact with any transactional monitor once its abstraction has been integrated to OTDF. Special abstractions are linked to a type of transactional monitors like JTS, OTS, WS-T or BTP. So different implementations of a same type uses the same abstraction.

- **Transaction demarcation abstraction:** OTDF defines an abstraction of transaction demarcation which is sufficiently generic in order to be used by most of the existing platforms. Transaction demarcation introduced by OTDF is based on an encapsulation of the method invocation using pre-invocation and post-invocation pattern.
- **Modularity:** The architecture of OTDF is well defined. It uses several types of modules (policies, domains, transaction managers) which could give several entities.
- **Extensibility:** The structure of OTDF has been designed for extensibility. The new transaction manager could be considered and the same for the new TD policies even if they need to define specific domains. OTDF would support new models of transactional monitors thanks to the notion of features presented in the transaction manager abstraction. Considering this architecture, it is easy to introduce new policies with the associated domains, scopes, features and transaction manager abstraction if needed.
- **Dynamic reconfigurability:** OTDF models the connection between the components of the framework. It becomes easy to modify these bindings at runtime.
- **Adaptability:** TD policies are independent entities. The platform could choose and configure the policies required by the specification and use a subset of the policies proposed by OTDF.
- **Component and Aspect:** OTDF is component-oriented, its architecture is modeled using component model. Moreover components could be propagated through several models to the programming level.
  Aspects [25] are a way to treat the separation of concerns. So policies could be integrated in an aspect for adding transaction demarcation properties to an application which has not envisaged such a property during its development.
- **Validation and safety:** The TD policies of OTDF are validated and reliable. The development of new middleware could lean on OTDF for transaction demarcation scope with the confidence of using reliable TD policies. The process of validation is facilitated by the decomposition of a TD policy in three elementary entities which are easier to test and are shared by the TD policies.

The JOTDF implementation proves that one can obtain:

- **No CPU performance degradation:** The benchmark realized shows that even if the performance is not a hard constraint when OTDF has been defined, one can obtain nevertheless interesting results. This is an illustration that architecture and performance are compatible.
- **CPU predictability:** The *Command* Design Pattern provides better predictability and better scalability in terms of CPU computation.
- **Memory predictability:** Sharing policies and factorizing the demarcation management minimize the memory cost of transaction demarcation.

8   Conclusion and Perspectives

This paper has presented a framework for addressing the problem of managing transaction demarcation over heterogenous transactional monitors.

It introduced several technical concerns which could be noticed when the problem of transaction demarcation is studied. The OTDF framework is an answer to these concerns and provides a long-term solution. OTDF provides an architecture based on the abstraction of technical code for keeping only the business code of the demarcation. The separation of concerns used by OTDF provides added values to the framework. Next, OTDF results in a Java implementation called JOTDF. This implementation is then used in several experiments which provide concluding results.

From there, the work begun by OTDF opens three main perspectives. Considering the context of transaction demarcation, OTDF answers many of the actual concerns raised by the current platforms. But OTDF needs to be experimented on other platforms like JBoss, OSGi or EJCCM. OTDF should experiment new policies like the *MandatoryNewSub* [26] in order to validate its extensibility properties. OTDF gives a brief reply to problems of defining a framework for addressing the transactional deployment. The definition of deployment policies could be a track. Finally JOTDF would become a plugin of the JOTM project which targets to provide a framework for addressing all features related to transactions in distributed systems.

Considering a more generic context, OTDF gives a template for the definition of policies for log, security, naming or trading services. Some of them are currently supported by platforms specification like security policies. Other policies are not introduced by specifications but JOTDF could be a template for integrating new policies inside containers.

The other perspective is the integration of the policies with the platform. Different mechanisms could be considered like interception, Aspect-Oriented Programming (AOP) [25] or Meta-Object Protocol (MOP) [27] mechanisms. Another aspect of the integration is to widen the application of policies. Policies could extended to other contexts other than method invocation. Indeed policies could be used for the bind/unbind, create/destroy and activate/passivate events of a component.

References

1. DeMichiel, L., Yalçinalp, L., Krishnan, S.: Enterprise Java Beans Specification Version 2.0 - Public Draft. Sun Microsystems. (2000)
2. OMG: CORBA Components Specification Version 3.0. Object Management Group. (2002) OMG TC Document formal/2002-06-65.
3. Thai, T., Lam, H.: .Net Framework Essentials. O'Reilly (2001)
4. OSGi: OSGi Service Gateway Specification Release 1.0. Open Service Gateway Initiative. (2000)
5. Cheung, S.: Java Transaction Service Specification Version 1.0, Sun Microsystems (1999)
6. OMG: Object Transaction Service Specification Version 1.2. In: CORBAservices : Common Object Services Specification, Object Management Group (2001) OMG TC Document formal/2001-05-02.
7. Cabrera, F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., Thatte, S.: Web Services Transaction Specification Version 1.0, IBM (2002)
8. OASIS: Business Transaction Protocol Specification Version 1.0, Organization for the Advancement of Structured Information Systems (2002)
9. ObjectWeb: Java Open Transaction Manager (2002) *http://www.objectweb.org/jotm*.

10. Gamma, E., Helm, R., Johnson, R., Vlissides, J., Booch, G.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Westley Professional Computing, USA (1995)
11. ObjectWeb: Java Open Application Server (2002) *http://www.objectweb.org/jonas*.
12. ObjectWeb: Open CORBA Component Model (2002) *http://www.objectweb.org/openccm*.
13. Barga, R., Pu, C.: Reflection on a legacy transaction processing monitor. In: Proceedings Reflection '96, San Francisco, CA, USA (1996)
14. Procházka, M.: Advanced Transactions in Component-Based Software Architectures. PhD thesis, Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranské námestí 25, 118 00 Prague 1, Czech Republic (2002)
15. OMG: Unified Modeling Language Version 1.4. Object Management Group. (2001) OMG TC Document formal/2001-09-67.
16. Barga, R., Pu, C.: A Practical and Modular Method to Implement Extended Transaction Models. In: International Conference on Very Large Data Bases, Zurich, Switzerland (1995) 206–217
17. Procházka, M.: Advanced transactions in Enterprise Java Beans. Lecture Notes in Computer Science (2001) 215
18. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. (1996)
19. JBoss Group LLC: JBoss (2002) *http://www.jboss.org*.
20. ExoLab: Open Enterprise Java Bean (2002) *http://openejb.exolab.org*.
21. Computational Physics, I.: Enterprise Java CORBA Component Model (2002) *http://www.cpi.com/ejccm*.
22. ObjectWeb: JEFFREE (2003) *http://www.objectweb.org/jeffree*.
23. SourceForge: OpenORB Transaction Service (2002) *http://openorb.sourceforge.net*.
24. ExoLab: Tyrex (2002) *http://tyrex.exolab.org*.
25. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
26. Procházka, M., Plasil, F.: Container-interposed transactions. In: Proceedings of the Component-Based Software Engineering special session of the SNPD 2001 Conference, Nagoya, Japan (2001)
27. Robben, B., Vanhaute, B., Joosen, W., Verbaeten, P.: Non-functional policies. Lecture Notes in Computer Science (1999) 74