Reflective Java
and
A Reflective Component-Based Transaction Architecture

Zhixue Wu
APM Ltd., Poseidon House, Castle Park, Cambridge CB3 0RD UK
+44 1223 568930 zhixue.wu@citrix.com

ABSTRACT
In this paper, we first give a brief introduction to our Reflective Java technology. Then we describe our work on using Reflective Java to implement a flexible transaction architecture that meets the needs of Internet applications. Finally, we report some of our experience, particularly our two-layer approach on constructing metaobjects.

Keywords
Reflection, metaobject, Java, transaction, component

INTRODUCTION
Java has become popular as a programming language for the Internet because of its ability to simplify the development of flexible, portable applications with graphical user interfaces. Using Java, users can write a program once and run it on any platform. However, when using Java for developing middleware, it lacks a good mechanism to support adaptability. The Reflective Java system implemented by APM Ltd. in the ANSA programme aims to solve this problem through metalevel programming [5][7]. Metalevel programming allows a clear separation to be made between those parts of an application that are Our architecture tracks closely the Enterprise JavaBeans specification [13], and can be used to execute Enterprise JavaBeans.

concerned with implementing its business logic and those parts that are concerned with addressing system issues. Thus, metalevel programming makes it easier to tune a system component, for example, in order to cater for new application demands, or to adapt to a new environment.

Transaction management is a typical middleware capability. Based on our Reflective Java technology, we have developed a transaction framework for Internet applications. The characteristics of Internet applications requires an architecture that is scaleable, flexible and adaptable, whilst still easy to use, to develop and to deploy. Reflective Java plays an important role in our transaction framework for meeting this demand. First, it enables transaction management to be easily adapted to new application requirements and changing environments. Secondly, it allows programmers to provide application-specific information declaratively and separately from application code. This information can be used either at deployment time for configuring the transactional set to best suit individual application components, or at execution time for improving system performance

REFLECTIVE JAVA
Making Java fully reflective would be a very complex task, and cannot be done without support from its virtual machine and compiler. Because we want to ensure the pure Java requirement, we take an approach that requires no changes to the Java language, its compiler, or its virtual machine. Therefore, we can ensure that applications using Reflective Java can still run on any platform.

The current implementation of Reflective Java makes only one property of Java become reflective, namely method invocation. The basic idea is that method invocations can be intercepted and passed to the corresponding metaobject. The metaobject then decides how to deal with an invocation. In this way, programmers can make method invocations behave according to their particular needs through the implementation of metaobjects, without making changes to either the language, or the application program.

Reflective Java enables programmers to change the behaviour of method invocations by specifying what should be done before or/and after "normal" method execution. For example, Figure 1 shows how to provide concurrency control to an object that is implemented for a sequential environment. The object is bound to a metaobject that locks the object and makes a checkpoint of its state before calling the original object method, and unlocks the object and frees the checkpoint afterwards.

From the example we can see that by using reflection, it is straightforward to provide concurrency control to an object

that is totally transparent to its developer. Other typical system issues can also be addressed in this way.

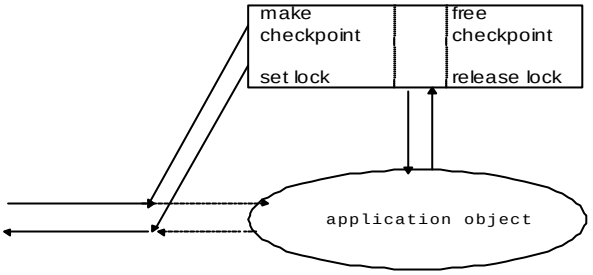

Figure 1: Behaviour of a reflective method

Like OpenC++ [1], Reflective Java implements invocation interception through class inheritance. The general idea is to implement a reflection class as a subclass of an application class. Each operation of an application class is overridden in the reflection class in such a way that an invocation is passed to a metaobject. In this way, if programmers create objects from the reflection class instead of the original application class, the invocation of an object method will be intercepted and dealt with by a metaobject. A reflection class is generated automatically for an application class by the Reflective Java preprocessor.

To be reflective, an object needs to be bound to a metaobject. There are two kinds of binding: static and dynamic. Static binding is done at compile time and cannot be changed subsequently. Dynamic binding allows an object to change its binding to a metaobject at runtime, thus enabling objects to change their behaviour dynamically without stopping. This is an important requirement for Internet applications. Therefore, both static and dynamic bindings are supported in Reflective Java. For each reflection class, Reflective Java provides a pair of public methods, *getMeta* and *changeMeta*, for checking and changing the metaobject of an application object.

A simple declarative language is provided in Reflective Java for end-users to describe the binding specification, that is, to specify the relationship between an application class and a metaobject class. In order to provide more flexibility, Reflective Java allows end-users to make only some of the operations of a class become reflective. Thus, non-reflective operations of a reflective object will maintain their original behaviour, and will not suffer any performance penalty due to the indirection imposed by reflection.

Metaobjects are usually used to address system issues. Thus, it would be ideal to implement them in an application-independent fashion. However to make metaobjects effective, some information about the application classes need to be passed to the metaobject. For instance, in the case of implementing a metaobject for concurrency control, information about whether an operation is read-only would help the metaobject to decide what kind of lock should be used. Without this information, the metaobject can only apply write locks for each operation; thus losing concurrency to a great extent. Generally speaking, the more application information that is available in a metaobject, the more effective it becomes. On the other hand, if too much application information is revealed, the metaobject becomes too application dependent, and lose its generality. A balance must be achieved.

To resolve this, Reflective Java supports the concept of method category so that a metaobject can perform different meta-actions for different categories. In the concurrency control metaobject example, we can define read-only operations as category *READ*, and update operations as category *WRITE*. In this way, although the metaobject does not know which operation from which object will use it, it can still provide appropriate locks.

Currently, we are extending the Java BeanBox [13] through which users can bind an application class with a metaclass by simply drawing a line between them. When an application class is selected, all its public operations will be displayed in a window. Then a user can specify which operations would be reflective by highlighting them. A user can also input a category number for each selected operation. Afterwards, the graphical tool will generate the reflection class automatically according to user' s choice.

A REFLECTIVE COMPONENT-BASED TRANSACTION ARCHITECTURE

We have developed a transaction architecture based on Reflective Java to meet the needs of Internet transaction applications. The architecture is component-based, thus enables users to develop portable, customisable components, and assemble them into applications. It enables rapid application development and deployment using standard components and off-the-shelf tools. The architecture supports implicit transactions, removing from developers any concern for transaction management details. Any component installed on the server is a candidate for participation in a transaction. No component in a transaction need concern itself with the behaviour of other components in regards to their effect on the transaction.

Our architecture tracks closely the Enterprise JavaBeans specification [14], and can be used to execute Enterprise JavaBeans. More importantly, and unique in our approach, the architecture is also reflective, providing additional features that are important for supporting Internet transaction processing. By supporting reflection our architecture allows a server component container to be easily customised, for example, in order to cater for new application demands, or to adapt to a new environment. It also allows application developers to provide application-specific information, declaratively and separately from application code, to the container so that the latter can make use of the information to improve system performance. This

is in direct contrast to the "one size fits all" approach of EJB architecture.

The architecture consists of six kinds of entities (Figure 2): the server component, application information script, server component container, client component, metaobjets, and underlying transaction system. The server components implement the business logic for an application. The container provides transaction and persistence capabilities, as well as management and control services to the encapsulated server components.

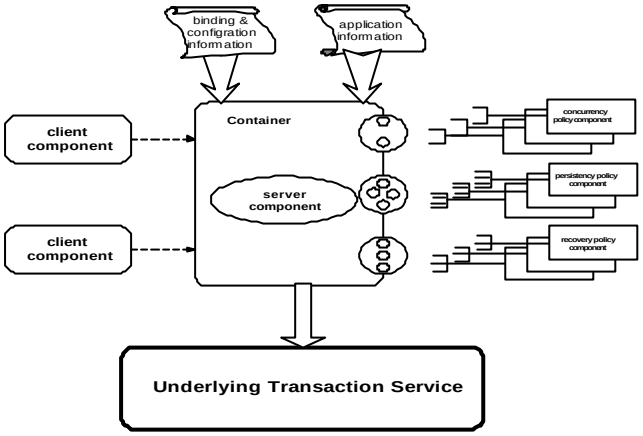

Figure 2: A reflective component-based transaction architecture

In order to provide these services transparently to both the client and server components, we use Reflective Java to enable the container to intercept all operations made on the server components within it. Each time a client component invokes a method on a server component, the request goes through the container before being delegated to the target server component. The container can thereby implement capabilities such as concurrency control, security and transaction management transparently.

A container also insulates server components from the underlying infrastructure. The container automatically allocates system resources on behalf of the components and manages all interactions between the components and the underlying system. This ensures that the server components can be run in any system as long as it supports a compatible sever component container.

To provide flexibility and adaptability, a container represents implementation strategies in the form of metaobjects. A metaobject can be replaced by another metaobject that implements the same functionality, but with a different strategy. In each server component container there are a number of "sockets" for plugging in metaobjects. Users can choose "off-the-shelf" metaobjects that are best suited to their applications at deploy time. They can also supply their own metaobjects, if they like. Metaobjects can also been changed dynamically at runtime to cater for changing environment conditions.

Server components are built by using a component builder. Through the builder, users can manipulate and customise a server component through its property tables and customisation methods. Users can also assemble a server component with other components to create a new application. Furthermore, they can also attach some component-specific information with the component, such as concurrency semantics, deploying policy, and concurrency policy. This information is understandable and usable by the server component container.

The runtime structure of a container is shown is Figure 3. At runtime, for each active server component, the container maintains a context object to record its information, and a number of metaobjects to implement corresponding functionality, such as concurrency control and security checking. A reflection object will interact with the context object that in turn will interact with the corresponding metaobjects at particular points to enforce transaction and security rules.

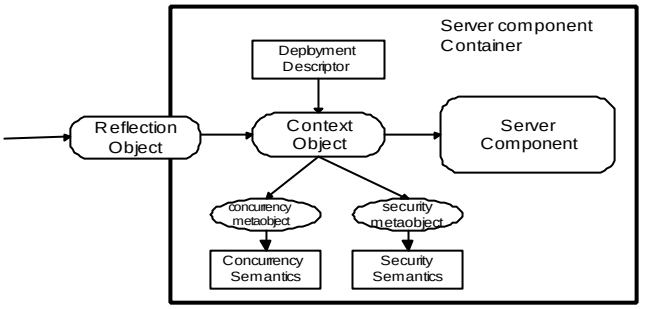

Figure 3: The structure of a container

To enable containers to utilise component-specific information to improve system performance, users can provide this information through scripts at assembly time. The container will ensure this information to be used by corresponding metaobjects. For example, the concurrency control metaobject would use the concurrency semantics of a component to increase concurrency degree.

CONSTRUCTING METAOBJECTS

Although Reflective Java provides great potential flexibility for application users to customise system behaviour via metaobjects, it is not easy to construct metaobjects straightaway. Particularly when developing a system that intends to provide multiple system capabilities, for example, concurrency control and persistency in our case. Therefore, we used a two-layer structure for building metaobjects. We implemented a first-layer metaobject, called *TransactionalMetaobject* that is a standard metaobject as specified in Reflective Java. However, instead of providing the concurrency control and persistency capability directly inside the *TransactionalMetaobject*, we implement them in two second-layer metaobjects respectively. The *TransactionalMetaobject* will invoke operations defined in

the second-layer metaobjects to perform concurrency control and persistency at appropriate time.

Generally speaking, the first-layer metaobject intercepts invocations from clients, and implements a general framework for providing a middleware function. However, instead of implementing the function directly inside the first-layer metaobject, it is divided into a number of smaller tasks that are implemented in second-layer metaobjects. The first-layer metaobject is responsible for coordinating the second-layer metaobjects to fulfill the whole function.

For providing flexibility and adaptability, each task can be implemented in a number of ways in different metaobjects, thus users can customise a function to suit their application by using the most appropriate metaobject. For example, fault tolerance can be realised via replication so that failures on an object can be recovered by using its replicas. Alternatively, fault tolerance can be achieved through transactions. A transaction ensures that either all or none of its operations are executed. If a transaction is interrupted by a failure, its partial results are undone.

In order to make sure all the metaobjects that implement the same task are compatible, an interface should be defined for each task which must be implemented by all metaobjects implementing that task. A first-layer metaobject may only access second-layer metaobjects that implement the interface, and an access must be done via operations defined in that interface. Therefore, the interface works as a contract between the first-layer metaobject and the second layer metaobjects. In this way, first-layer and second-layer metaobjects can be developed independently without any mismatch risk. This means users can choose third-party metaobjects freely for their purpose. This also means that a metaobject can be reused to fulfill the same functionality in other systems.

For our transaction architecture, we have built one first-layer metaobject, *i.e. TransactionalMetaobject*, and defined two second-layer metaobject interfaces for concurrency control and data persistency: *Concurrency* and *Persistence*. We also developed two second-layer metaobjects, *TplConcurrency* and *LazyPersistency*, for implementing the two interfaces respectively.

THE TRANSACTION MODEL

Our transaction model is based on the OMG's Object Transaction Service (OTS) specification [10]. It is a well-defined transaction model, bringing the transaction paradigm and the object paradigm together. A major advantage of the model is that it enables every object to provide its own concurrency control and recovery, thus providing the possibility for an object to apply an individual concurrency control and recovery policy to cater for its specific requirements. However, this advantage is not exploited fully in the OMG's specification. The reflection functionality of our architecture provides the right tool for exploiting this advantage.

Our transaction model is object-oriented, rather than database-oriented. In a database-oriented model, the container focuses mainly on robust messaging. It is the database system that is responsible for concurrency control, recovery and persistence. This approach makes it easy to leverage existing database systems and transaction processing monitors to Internet applications. However, most database systems deal with concurrency based on file or records rather than objects. This makes impossible for them to utilise application semantics to improve concurrency control, and hence system performance. It also means that all components stored in a database system can only use the concurrency control method provided by the database system, whenever whether or not it is suitable for their applications.

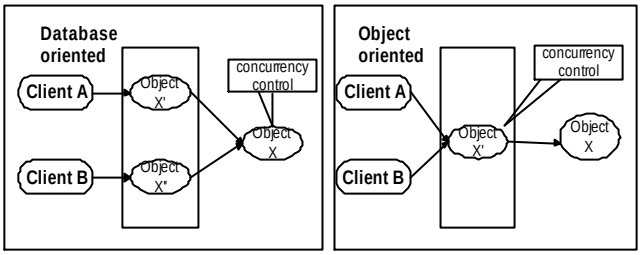

Figure 4: Database oriented *vs.* object oriented transactions

Another drawback of the database-oriented model is that it keeps unnecessary copies of components in memory. For example as shown in Figure 4, if two clients access a component *X* through a server concurrently, there would be two copies of *X* inside a container, each for one client. This wastes system resources and increases interactions to the database system.

By taking the object-oriented transaction approach, our architecture enables users to choose the best suitable concurrency control method for their application. The server component container automatically uses the selected concurrency control method to implement the transaction services.

RELATED WORK AND SUMMARY

Our work is related to three research fields: component-based software, reflection and transaction server systems. Reflection has been used in many application areas: flexible programming [5, 9], distributed systems [2], concurrent programming [8], and fault tolerant applications [3]. Using reflection as a general approach to implementing non-functional requirements has been discussed in [11]. Some research results [12] have also shown the feasibility of using reflection to implement middleware. However, the lack of a common framework for integrating metaobjects created by different parties into one application make it impossible for them to be used in multiple applications. Therefore, the

potential of the reflection technology to provide flexibility and portability for system middleware is not fully exploited.

Component-based software techniques have become popular because of their emphasis on modularity, re-usability, reliability, and their ability to function in a network environment. To address the requirements of middle-tier development, component-based server systems have emerged. Microsoft has retooled its existing component model into ActiveX. ActiveX provides the glue for components to communicate with each other. The server side ActiveX components are executed under Microsoft Transaction Server (MTS) [6] that performs any application processing. MTS handles all the management of sharing, processes, and threads. MTS also automatically manages the transactional behaviour of server components.

Another component-based transaction server is Sybase' s Jaguar CTS (Component Transaction Server) [4]. The Jaguar CTS Transaction Manager hides virtually all the complexity of transaction management and coordination from application developers with "implicit transactions." The Jaguar CTS Transaction Manager automatically manages transaction boundaries and ensures transactional consistency across all transactional components and the underlying DBMS.

Enterprise JavaBeans (EJB) [13] from JavaSoft is an extension of JavaBeans for handling middle-tier/server side transactional business applications. The EJB architecture places EJB components on top of the EJB Executive. The Executive, in turn, gives EJB beans access to APIs, remote objects, and transaction services. Part of the attraction of the EJB architecture is that the developer does not need to know about Java interface definition language, multithreading, or security, the Executive runtime abstracts APIs and remote object calls. EJB does not support a full transaction-server environment, thus needs some kinds of transactionally aware execution environment, such as a transaction processing server or database engine.

Although there are some differences between these technologies, they are all component-based software, thus provide portability, scaleability and flexibility to the server components. However, a common weakness of them is the lack of addressing the issue about how to make the container itself flexible and customisable. We believe that reflection can play an important role here.

Our transaction architecture combines the advantages of the reflection and component-based software. Like the above transaction servers and Enterprise JavaBeans, our work aims to meet the requirements of transactional business applications by providing a component architecture for the middle-tier/server side. Moreover, our architecture also defines a standard structure for the server component, which enables a container to change its functionality by plugging/unplugging its metaobjects. This makes it easy for a container to be customised to meet new application requirements or changing environment conditions. Furthermore, our architecture allows application developers to provide component-specific information, like concurrency semantics, to containers so that the information can be used at runtime to improve server performance.

Generally speaking, our transaction framework has demonstrated the benefits and feasibility of Reflective Java in developing middleware and server applications. By combining with the component technology and the two-layer approach for constructing metaobjects, the flexibility provided by reflection can be used in an appropriate, powerful and controlled way.

REFERENCES

1. S. Chiba: A Metaobject Protocol for C++. In Proceedings of the 10th Conference on Object-oriented Programming, pages 483-501, 1993.
2. S. Chiba and T. Masuda: Designing an extensible distributed language with meta-level architecture. In Proceedings of the 7th European Conference on Object-Oriented Programming, pages 482--501, 1993.
3. J. Fabre, V. Nicornette, T. Perennou, R. J. Stroud, and Z. Wu: Implementing Fault Tolerant Applications using Reflective Object-Oriented Programming. Proceedings of the 25th IEEE Symposium on Fault Tolerant Computing Systems, 1995.
4. Jaguar: Java Components and Transactions. Byte, February 1998
5. G. Kiczales, J. des Rivieres, and D. G. Bobrow: The Art of the Metaobject Protocol. MIT Press, 1991.
6. Microsoft Transaction Server White Paper. <http://www.microsoft.com/transaction/>
7. P. Maes: Concepts and experiments in computational reflection. In OOPSLA '87 Proceedings, pages 147--155, October 1987.
8. S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In Proceedings of the European Conference on Object-Oriented Programming '91, pages 213--250, July 1991.
9. J. McAffer: Meta-Level Programming with CodA. In proceedings of the 9th European Conference on Object-Oriented Programming, pages 190-214, 1995.
10. OMG: Object Transaction Service. OMG document 94.8.4, August 1994
11. R. J. Stroud: Transparency and reflection in distributed systems. ACM Operating Systems Review, 27(2): 99--103, April 1993.

12. R. J. Stroud, and Z. Wu: Using Metaobject Protocol to Implement Atomic Data Types; In proceedings of the 9th European Conference on Object-Oriented Programming, pages 168-189, 1995.

13. Sun Microsystems: Enterprise JavaBeans™. V1.0 <http://www.javasoft.com/products/ejb/docs.html>

14. The Beans Development Kit <http://java.sun.com/beans/software>