APM

POSEIDON HOUSE • CASTLE PARK • CAMBRIDGE CB3 0RD UNITED KINGDOM

+44 1223 568930 • Fax +44 1223 359779 • Email: Zhixue.Wu@ansa.co.uk • URL: http://www.ansa.co.uk

ANSA Phase III

A Reflective Component-Based Transaction Architecture (A Full Paper for Middleware '98)

Zhixue Wu

Abstract

The trend in Internet applications is to move from browsing to real-time transactions with business critical information. The characteristics of Internet applications requires system architectures that are scaleable, flexible and adaptable, whilst still easy to use, to develop and to deploy. The "three-tier" architecture is now emerging to address the needs of Internet applications in terms of scaleability and dynamic access. In a three-tier architecture, most of an application's logic is moved from the client to one or more servers in the middle tier. This makes the middle tier the single most critical component of the emerging Internet application architecture from a developer's point of view.

In this paper we advocate a reflective component-based transaction architecture. Using components enables users to assemble portable, customisable components into applications. Reflection provides two additional features that are important for supporting Internet transaction processing. First, it enables the transaction infrastructure 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 transaction infrastructure to best suit the application component, or at execution time for improving system performance. The architecture supports implicit transactions, thus removes from the developers any concern for transaction management details.

APM.2118.00.01 | **Draft / Approved** External | 02 March 1998

Distribution:
Supersedes:
Superseded by:

TABLE OF CONTENTS

1 INTRODUCTION

The trend in Internet applications is to move from browsing to real-time transactions with business critical information. Examples include online banking, order/entry, and customer service. Transactions have been used in online processing on mainframe systems for many years. However, there are some distinguishing characteristics of Internet applications, which change the way in which transactional applications are constructed and deployed.

Thin clients. In traditional client/server computing, an application-specific client needs to be pre-installed on the user machine to run an application. In Web-based Internet applications, the runtime components are downloaded from the Web site. Such a thin client model delivers two key benefits: universal access, and reduced installation and management costs.

Scale. Unlike traditional applications, Internet user communities can extend well beyond department or company. With these new "self-service" applications, access to a server becomes open to thousands of users all executing transactions simultaneously. This requires highly scaleable server architectures to support transactional applications.

Rapid development. Many corporations have started using Internet for publishing and collecting information, and intend to doing business over the Internet. Therefore, the technology for building Internet application systems must be very easy to use, develop and deploy.

All these characteristics of Internet applications requires system architectures that are scaleable, flexible and adaptable, whilst still easy to use, to develop and to deploy. “Three-tier” or “multi-tier” architectures are now emerging to address the needs of Internet applications in terms of scalability and dynamic access. In a three-tier architecture, most of an application’s logic is moved from the client to one or more servers in the middle tier. This provides a number of benefits. Server components can be replicated and distributed across many servers, to boost system availability. Server components can be easily modified to adapt to changing business rules and economic conditions, thus providing flexibility. Server components are also location independent, if they are built using distributed objects (e.g. CORBA), therefore system administrators can easily reconfigure system load.

In this new model, users find and launch applications on HTML pages at Web servers. Instead of simply loading a static page, a dynamic "applet" is downloaded to the individual’s browser. The applet bring with it protocols that allow the applet to communicate directly to application servers running in the middle tier. These servers access data from one or more databases, apply business rules, and return results to the client applet for display. This makes the middle tier the single most critical component of the emerging Internet application architecture from a developer's point of view.

There are three popular architectures for building middle tier components: CORBA-style Object Request Brokers (ORBs), Transaction Processing Monitors, Web Application Servers. Although each has its strengths, none of them is ideally suited for the middle tier requirement of Internet transaction processing.

CORBA ORBs [1] have excellent multi-tier capabilities with strong distributed object invocation and related infrastructure services such as transactions and security. Unfortunately, the complexity of the overall solution and a lack of strong tool support limits their appeal to sophisticated developers. Additionally, most ORBs also have primitive server-side execution engines, limiting performance and scaleability.

TP monitors, on the other hand, have robust and mature execution engines that deliver excellent performance and scaleability. However, like ORBs, their overall complexity and proprietary APIs often make them difficult to use and expensive to install, administer, and maintain.

Web Application Server technology emerged in an attempt to transform Web servers to application servers. Web Application Servers are generally customer generated from one of several Web or site development tools. This strong tools focus leads to high developer productivity. On the flip side, scaleability is severely limited by the application server's direct tie to Web servers and the lack of non-HTTP protocols for application-to-application communication.

To address the need for a scaleable and easy-to-use middle tier, component-based transaction servers are emerging, such as Sybase's Jaguar CTS [2], and Microsoft Transaction Server [3]. They combine the best features of ORBs and TP monitors with component-based develop tools. This enables quick creation of scaleable applications. Component-based transaction servers offer built-in transaction management capabilities, and support distributed object invocation for multi-tier application communication. They also support rapid middle tier development and provide an execution environment for server components.

In this paper we advocate a reflective component-based transaction architecture. Like Jaguar CTS and MTS, it 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 in the server is a candidate for participation in a transaction. More importantly, no component in a transaction need concern itself with the behaviour of other components in regards to their effect on the transaction.

Unlike Jaguar CTS or MTS, the architecture is also reflective, providing two additional features that are important for supporting Internet transaction processing. First, it enables the transaction infrastructure 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 transaction infrastructure to best suit the application component, or at execution time for improving system performance. Our architecture tracks closely the Enterprise JavaBeans specification [4], and can be used to execute Enterprise JavaBeans.

2 A REFLECTIVE COMPONENT-BASED TRANSACTION ARCHITECTURE

In this section, we present our reflective component-based transaction architecture after a brief introduction to the component model and the reflection technology.

2.1 Component-based software

In recent years, constructing applications through the assembly of re-usable software components has emerged as a highly productive way to develop custom applications.

The term component-based software is used to describe a software model, which specifies how to develop reusable software components and how these component objects can communicate with each other. A component is an encapsulated piece of code that can be combined with other components and with handwritten code to rapidly produce a custom application. A component is designed to be used within another application, called a container, and designed to be reused and customised without access to its source code. A container provides an application context for components and provides management and control services to the component it stores.

In order to qualify as a component, the application code must provide a standard interface that enables other parts of the application to invoke its functions and to access and manipulate the data within the component. This is often termed “introspection” and enable the application developer to make full use of the component without requiring access to its source code. Components can be customised to suit the specific requirements of an application through a set of external property values, often called “reflection”.

Server components are application components that run on a server. As we discussed in Section 1, in a three-tier architecture, most of an application’s logic is partitioned into separate server components to be deployed on a server system. A server component container provides a runtime environment to support the execution of server components. In our case we combine the robust runtime features of a traditional transaction processing (TP) monitor with the flexibility and reusability features of distributed components. The container provides the complex management services that are required to support high-volume business transactions, including multithreading, resource-pooling, and transaction co-ordination. The introspection and reflection facilities allow the container to take design time and runtime policy on how to couple its servers to application components.

2.2 Reflection and metaobject protocol

Reflection [5] is the capability of a computational system to reason about and act upon itself. Unlike conventional system, a reflective system allows users to perform computation on the system itself in the same manner as in the application, thus providing users with the ability to adjust the behaviour of the system to suit their particular needs.

In an object-oriented programming environment, reflection can be realised in the form of metaobjects that represent some internal information and implementation of the system. The interfaces of these metaobjects are called *metaobject protocols* (MOPs) [6], because they allow application objects to communicate with metaobjects. Through MOPs, users can modify the systems' behaviour and implementation incrementally.

Using metaobject protocols, the actual behaviour of an application object is determined not only by itself, but also by the metaobject which it is associated with. The association can be thought of in terms of a binding between the application object and the metaobject. An application object can obtain the capability of a metaobject by binding to it. In this way, the functionality of an application is determined by its application objects, whilst the quality of application delivery is determined by the associated metaobjects. The quality of application delivery can be changed through alternative metaobjects without making changes to application objects. This makes it possible to provide system capabilities to an application program transparently and flexibly.

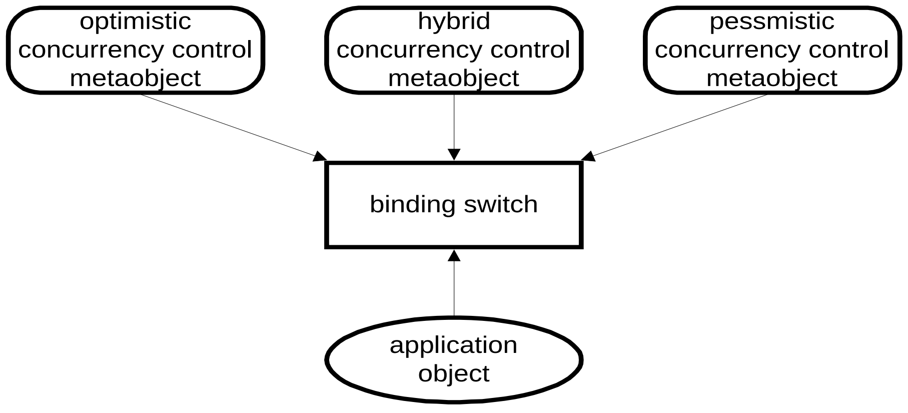
optimistic concurrency control metaobject
hybrid concurrency control metaobject
pessmistic concurrency control metaobject
binding switch
application object

Figure 2.1 The metaobject protocol approach

For example as illustrated in Figure 2.1, an application object can become usable in a concurrent environment by binding to one of the concurrency control metaobjects. There is no need to make any change to the application object. A binding between an application object and a metaobject can be changed dynamically according to the run-time conditions. For example, when the conflict rate of concurrently accessing an application object is low, it would be better to bind it to an optimistic concurrency control metaobject. However, when the conflict rate becomes high, the binding can be switched to a pessimistic concurrency control metaobject. By monitoring its components and applications, the system can perform this switching automatically without disturbing application programs.

2.3 The computation model

The characteristics of Internet applications requires system architectures that must be scaleable, flexible and adaptive, whilst still be easy to use, to develop and to deploy. Our reflective component-based architecture meets the rigors of Internet applications by taking advantage of both the reflection technology and the component model.

To achieve the most benefits from the multi-tier architecture, server components should be implemented as shared servers. However, building a shared server is not an easy task. It is much harder than building a single-user application. Usually, shared servers need to support concurrent users, and they need to share system resources, such as threads, memory, and network connections. They also need to participant distributed transactions and enforce security policies.

It would be very hard for application developers, who are experts in business logic, but not necessarily in transaction monitor engineering, to address all these system issues. To solve this problem and to provide portability to application programs, our computation model allows a clear separation to be made between business logic and system issues. This enables application program to focus on application requirements without concern about the system issues. The separation also makes it possible for an application program to be executed in different system environment without making changes to its source code.

Because of the wide range of potential applications, with varying needs, it is impossible to provide a single monolithic application server infrastructure suitable for all applications. The implementation of an application server's infrastructure must be flexible and adaptive so that it can be customised easily to cater for a particular application. To achieve this aim, our infrastructure represents alternative infrastructure choices as alternative metaobjects, as shown in Figure 2.2. Thus at deployment time, an application assembler can choose the most suitable infrastructure for his application by selecting the corresponding metaobject. The selected metaobject is then integrated with the application object to form a server component.

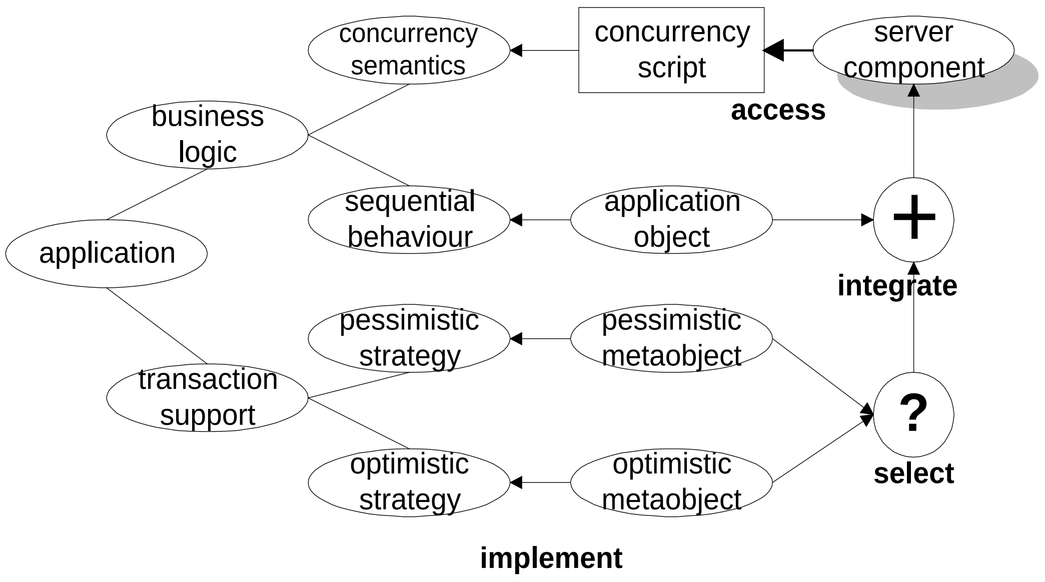

Figure 2.2 A reflective computation model

A clear separation between the application program and the implementation of system issues is essential for making application component portable. However, without any information about the application, it would be hard for a system to provide a good quality of service to that application. Only when detailed knowledge about an application is available such as ordering constraints for consistency, it is possible for a system to optimise its behaviour and to improve its performance. We solve this dilemma by allowing application developers to specify application-specific information declaratively and separately from application programs. In such a way, we enable application information available to a system, but without increasing the burden of an application developer, nor losing the portability of an application program.

For a transaction system, the concurrency semantics of an application can be used to reduce delay due to blocking. Thus, we allow concurrency semantics to be represented in our model, but it is represented declaratively and separated from the sequential behaviour of an application. The sequential behaviour is implemented in application code; whilst the concurrency semantics are represented as a concurrency script. At runtime, the concurrency semantics would be used by the transaction strategies to schedule operations.

2.4 Concurrency semantics

By taking into consideration type-specific semantics of operations, a transaction system can allow concurrent executions that would otherwise be forbidden if operations were simply characterised as *reads* and *writes*. The concurrent semantics of operations are usually represented by relationships between operations, such as commutativity[7]. Given two operations on the same type, *p* and *q*, we say that they are commute if the result of executing *p* and then *q* on *d* is the same as the result of executing *q* and then *p* on *d*.

Consider, for example, a bank account class, *Account*. It has an associated set of operations: *credit* money to an account, *debit* money from an account, and *check* the balance of an account. In this example, two *credit* operations are commute, so do two *check* operations. Therefore, it is possible for two *credit* operations to be executed concurrently. However, this execution would not be permitted in a system that classified operations into *reads* and *writes* only. This shows a general phenomenon: by taking into consideration type-specific information, a system can permit greater concurrency than would otherwise be possible.

However, utilising concurrency semantics is not an easy task. It would makes the application program complicated, if the concurrency code is intertwined with the implementation of business logic. In order to avoid this, we allow application users to expressing concurrency semantics decaratively and separately from the implementation of the objects.

The concurrency semantics can be represented easily in pairs of operation names. For example, a pair *(p, q)* means that operation *p* and *q* are commutative. In judging whether two operations are commutative, one needs only to consider the logical relationship of the two operations, not the implementation of the operations. Our transaction system will ensure the atomicity of individual operations.

2.5 The architecture

There are six kinds of entities in our architecture (see Figure 2.3): the server component, application information script, server component container, client component, metaobjets, and underlying supporting system. The server components implement the business logic for an application.

A server component container provides certain system capabilities, such as multithreading, transaction and security. It also provides an application context, management and control service to the encapsulated server components. To provide flexibility and adaptability, a server component container represents implementation strategies in the form of metaobjects. A metaobject can be replaced by a new 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.

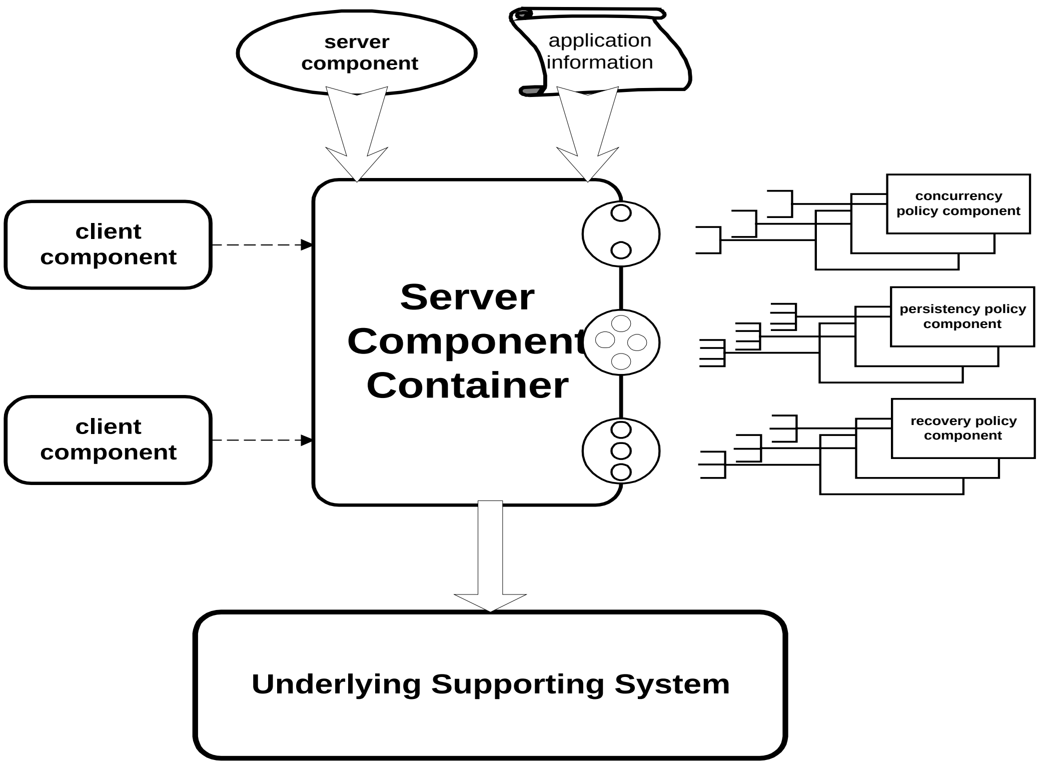
server component
application information
client component
client component
Server Component Container
concurrency policy component
persistency policy component
recovery policy component
Underlying Supporting System

Figure 2.3 The reflective component-based transaction architecture

A server component container insulates server components from the underlying supporting system. 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.

A server component container maintains control over a server component through a wrapper. A container provides an external representation of a server component. Client components do not directly interact with a server component, but with the external representation. This allows the container to intercept all operations made on the inside server components. 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 implements system capabilities, such as concurrency control, security and transaction management transparently. The behaviour of the server component container are decided partly by the associated metaobjects.

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. These information are understandable and usable by the server component container.

2.6 Structure of the server component container

In this section, we present the structure of the server component container. As we described in the last section, a server component container manages a server component via intercepting invocations to the component. The interception is implemented through our reflection system. For each server component, a reflection object is generated which provides a client view of the server component. Clients access a server component through interacting with the corresponding reflection object.

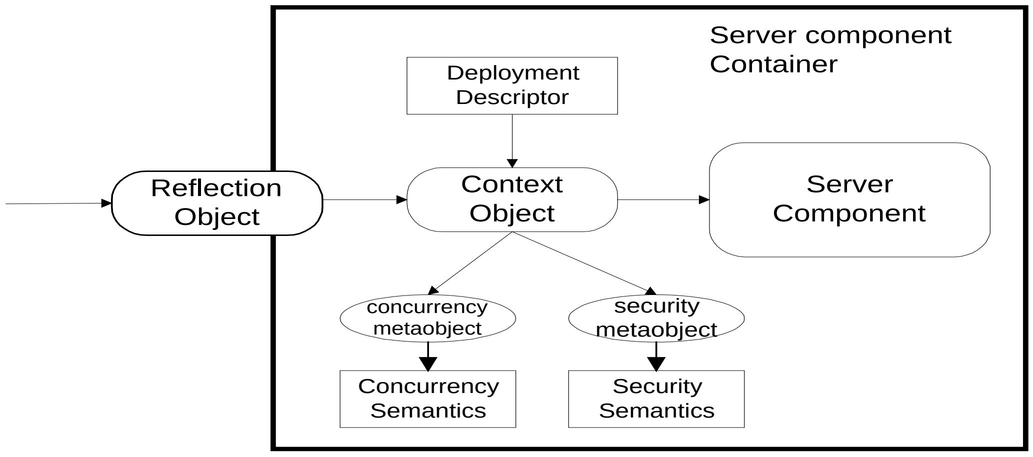

Figure 2.4 Server component container

For each active server component, the server component container generates a context object to maintain its information, and a number of metaobjects to implement corresponding functionality, such as concurrency control and security checking. The 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.

To enable containers to utilising component-specific information to improve system performance, users can provide these information through scripts at assembly time (see Section 3 for detail). The container will ensure these 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.

2.7 The transaction model

Our transaction model is based on the OMG's Object Transaction Service (OTS) specification [8]. 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.

OTS supports distributed transactions that can span multiple databases on multiple systems co-ordinated by multiple transaction managers via a distributed two-phase-commitment protocol. Therefore, by using OTS our architecture ensures that a transaction of a server component can inter-operate with other server component servers.

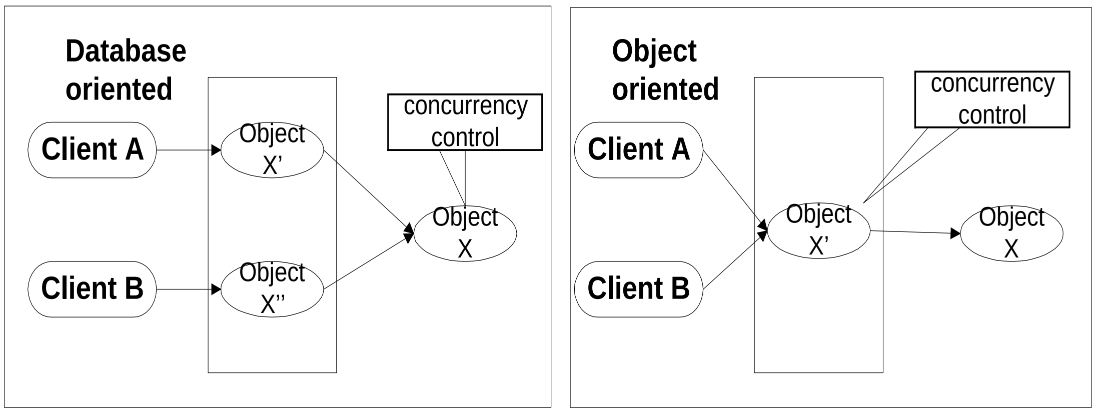

Figure 2.5 Database oriented and object oriented transactions

Our transaction model is object-oriented, rather than database-oriented. In a database-oriented model, the system component 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.

Another drawback of the database-oriented model is that it keeps unnecessary copies of components in memory. For example, if two clients access a component *C* through a server concurrently, there would be two copies of *C* 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. We also enable users to change the concurrency control method of a server component dynamically at runtime to cater for environment changes.

3 DEVELOPMENT, ASSMBLY AND CUSTOMISATION OF SERVER COMPONENTS

In this section, we present the development, deployment and assembly of server components. Particularly, we focus on the aspects of supporting component-specific concurrency control and free choice of concurrency control methods, which are unique to our work.

3.1 Component development

As described earlier, by taking a reflective component-based approach for implementing application servers, our architecture enable a clear separation to be made between the code to implement business logic and the code to implement system issues. Thus the server components need only focus on implementing the business logic without any concern about the environment where the components will be deployed. This makes the server component much easier to implement. Users can programming based on the assumption that the components would be used only by a single user and that there is no system failures. The pure responsibility of the components is to ensure the correctness of the business logic in a sequential and reliable environment. The server component container will provide functionality of concurrency control, transactions and security implicitly.

We follow the design pattern of Java Beans so that a server component can be customised via any building tool supporting the Java Beans specification. We are not going to discuss this issue here. Interested readers can refer to the specification of Java Beans [9] for details.

Although our server component container provides additional features beyond those defined in the Enterprise JavaBeans (EJB) specification. We do not imply any special requirements on the implementation code of a server component. As long as it fulfil the requirements of the EJB specification, it can be deployed in our specialised container. However, we do require application users to provide the concurrency semantics of a component at deployment time in order for it to enjoy the benefits of type-specific concurrency control.

3.2 Assembling and customising a component

A server component can be manipulated in our visual build tool, an extension of the BeanBox from the BDK [10]. It maintains all the original functionality, but gains some new features to meet our special requirements.

We have build a general container (Reflection Frame) and a specialised container (Transaction Frame). When starting the Extended BeanBox, the Transaction Frame will appear on the ToolBox as well as all the registered components. Instantiating a Transaction Frame in the BeanBox can be done by clicking on it and then on the BeanBox area. A component can be instantiated in the same way. For an instantiated frame or component, there will be a box with its name appearing on the BeanBox area.

A server component is put inside a Transaction Frame, by first selecting the Transaction Frame and then clicking the reflection item in the edit menu. When a *red line* from the frame appears, a click on the server component will connect it to the Transaction Frame.

To provide flexibility, we make it possible for a component to expose only a subset of its operations for the outside world. This provides the component assembler with the power to remove from the external interface some operations who thinks are improper or unnecessary to be exposed. To support this feature, the extended BeanBox displays a list of the public operations of a component when it is selected. The assembler then can choose the operations to be exposed by clicking on the corresponding items. Other components can only access this component via these chosen operations.

After the decision has been made on what operations are to be exposed, a reflection class will be automatically generated for the component. The reflection class provides an interface which includes those chosen operations. Each of these operations will have the same signature as the corresponding operation of the component. The reflection class provides a client's view to the server component.

At this stage, the component assembler can also, as in a normal BeanBox, customise other properties of a component to cater for the particular application. The assembler can also use the event properties of a component to establish connections with other components to form a service.

3.3 Selecting concurrency method and providing concurrency semantics

As we described earlier, we enable an application to choose a concurrency control method that is most suitable for its requirements. This is done by providing multiple concurrency control methods via a set of concurrency metaobjects. To ensure a seamless connection between a concurrency metaobject and the container, both of them need to confirm to a contract. A concurrency metaobject must implement the *ConcurrencyMetaobject* interface. This ensures that a concurrency metaobject provides all the necessary operations for concurrency control. On the other hand, the container must invoke these operations at proper time to enforce concurrency control.

A concurrency metaobject needs to describe its concurrency strategy in a special property. When it is instantiated in the BeanBox, that property will be displayed, so that an assembler can understand the strategy and to decide whether it is suitable for a particular application. After choosing a particular concurrency control method, an assembler connects it to the container. This will ensure both the BeanBox and the container can find the necessary information about the concurrency control method.

Now, it is time to provide the concurrency semantics of a component to the container. As we discussed earlier, the concurrency semantics can be represented by relationships between operations. The default one is the commutativity relationship, but a metaobject can use other kinds of relationship for this purpose. In the case a relationship rather than the commutativity is used, the concurrency metaobject is required to provide the definition of the relationship in a special property. When the concurrency metaobject, is selected, this information will be displayed so that the assembler can understand the relationship and represent the concurrency semantics accordingly.

When the commutativity relationship is used, the container will interpret the specification provided by an assembler, translate it into a standard form, and store it into a place the metaobject can access. In the case that an other relationship is used for representing concurrency semantics, it will be the metaobject that is responsible for interpreting and translating the specification. The metaobject must provide a method for this purpose for the container to invoke.

4 RELATED WORK

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 [6, 11], concurrent programming [12], distributed systems[13, 14], and fault tolerant applications [15]. Using reflection as a general approach to implementing non-functional requirements has been discussed in [16]. Some research results [17] have also shown the feasibility of using reflection to implement system 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. One of the most successful component product has been Borland's Delphi [18]; however, the first one to become widespread use was Microsoft's Visual Basic [19]. JavaSoft came into the field with a component architecture called JavaBeans [9]. The distinguishing feature of Java Beans as reusable components is that they interoperate across all platforms supported by Java. Since the Java Virtual Machine runs on a variety of platforms, a Java component, affectionately called a "bean" can be used on any platform.

To address the requirements of middle-tier development, component-based server systems have emerged. Microsoft has retooled its existing component model into ActiveX [20]. ActiveX provides the glue for components to communicate with each other, regardless of their implementation language and platform. The server side ActiveX components are executed under Microsoft Transaction Server (MTS) [3] that performs any application processing. MTS handles all the management of sharing, processes, and threads. All the components that make up an application can share these resources. Consequently, using MTS may actually improve performance compared to stand-alone execution, in both time and memory. MTS also automatically manages the transactional behaviour of server components. A programmer can make an ActiveX component transactional by setting a transactional property.

Another component-based transaction server is Sybase's Jaguar CTS (Component Transaction Server) [2]. The Jaguar CTS Transaction Manager hides virtually all the complexity of transaction management and coordination from application developers with "implicit transactions." With implicit transactions management, developers at deployment time simply specify whether or not a component is transactional. At runtime, the Jaguar CTS Transaction Manager automatically manages transaction boundaries and ensures transactional consistency across all transactional components and the underlying DBMS.

The Enterprise JavaBeans (EJB) [4] 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 work, 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. The architecture we presented at this paper 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. Moreover, 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. We enable the information to be specified declaretively and separately from application code.

Our work started before the emerging of the Enterprise JavaBeans and we are now aligning some of our design and implementation to it so that our server component container can be used as a specialised Enterprise JavaBeans container. This ensures all EJB beans can be executed in our container. However, we provide the flexibility for customising containers to cater for application requirements and the power for using application information to improve system performance. Therefore, our work is an extension to Enterprise JavaBeans, rather than a competitor. Or in the terms of EJB, a specialised container.

5 SUMMARY

The characteristics of Internet applications requires system architectures that are scaleable, flexible and adaptive, whilst still easy to use, to develop and to deploy. In this paper we have presented a reflective component-based transaction architecture that takes advantage of the reflection and component technologies. Like most component models, it 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. Moreover, unique to our architecture, by supporting reflection it 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. We also allow application developers to provide application-specific information, declaritively and separately from application code, to the container so that the latter can make use of the information to improve system performance.

The work on component-based software is now more focus on how to make server components on the middle-tier easy to develop, to deploy and to manage. We think it is also very important to make the execution environment of the components, i.e. the container, flexible and customisable. In the other front, although we believe that a clear separation between business logic and system issues is a key to the success of component-based software, we also believe that application-specific information could play an important role for improving system performance. Our work has contributed some ideas in these aspects.

We have completed our design stage, and a builder tool is available for developers to assemble and customise server components, and to specify and provide concurrency semantics. The implementation of the server component container has been underway and the first prototyping should be completed before the end of this year.

6 REFERENCE

1. OMG: CORBA 2.2/IIOP Specification. OMG Technical Document formal/98-02-01.

2. Jaguar: Java Components and Transactions. Byte, February 1998

3. Microsoft Transaction Server White Paper. <http://www.microsoft.com/transaction/learn/mtswp.htm>

4. Enterprise JavaBeans*TM*. <http://www.javasoft.com:80/products/ejb/docs.html>

5. P. Maes: Concepts and experiments in computational reflection. In OOPSLA '87 Proceedings, pages 147--155, October 1987.

6. G. Kiczales, J. des Rivieres, and D. G. Bobrow: The Art of the Metaobject Proto- col. MIT Press, 1991.

7. W. E. Weihl and B. Liskov: Implementation of resilient, atomic data types. ACM Transactions on Programming Languages and Systems, 7(2):244--269, April 1985.

8. OMG: Object Transaction Service. OMG document 94.8.4, August 1994.

9. The JavaBeans*TM* Specification <http://java.sun.com/beans/docs/spec.html

10. The Beans Development Kit <http://java.sun.com/beans/software/bdk_download.html>

11. S. Chiba: A Metaobject Protocol for C++. In Proceedings of the 10th Conference on Object-oriented Programming, pages 483-501, 1993.

12. 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.

13. 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.

14. J. McAffer: Meta-Level Programming with CodA. In proceedings of the 9th European Conference on Object-Oriented Programming, pages 190-214, 1995.

15. J. Fabre, V. Nicornette, T. Perennou, R. J. Stroud, and Z. Wu: Implementing Falut Tolerant Applications using Reflective Object-Oriented Programming. Proceedings of the 25th IEEE Symposium on Fault Tolerant Computing Systems, 1995.

16. R. J. Stroud: Transparency and reflection in distributed systems. ACM Operating Systems Review, 27(2):99--103, April 1993.

17. 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.

18. Delphi: <http://www.borland.com/delphi/papers>

19. Visual Basic: <http://www.microsoft.com/vbasic/controls>

20. ActiveX Tutorial <http://www.microsoft.com/intdev/activex/tutorial>