Aspectizing Middleware Platforms*

Charles Zhang, Hans-Arno Jacobsen
Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto
10 King's College Circle
Toronto, Ontario, Canada
{czhang,jacobsen}@eecg.toronto.edu

ABSTRACT

Over the past decade, middleware platforms such as DCOM, CORBA, and .NET, have become increasingly popular, addressing software engineering problems for distributed application development. Often, the same functional middleware platform model has also been applied to application domains with varying requirements. CORBA, for example, aims at supporting distributed enterprise, real-time, embedded systems, and high performance computing applications. Consequently, the architecture of middleware systems has also evolved drastically to accommodate a large number of new design requirements from a broad range of application domains. Those new design requirements include, for instance, transaction support, security, fault tolerance, real time, various performance enhancements, and many more. Although those additional features have made middleware platforms more mature in terms of functionality; they have, at the same time, made middleware systems more structurally sophisticated, expensive to run and difficult to evolve further. We therefore strongly believe that new architectural methodologies should be applied to the current middleware design to provide solutions to those problems. This paper employs the methodology of aspect oriented programming to analyze the architecture of middleware system, by using the openly specified middleware platform standard, CORBA, as a case study. We use AspectJ, a Java based aspect language, to illustrate that certain features in CORBA, such as fault tolerance and interceptor support, which are not possible to be properly modularized using traditional programming paradigm, can actually be modularized using aspect oriented programming. We then show that, by applying AOP techniques, we can factor in and out pervasive characteristics of middleware systems and, thus, make the architecture more modularized and more customizable.

*University of Toronto Technical Report CSRG-466 January 2003

Keywords

Aspect Oriented Programming, Middleware Architecture, Software Architecture

1. INTRODUCTION

Middleware platforms, such as CORBA, DCOM, J2EE, and .NET, have provided abstraction and simplicity for the complex and heterogeneous computing environment. They facilitate the development of high quality distributed applications with shorter development cycle and much smaller coding effort. Middleware systems are being adopted in a very broad spectrum of application domains, ranging from traditional enterprise platforms to mobile devices, embedded systems, real time systems, and mission critical systems.

The problem with today's middleware architectures is that they cannot simultaneously satisfy multiple design requirements in order to support a large variety of target domains. On one hand, implementations that are conglomerations of a large number of features become increasingly sophisticated in module compositions and more dependent on powerful processors and vast memory spaces. On the other hand, although the common distributed computing requirements do not change, there exist many flavors of middleware implementations. Each of the flavors incorporates different design alternatives and engineering tradeoffs to optimize the performance for specific domains, which are characterized by stringent resource constraints, execution deadlines, high performance, and high availability. However, it is still difficult to harness the complexity of the middleware platform and tailor middleware for a specific user need, a concrete usage scenario, or a particular deployment and runtime instance. Current middleware architectures lack of methods to address common distribution concerns and particular domain requirements without incurring great architectural complexity and performance penalty.

Recent research, such as OpenCOM [15] and DynamicTAO [4], addresses those issues by introducing new software engineering techniques like component based architecture and reflection to establish better architectural abstractions. Those techniques achieve higher levels of modularity and customizability. However, they are also insufficient in addressing some other design concerns. For instance, although DynamicTAO addresses configurability, it has limited ways of achieving small memory foot print. OpenCOM addresses adaptability by using meta frameworks, which introduces performance overhead.

We believe that today's distributed computing environment requires an even higher degree of modularity for middleware architectures. That high level of modularity is very hard to obtain via traditional architectural methodologies. In this paper, we examine middleware architectures from a new perspective by applying aspect oriented design and development methodologies to the requirements domain and the corresponding functional decomposition domain. We think the design challenges created by the dramatic evolution of middleware platforms can be well addressed by the aspect oriented approaches, because it enables the separation of pervasive design concerns and allows us to optimize solutions for particular design goals without coordinating with others. Via aspect oriented techniques, we then are able to include or exclude new features with minimum or even no changes to the existing architecture. The deployment of middleware systems can be more flexible, customizable, and adaptive in terms of code size, memory footprint and resource utilization.

In this paper, we focus on developing aspect oriented middleware architecture methodologies and make the following contributions: 1. We present the definition of orthogonal requirements and the method of horizontal decomposition, which addresses tangled concerns introduced by orthogonal design requirements, which include independent and conflicting design requirements. 2. We provide the aspect oriented analysis to the architecture of middleware and define a classification method of aspects in middleware. 3. We provide the evaluation of the horizontal decomposition method through the aspect oriented implementation of several aspects of middleware, using CORBA as a case study. 4. We list a few lessons learned that can serve as guidance for the aspect oriented analysis and design of middleware.

The rest of the paper is organized as the following: We first present more detailed discussion of the problems of middleware architecture in section 2. We then briefly introduces the aspect oriented programming techniques in the context of middleware design in section 3 . Section 4 conceptually discusses how AOP can be applied to the middleware architecture. In light of that, section 5 provides a case study of those concepts, which includes the AOP implementation of several aspects of CORBA. Related work is summarized in section 6. And section 7 concludes the paper.

2. PROBLEMS OF MIDDLEWARE ARCHITECTURE

2.1 Evolving to the unmanageable

A prominent challenge to architecture of middleware systems is that, in recent years, the spectrum of target platforms is being greatly broadened from traditional enterprise systems to mobile devices, network devices, control units and other platforms. The characteristics of those platforms, referred to as the emerging application domains, differ from each other significantly. For example, middleware systems are used on the Cisco ONS 15454 optical transport platform to deal with hardware customizations and the communications between management software and hardware drivers . Middleware systems are also adopted by the US Navy as the software bus for subunits in the submarine combat control systems [14]. Those types of platforms have introduced new design requirements and new features to middleware systems, such as high performance, real time, high availability, handling expensive memory resource and limited computing power.

Although the functionality of middleware platforms has come to a great maturity, the traditional methodology for architecting middleware platforms exhibits a lot of inherent limitations which has made middleware system relatively complex to develop and to deploy. Those limitations can be shown by using CORBA as an example.

The Common Object Request Broker Architecture (CORBA) [9] is an open specification defined by the Object Management Group to unify and to standardize key elements of the middleware architecture. The original design goal of CORBA, more specifically the Object Request Broker, was to "provide interoperability between applications in heterogeneous distributed environments and seamlessly interconnect multiple object systems." [7] The distributed environment today mandates CORBA to carry more responsibilities. Examples of those responsibilities include transaction support, security, high performance, small memory footprint, fault tolerance support, real time computing, high performance, and more. The enrichment of features has made the architecture of CORBA extremely sophisticated in terms of its module compositions.

We illustrate that problem by inspecting historical releases of JacOrb , an open source CORBA implementation in Java. The earlier release of JacOrb implements the CORBA 2.0 specification with BOA(basic object adaptor), ACL(access control list) based security and proprietary interceptor implementation. The latest release is a significant evolution with a CORBA 2.4 implementation of POA (portable object adaptor), portable interceptor and the full range of CORBA services. Figure 1 plots chronicle releases of JacOrb against the number of files, which roughly equals to the number of Java classes, and the size of the code in thousand line of code (KLOC). It shows that in JacOrb the number of modules has increased around 50%, and the lines of code tripled during a development period of approximately four years . The trend undoubtedly points to a direction of ever growing development size and maintenance task.

In addition to the increasing size, the typical runtime of middleware platforms also requires more and more computing resources, such as CPU, memory and network resources. That has become the main challenge for applying middleware systems, such as CORBA, to platforms with stringent resource constraints, despite the fact that middleware systems are greatly needed on those platforms. For instance, in the context of wireless mobile computing, industries have realized that middleware can provide an abstraction to the underlying network detail, and decouple the application logic from the complex, error prone low level details including optimization of data packaging and the compression of transmission[1]. That makes mobile applications more portable,

[1] Joey Caron, Scott Herscher, Ann Marie O'Connor *CORBA in the palm of your hand whitepaper* Vertel Corporation

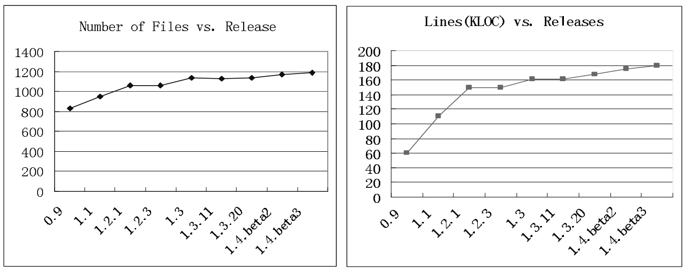

Figure 1: Evolution of CORBA: The JacOrb example

faster to develop, and easier to evolve. Unfortunately, unless specifically designed, the runtime cost of most of the middleware infrastructures found today is too high to justify the benefit of middleware.

Another problem is that there has been a proliferation of middleware specifications to accommodate different requirements that come from many application domains. In the case of CORBA, OMG defines Real Time CORBA specification in order to address execution predictability, Fault Tolerant CORBA specification for high availability and Minimum CORBA for embedded platforms. The Java platform exhibits the same syndrome of having multiple editions targeting different application domains. Microsoft .NET platform is moving into the same direction. Those specifications bring challenges to vendors, who must architect the system differently to comply with particular specifications while maintaining platform compatibility and functional consistency. The new adopters often find the platform hard to comprehend and to use since it is not always easy to match a particular specification with user specific deployment settings.

Furthermore, in traditional middleware architectures, many systematic properties are not implemented in modules. At the same time, not all of these systematic properties need to participate in middleware operations for every application domain, deployment instance, or runtime condition. For example, the property of thread-safeness could lose it applicability if the middleware is deployed on a platform only supporting a single thread of execution. And the support for security might not be needed for middleware deployed on internal networks for application integration. Losing of modularity greatly hinders the adaptability and the configurability of middleware platforms.

2.2 The AOP alternative

Aspect oriented programming (AOP), a new software engineering methodology, can help us to address those problems of middleware architecture design. We introduce, complementing the method of "vertical decomposition" the idea of "horizontal decomposition", which can be considered as a superimposition method. We can use aspect oriented techniques to "horizontally" compose or to "superimpose" the implementation for orthogonal design requirements onto the existing architecture without modifying the existing architecture.

In the following sections, we first introduce AOP in the context of middleware architecture. We then discuss how to apply AOP methods to the design of highly modular middleware architectures.

3. ASPECT ORIENTED PROGRAMMING

Aspect oriented programming provides an alternative design paradigm that achieves a very high degree of the *separation of concerns* in software development. The primary purpose of AOP is to liberate developers from coordinating with different and potentially conflicting sets of systematic properties. A systematic property can be treated as a different purpose of the system other than its primary operational logic. Examples of such properties can be found in [6]. AOP overcomes the limitations of traditional programming paradigms by providing language level facilities to modularize those systematic properties as an independent development activity. The AOP compiler is capable of producing the final system by merging the aspect modules and the primary functionalities together. To understand AOP further, we first look at the definition of crosscutting concerns and aspect.

3.1 Crosscutting concern

A concern, in the context of software engineering, can be typically identified as a purpose, a property, a concept or a design goal. The problem of crosscutting concerns arises when "two properties being programmed must compose differently and yet be coordinated." [12]. Those two properties are said to cross-cut each other.

The crosscutting phenomenon is quite common in middleware platforms. For example, one of the primary design concerns of the ORB is to transparently enable invocations on remote objects through the marshalling and unmarshalling mechanism. In ORBacus[2], an open source industrial CORBA implementation, the steps involved in remote invocations can be exemplified by the sequence diagram in Figure 2.

This design satisfies the OMG specification and works efficiently. However, it is quite possible that the remote objects can sometimes be deployed or migrate into the client machine or even the client process. Therefore, the design goal, in the case of in-process server objects, is that invocations should largely cost the same as a normal method call. The design should avoid socket communications and the marshalling/unmarshalling work. It is easy to observe that those two concerns, location transparency and optimization of in-process invocations, require very different designs and dis-

[2]ORBacus http://www.iona.com/products/orbacus_home.htm

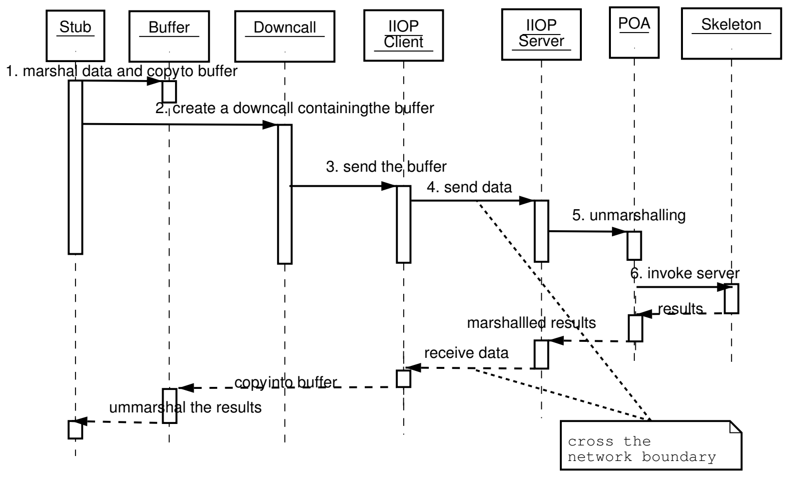

Figure 2: Remote Invocation: ORBacus

tinct decomposition models. Conventional decomposition methods cannot implement both requirements in separate modular forms simultaneously. The result is a decomposition model with tangled logic. That can be illustrated by the ORBacus implementation of dynamic invocation in Figure 3. Figure 3 shows that, in order to maintain object transparency and also to be efficient in invoking in-process server objects, the actual implementation changes the picture in Figure 2 by introducing the collocated servers and the collocated clients. Collocation in ORBacus means in-process. Step 3 in Figure 2 is replaced by a call to `POAManager`, typically a server-side object, to check if the target exists in the same process. If it does, the `Downcall` object simply copies the data to the buffer of the collocated server. Otherwise, the data is sent through the IIOP client to TCP/IP sockets.

There are several drawbacks of this implementation. Firstly, it adds multiple execution paths to Figure 2 and involves server objects, such as POA manager, in the client side downcall process. Secondly, for server objects located on the same host but not in the same process, the ORBacus implementation still uses TCP/IP based sockets, which is not efficient for inter-process communications. Adding corresponding support will complicate the picture even more.

From the example above, we say that the concern for object location transparency and the concern for optimization of local invocations crosscut each other. It is an example of satisfying conflicting design requirements. The result of the crosscutting is the tangled logic as in Figure 3. What we desire in this case is that the mechanism of making optimized local invocations does not break the modularity of the remote object invocation mechanism. We then have opportunities of optimizing the performance of both mechanisms. That can be accomplished using aspect oriented programming.

3.2 AOP Artifacts

"Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways." [12] The existence of aspects is attributed to handling crosscutting concerns via the traditional "vertical" decomposition paradigms. Aspect oriented programming methodology can help us to separate the tangled concerns from each other and to compose them independently. That is particularly beneficial to the middleware design. To address crosscutting concerns such as in the location transparency problem, we can utilize the following artifacts provided by aspect oriented programming techniques.
a. *A component language.* A component language is used for performing the primary decomposition. ORBacus can be treated as the component program written in the Java component language.

b. *An aspect language.* The aspect language defines logic units that can be used to compose aspects into modules. Representative aspect languages are AspectJ [1] and HyperJ [2]. We can use those languages to implement crosscutting concerns, such as the local invocation optimization in our example.

c. *Aspect weaver.* The responsibility of an aspect weaver is to instrument the component program with aspect programs to produce a final system. In our case, the implementations of both remote and local invocation mechanisms can be defined separately and coexist in the final "woven" system.

There are a number of aspect oriented languages. AspectJ[3] uses Java as the component language. The AspectJ compiler is the aspect weaver that weaves the aspect program back into the component program on the source level or on the byte code level. The produced system is a regular java program, which can be compiled by a regular java compiler or executed by a regular JVM. In addition to Java language features, AspectJ defines a set of new language constructs to model the aspects. A `joinpoint` represents an interception point in the execution flow of the component program. For convenience and elegance, a `pointcut` construct can be used to denote a collection of joint points. Actions can be triggered before, after or in place of the program execution when a joint point is reached. Those actions are defined using

[3]AspectJ `http://www.aspectj.org`

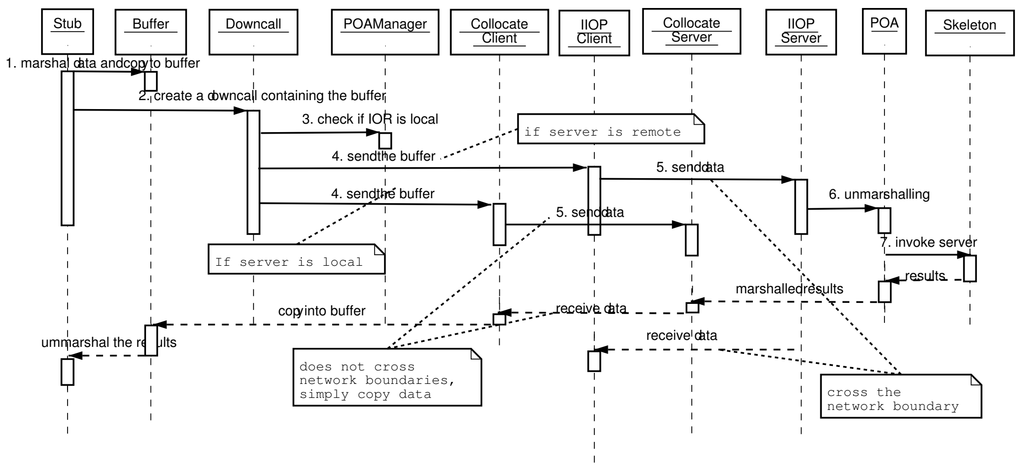

Figure 3: Addressing Remote and Local Invocations Simultaneously in ORBacus

`advices`. An aspect module in AspectJ contains `pointcuts` and the associated `advices`.

HyperJ[4] supports multi-dimensional programming by allowing programmers to compose the system differently according to specific concerns in Java. the HyperJ compiler performs bytecode transformations to generate different final systems according to extraction specifications. Each extraction is analogously termed as "hyperslicing".

4. APPLYING AOP TO MIDDLEWARE ARCHITECTURE

We observe that the high degree of orthogonality among the design requirements of middleware is one of the main causes to the problems with today's middleware architecture. Orthogonality between two requirements means that two requirements can be implemented independently without the need to coordinating with one another. Two requirements could be independent or conflicting. The goal of satisfying multiple orthogonal design requirements simultaneously cannot be handled adequately by the traditional software decomposition paradigms due to the problems of the dominant decomposition model [16]. The modularity designed to address a particular set of requirements will eventually get broken by trying to address other existing orthogonal requirements, and especially the new ones incorporated at later times.

In the context of middleware design where the orthogonality among the design requirements arises very often, we believe AOP appropriately fits in producing unprecedented degree of modularity and customizability comparing to conventional techniques. The first step of applying the aspect oriented methodology is to clearly understand from the definition of aspects the primary functional decomposition of middleware systems and the pervasive properties. Those properties crosscut the basic functionality of middleware and therefore can be considered as middleware aspects.

4.1 Defining The Aspects Of Middleware

[4]HyperJ `http://www.alphaworks.ibm.com/tech/hyperj`

To correctly identify aspects, we need to use the primary decomposition model as the reference to discover and to evaluate the orthogonality of design goals. It is then possible to observe, in the primary model, the crosscutting properties, which can be treated as aspects. We think that the fundamental functionality of a middleware system mainly consists of the following:
1. A standardized programming model or API that allows applications to make abstractions of the distributed objects or services.

2. The mechanism of publishing the representation of an object or a service to client programs.

3. The dispatching mechanism that forwards the requests associated with the published representation to its concrete instance.

4. The commonly agreed representation of data and operations on the network layer in order to exchange information with its remote counterparts.

To be more specific, Table 4.1 shows the architecture elements that fall into the categories listed above for popular middleware platforms, including CORBA, DCOM, Java RMI and .NET. Therefore, an aspect in middleware can be defined as the decomposition of a design requirement that is orthogonal to middleware's fundamental functionality, which includes all four mechanisms identified above. A design requirement is orthogonal if its implementation in the component language crosscuts the implementation of the fundamental functionality of middleware, which includes any of the four mechanisms. Using this definition, we discuss in further details the aspects of middleware in the following section.

4.2 Aspects of Middleware

As we have shown in the previous section, crosscutting concerns, or equivalently speaking, the presence of aspects, hinder the primary operational logic of middleware in many ways. It is unavoidable for software systems decomposed in traditional methods to suffer from the presence of aspects

	CORBA	*DCOM*	*.NET Web services*
Programming model	IDL	MIDL	C#,CLR languages
Identity Publication	IOR	OBJREF	WSDL File
Request Dispatching	POA	Service Control Manager	ASP.NET process [5]
On-wire representation	IIOP	Object RPC	SOAP

Table 1: Middleware Architecture Elmements

in terms of performance, maintenability, configurability, and more. Although AOP does not solve all those problems, it gives us the capability of modularizing aspects and having more opportunities to optimize the system under certain circumstances. To better understand how AOP would benefit the architecture of middleware, we group middleware aspects into three categories depending on their relationships with the primary functionality of middleware.
1. *Non-functional Aspects.* Those aspects are properties that relate to the maintenability or debuggability of middleware. Examples of such aspects are logging, tracing, coding rule enforcements, and others. Those design requirements are related to the human factors in software engineering which do not carry any operational purposes. However, those aspects could consume considerable computing resources and major development efforts. Traditional programming paradigms are not capable of modeling them as modules and of decoupling them entirely from the system.

2. *Augmentative Aspects.* Augmentative aspects are additional operational desgin requirements incorporated into middleware to support specific target platforms. Examples of such requirements are error handling, pre/post condition checking, transaction support, interception support, synchronization, high performance enabling, object persistency, fault tolerance, realtime characteristics, and many more. The common characteristic of those aspects is that they "augment" the primary functionality of middleware in serving its primary operational purpose in particular domains or on particular platforms. That also entails augmentative aspects are not necessarily beneficial to many other domains or platforms. However, traditional decomposition methods are not capable of flexibly separating them entirely out of the middleware architecture and adding them back only when necessary.

3. *Primary Aspects.* Primary aspects deal with the architectural choices about the essential functionality of middleware, e.g. the primary decomposition. Many of those choices are conflicting among each other as each design choice works the best in a different setting and requires a different set of high-level abstractions. There are several such conflicts in the CORBA architecture, such as the dynamic and static programming interfaces, object location transparency and local invocation optimization, and others. Traditionally decomposed models have no better ways but to pick one design alternative and to coordinate with other alternatives at the same time. The result is a degradation of modularity and performance for both design alternatives.

Through the categrization of the middleware aspects, we have a clearer picture of the different types of crosscutting concerns in middleware architectures. The problems of each type of aspects and how AOP would improve the quality of the architecture can then be better understood. Consequently, the decomposition method for middleware can be improved accordingly to incorporate AOP techniques. We define the aspect oriented decomposition process more concretely in the following section.

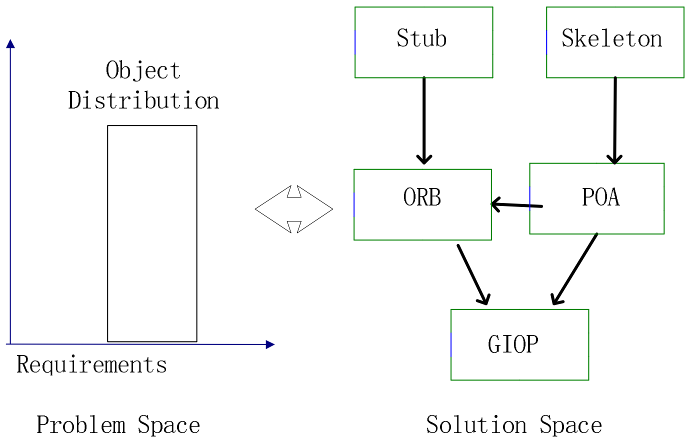

Figure 4: Prmary ORB decomposition

4.3 Aspect Oriented Decomposition of Middleware

Abstractly speaking, a decomposition paradigm is a mapping from the problem domain to the solution domain using data and operations. For instance, we use procedural decomposition to provide a solution to a given problem via data structures and functions. We can also use object oriented paradigm to decompose the same problem using objects. The model obtained in the solution domain, a calling tree of procedures or a graph of object dependencies, can be referred to as the primary decomposition model of that particular problem.

Using CORBA as an example, we identify that the key requirement of the ORB functionality is to provide object location transparency. the OMG specification can be viewed as an object oriented decomposition of that particular requirement which we define as the primary decomposition model of the ORB. To aid our discussion, we simplify that model to only include the following major building blocks: Stub, Skeleton, POA, ORB, GIOP. Figure 4 illustrates that transformation from the requirement space, consisting of one requirement, to the solution space, consisting of five components.

Conventional decomposition paradigms, either procedural or object oriented, achieve good modularity with a relative fixed set of requirements. That is because only after the problem is largely defined can we begin the process of architecture by making very high level abstractions as the first stage. That abstraction is usually in the forms of API interfaces or abstraction layers. The detailed solution, which is at the lower level below the abstraction, can be defined later by the techniques of late binding. Late binding associates the implementation to the declaration of the definition later than the compile time. It allows certain flexibility of the architecture because the detailed knowledge of the solution can be deferred till later stages. That top-down fashion of software architecture approach can be metaphorically referred to as "vertical decomposition"

However, we believe that the vertical decomposition approach cannot achieve good modularity when implementing multiple orthogonal design requirements simultaneously. That is because orthogonal design requirements produce different decomposition models with different high level abstractions. The decomposition model of the fundamental requirements of the system needs to be maintained consistently throughout the architecture process. Therefore, the implementation of requirements, which are orthogonal to the primary ones, needs to be compatible with and to be fitted into the primary decomposition model. That is referred to as the code tangling problem [12]or the drawback of the primary decomposition.

The case of satisfying multiple orthogonal design requirements happens often in middleware systems. For instance, although the primary functionality of CORBA, or more specifically the object request broker(ORB), is to facilitate transparent invocation of remote objects, in certain domains, particularly enterprise computing domains, it is often mandatory for CORBA to provide services like security, transaction, or fine-grained object access control as found in ACL-based or role-based security schemes [3]. Typically, those services are implemented through the support of interceptors [9] in the ORB. The implementation of interceptors requires changes not to one or two classes but rather systematically spanning many major functional areas in the architecture of the ORB. Another example of such orthogonal design requirements is the support for the dynamic programming interface. As illustrated in the Figure 5, the implementation for the requirements of supporting interceptors and the dynamic programming interface are scattered throughout the primary model for object distribution. Since those two properties or features crosscut the basic functionality of ORB, we can refer to them as two aspects of the ORB. Later sections will provide detailed analysis of the aspects of the ORB.

The aspect oriented programming methodology can help us re-architect middleware platforms in two ways. We can extract those scattered properties from the tangled model and compose them separately as aspect programs. Weaving back the aspect programs should yield the same system functionality as before. Secondly, new orthogonal requirements can be satisfied by weaving new features, which are implemented as aspect programs, into the existing architecture. Figure 6 depicts the concept of aspect decomposition and aspect weaving using CORBA as an example. Comparing to the diagram in Figure 5, not only is the modularity of

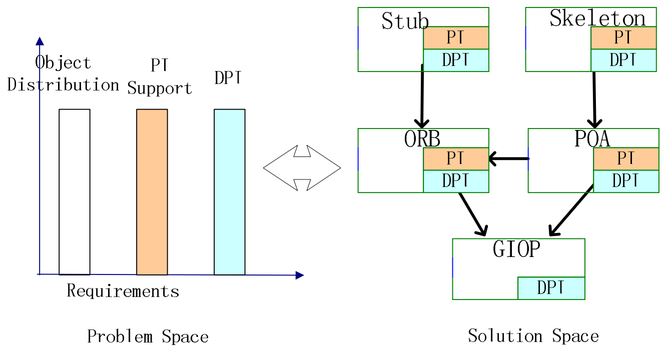

Figure 5: Decomposition of orthogonal requirements

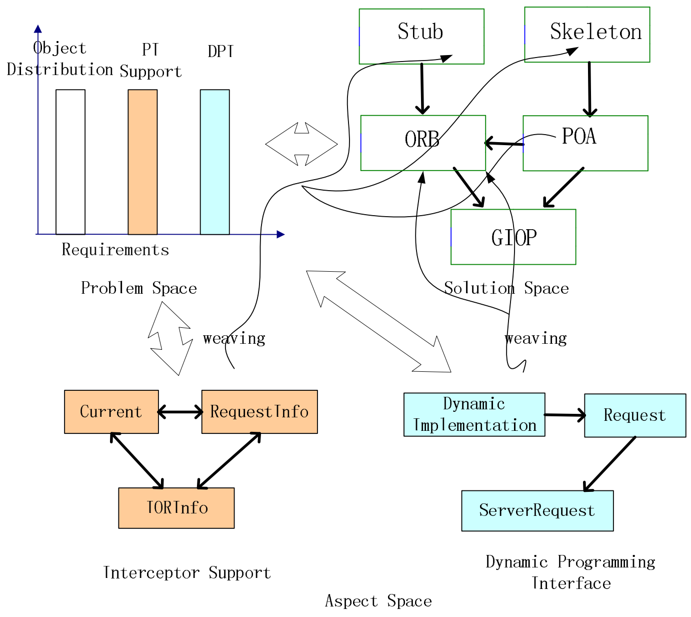

Figure 6: Aspect Oriented Decomposition for ORB

the original components preserved, the composition for interceptor support and dynamic programming interface also become modular.

The advantage of the aspect oriented approach in middleware architecture is multi-fold compared to the conventional approach. First of all, decomposing middleware requirements along both the vertical dimension and the horizontal dimension promotes a very high degree of specialization in the architecture. That is, architects and developers can now focus their attention on solving particular problems. The artifacts obtained at the end of such architectural activities are multiple abstraction models with one primary decomposition and multiple aspect decompositions. Each decomposition model focus on a specific orthogonal design requirement. That separation of development concerns dramatically reduces the development effort and errors due to the narrowing of the problem scope and the modularization of aspects. Domain expertise can be better applied.

Secondly, horizontal decomposition promotes the stableness and the openness of the architecture. In the case of traditional decomposed systems, an orthogonal design requirement typically brings changes not only to the implementational details but also to the abstractions made at the early design stage. When applying AOP principles, since each abstraction model is specialized to handle a much narrower scope of problems, it is much easier to make stable abstractions. Architectural techniques such as design patterns can help the architects to find solutions that are proven to be optimal and unlikely to evolve very rapidly. As a direct consequence, it becomes possible to "open up" the architecture and to allow third party to develop high quality code for a particular aspect. Taking CORBA as an example, if the architecture of the ORB's remote invocation mechanism can be made stable and more concretely specified, a third party is able to develop a specific synchronization library or a multi-functional logging library for the ORB.

Another advantage of development software in the superimposing fashion is that it makes software more adaptive and configurable not only "vertically" in the traditional dimension, but also "horizontally" in the dimension of aspects. That capability is extraordinarily attractive to middleware platforms because they support highly diversified platform types. This allows us, at least statically, to pre-configure the right set of systematic features to best suit a particular target platform. It also opens up the possibility to dynamically configure the middleware substrate according to the runtime information of the computing environment. For example, we can either statically or dynamically unload properties like exception handling or synchronization when the middleware substrate is deployed on an extremely resource-constrained platform. That also allows more opportunities of lower level optimizations such as the dynamic optimization techniques due to the decrease of runtime variants. Finally, from an economic point of view, the pricing model for aspect oriented middlewares can be tailored to match the exact user needs.

5. A CASE STUDY: RETROFITTING MIDDLEWARE WITH AOP

From the conceptual picture presented previously, we will now apply aspect oriented analysis and design to an existing middleware implementation to test our ideas. For system architectures that did not embrace aspect oriented concepts, the crosscutting phenomena in their implementations should exist inherently. This section explores the viability of aspect oriented approaches by analyzing some features that can be considered as aspects of CORBA. We then perform the aspect implementation to prove that those features like portable interceptors, dynamic programming interface, or certain fault tolerance operations can be composed as aspect programs in ORBacus.

5.1 Evaluation Metrics

The aspect oriented implementations should answer two major questions: 1. Can AOP at least preserve the fundamental functionality of middleware, in our case, ORBacus, in terms of its performance? 2. Can aspectizing certain orthogonal functionality lowers the complexity of the program structure for the primary decomposition? To anwser the first question, we verify the correctness of our aspect oriented implementation using the demo programs distributed with the ORBacus source code. We then collect the time taken to traverse the ORB stack by an integer CORBA message. We divide the time into four intervals: A. Client down call. B. Server up call. C. Server down call. D. Client up call. The data are collected as the average time in microseconds of 10,000 invocations on PIII 1G running with 2.4.19 Linux kernel.

To measure the change of complexity, we have employed four object oriented software metrics as described in [18]: *Cyclomatic complexity number(CCN)* is an heuristic index to measure the complexity of control flow in the program.*Coupling* measures the degree of interconnection between classes. *Weight of class* reflects how many methods are in a class. *Size* reports the total number of executable lines. Except *size*, all values are presented as the average per class.

5.2 Aspectizing Portable Interceptor Support

5.2.1 Analysis

"Portable Interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB." [9] They are observer [20] style entities that are invoked by the ORB at various points in the execution path. Interceptors allow a third party to plug in additional ORB functionalities such as transaction support and security, etc.

The case of interceptor is an example of a new orthogonal design requirement being added to the existing architecture. The specification for interceptors are added to the ORB architecture at a later time after the basic functionality of the ORB has been defined. The implementation of interceptor support crosscuts the implementation of the basic ORB functionality systematically as the following:
a. The CORBA::ORB interface contains methods and data members to allow the registration of interceptors. It also provides methods to allow access to those interceptors, when it is necessary to notify them upon reaching interception points.

b. During the propagation of the request, the invocation context is checked to see if it is modified by interceptors.

c. The POA needs to bundle requests with interceptors, if they are registered with the ORB.

Conceptually, the mechanism of portable interceptors is similar to that of AOP as they both base on interceptions of the execution flow of the program, a conventional solution, however, is not capable of modularizing the interactions among ORB objects and the interceptors summarized by the above list. The interaction code rather scatters around the ORB implementation. The code includes the initialization of interceptors, management of interceptors with upcall or downcall creation, notification of interception points, and more. Not only does the code introduce intermingled logic into ORB objects, it also introduces performance overhead and additional control paths because the extra code is always executed despite whether interceptors are used or not for a particular application.

In some domains, certain CORBA services that thrive on interceptors, such as transaction services and security service, are more applicable to enterprise application domains. Middleware platforms, being indiscriminate of any particular domain, should also cleanly support applications that are not willing to pay for the overhead of managing portable interceptors. By composing the interceptor support using AOP techniques, we can factor out the code dealing with portable interceptors, and weave the feature back into the ORB only if it is mandated by a particular application domain.

5.2.2 Implementation

We present a simplified version of our aspect implementation of interceptor support in AspectJ for the ORBacus ORB. The following aspect program performs two simple tasks. The ORBInitIS aspect initializes the interceptor initializers as specified in [9], and the POAIS aspect intervenes in the upcall creation of POA by creating a different UpCall with interceptors support. Figure 7 is the UML diagram of the relationship between aspects and primary models. Since UML has no specific notations for AOP idioms, we model `pointcut`s as attributes, and `before`,`after`, or `around` as methods. We omitted portions of the long signatures in AspectJ aspect definitions to make the diagram more concise. The diagram reflects the following facts:

a. The interceptor aspect for the ORB object is captured in the ORBIS aspect entity. Same as POAIS for POA object.

b. The `after` idiom indicates that some interceptor support code is executed after the execution of the bound method, namely, the instantiateORBInitializer method in ORB_Impl class. The `around` idiom indicates that the interceptor support code is executed on behalf of the bound method, which is the createUpcall method in POA_Impl class.

c. The directional association defines that aspects are oriented around primary objects. Primary objects are not aware of the existence of the aspect objects.

The code snippet 8 shows the aspect implementation for ORB_Impl. We first introduce a new attribute of `PIManager` to the singleton [20] instance of ORB. The PIManager is reponsible for notifying all interceptors of the inception points. The join point is defined as “tt Init”, which intercepts the method call of invoking all initializers in the ORB. (Line 9-17). The run time arguments, properties and logger, are exposed to be used later by advices. One advice is defined using the “after” idiom(Line 21), so that the interceptor specific initialization (Line 24-27) is executed after the initialization of other ORB initializers, designated by the `Init` joint point. Note that the advice body is omitted because it is merely an extraction from the original implementation in ORBacus. The role of this aspect module is to separate all ORB object related interceptor handling from the ORB object.

The code segment 9 shows the aspect module that collects all POA related interceptor support. A pointcut is defined to be the call to the regular Upcall creation method in POA. (Line 5). This example uses the *around* advice to suppress the original call of creating a regular upcall and to execute the advice, which creates a upcall that contains interceptor instances instead. (Line 20-31)

```
import java.util.*;                                        1
priviledged aspect ISOrbInit                               2
extends InterceptorSupport                                 3
{                                                          4
   //Add a new attribute to class ORBInstance              5
  private PIManager                                        6
        ORBInstance.interceptorManager_=null;              7
                                                           8
  //Define a weaving point when                            9
  //instantiateORBInitializer get invoked                 10
  pointcut Init(ORB_impl orb,                             11
          java.util.Properties properties,                12
          com.ooc.OB.Logger logger):                      13
  execution(private void                                  14
  ORB_impl.instantiateORBInitializers(                    15
  java.util.Properties, com.ooc.OB.Logger))               16
  &&target(orb)&&args(properties,logger);                 17
                                                          18
  //Define additional initialization logic performed      19
  //after the normal initialization                       20
  after(ORB_impl orb,Properties properties,               21
       com.ooc.OB.Logger logger):                         22
     Init(orb,properties,logger)                          23
   {                                                      24
     // The code is omitted. It is extracted from and     25
    // identical to the code in the original source       26
   }                                                      27
                                                          28
 }                                                        29
                                                          30
```

Figure 8: Interceptor Support: Initialization

The aspect implementation improves the efficiency of the original implementation. That is, for domains that do not require interceptors, application can enjoy simpler system structure, as certain classes such as `PIManager` needs not to be known to and loaded into the system. Consequently, the extra execution can be avoided as well.

5.2.3 Results and Evaluation

Table 2 shows a comparison of the original ORB, the aspectized ORB with the aspect of Portable Interceptors taken out. Table 3 shows a comparison of those metrics between

Interval	*A*	*B*	*C*	*D*
Original	78	8	42	118
Aspectized	79	9	42	122

Table 2: Response Time of Aspectizing PI

the original ORB and the aspectized ORB with the aspect taken out. The values are computed as the average per class, except *size*. We can conclude from the performance data and the structural changes that: 1. Aspect oriented implementation of supporting portable interceptors in ORB is able to preserve the original ORB functionality by correctly performing remote invocations and invocations of interceptors.

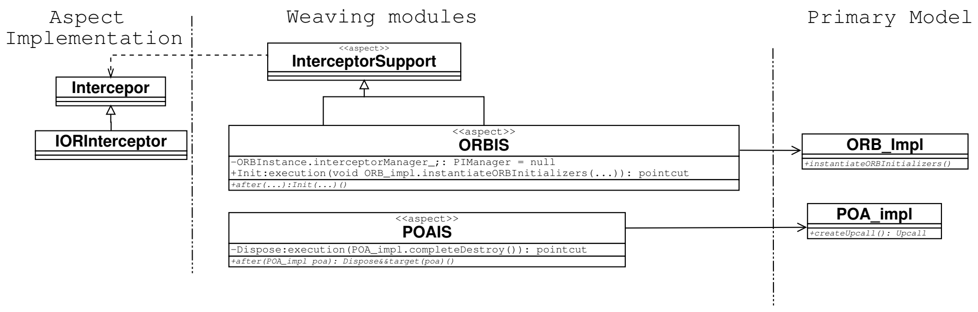
Aspect Implementation
Weaving modules
Primary Model
<<aspect>>
InterceptorSupport
Intercepor
IORInterceptor
<<aspect>>
ORBIS
-ORBInstance.interceptorManager_;: PIManager = null
+Init:execution(void ORB_impl.instantiateORBInitializers(...)): pointcut
+after(...):Init(...)()
ORB_Impl
+instantiateORBInitializers()
<<aspect>>
POAIS
-Dispose:execution(POA_impl.completeDestroy()): pointcut
+after(POA_impl poa): Dispose&&target(poa)()
POA_impl
+createUpcall(): Upcall

Figure 7: UML diagram for Interceptor Support Aspect

	CCN	*size*	*Weight*	*Coupling*
Original	4.11	3016	26.8	34.75
Aspectized	4.0	2909	26.7	32.38

Table 3: Structural Metrics of Aspectizing PI

2. The performance of aspectized ORB is equivalent to the original ORB. 3. The structural complexity of the primary model decreases as the result of the aspectization. 4. We are able to weave the aspect either in or out at the compile time which greatly enhances the configurability of ORBacus.

5.3 Aspectizing Dynamic Programming Model

5.3.1 Analysis

The dynamic invocation mechanism in an ORB is supported by the dynamic invocation interface (DII) and the dynamic skeleton interface (DSI), which are part of the CORBA specification. By using DII and DSI, an application can, during runtime, compose an invocation on an interface that it has no prior knowledge of.

IDL and its language mappings provide developers with a strongly typed programming model. Data are represented by their types and operations can be identified by their distinguished names and signatures. One of the advantages of using that programming model is that it imposes a rigorous and precise programming rule on developers to ensure the consistency and accuracy of their applications. That is why strongly typed programming languages are widely adopted.

However, a strongly typed programming model alone is not able to satisfy the requirements of some application domains. The reason is, a strongly typed programming model requires developers to have early knowledge of all the types and interfaces prior to the application development. ORB allows applications to be developed by different parties at different time. It is, therefore, extraordinarily hard to have that early knowledge because, in a collaborative computing environment such as what CORBA provides, it is a huge task to define all the types and interfaces, and to prevent them from evolving out of synch. Under certain circumstances, such as for application bridges that integrate multiple systems, the interface contracts simply do not exist. In these cases, static programming model is not applicable.

The dynamic properties supported by DII and DSI are independent of that of the static programming model. For the same design requirement, applications written using DII and DSI are composed very differently from using Stub and Skeleton based static programming model. Therefore, those two models are rarely used together. However, the ORB needs to coordinate the support for DII and DSI regardless in following ways:
a. The IDL language mappings, which are part of the programming interface of the ORB, need to support dynamic invocation. An example is the extraction operation from the Any type defined in the Java language mappings [17].

b. The CORBA::ORB interface contains operations that allow dynamical creation of invocation arguments and requests.

c. The request processing mechanism needs to support dynamic invocations including marshaling and unmarshaling of dynamically composed requests and their arguments.

Based on the above analysis, we conclude that the code for supporting DII and DSI is not modular and scattering around an architecture, of which the primary invocation model is static We argue that, since the dynamic programming model is orthogonal to the fundamental functionality of the ORB, it can be treated as an aspect of the ORB and implemented as aspect programs. We then are able to factor in and out dynamic programming model depending on available computing resources and target application requirements.

5.3.2 Implementation

The aspect oriented implementation of the dynamic programming interface for ORBacus consists of two parts: the dynamic invocation interface(DII) and the dynamic skeleton interface(DSI). For purpose of illustration, we present a simplified version of the implementation of DII. The UML diagram in Figure 10depicts the relationships among aspect implementations and the primary program.

The UML diagram shows two classes, namely `ORB_impl` and `Delegate`, are being manipulated by aspect modules, `ORB_implDII` and `DelegateDII`. The `after` idiom carries out extra initialization for DII after the normal initialization process completes. The `around` idiom replaces the bound method call to the `initialize` method of class `ORB_impl` with a new definition. The code snippet 11 shows how the

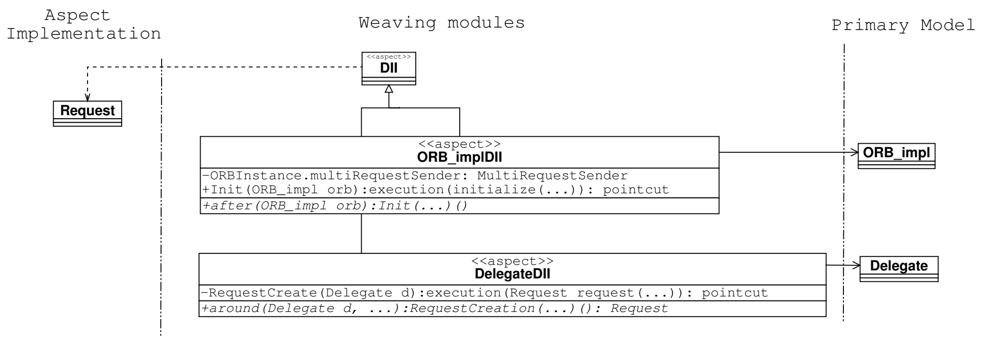
Aspect Implementation
Weaving modules
Primary Model
<<aspect>> DII
Request
<<aspect>> ORB_implDII
-ORBInstance.multiRequestSender: MultiRequestSender
+Init(ORB_impl orb):execution(initialize(...)): pointcut
+after(ORB_impl orb):Init(...)()
ORB_impl
<<aspect>> DelegateDII
-RequestCreate(Delegate d):execution(Request request(...)): pointcut
+around(Delegate d, ...):RequestCreation(...)(): Request
Delegate

Figure 10: UML Diagram for The Dynamic Invocation Interface Aspect

`after` idiom is implemented in AspectJ. This code performs two simple tasks: 1. It utilizes the *Introduction* capability of AspectJ to add an attribute of type `MultiRequestSender` to the class `ORBInstance` to handle multiply sent dynamic requests by the client side(Line 5). 2. It uses the `after` idiom to create a new instance of `MultiRequestSender` as the additional initialization performed after the normal initialization of ORB completes (Line 7-23). This implementation decreases the coupling of class `ORB_impl` because, if DII is not "woven" into the system, `MultiRequestSender` needs not to be known to the system and to be loaded. Extra class creation can be avoided as well. Code snippet 12 uses AspectJ to introduce a new method `request` in class `Delegate` to allow the creation of dynamically composed requests (line 4-5). This simple code shows how aspect oriented implementation can decrease the weight of the class by augmenting the class with additional interfaces only when it is necessary. There are a number of methods in class `Delegate` that are solely dealing with DII request. Separating them out into aspect modules has greatly simplified the complexity of the `Delegate` class while still preserving its DII capability.

5.3.3 Results and Evaluation

Table 4 presents the measurements, as in the case of portable interceptors, to examine changes in program structures and the response time for the aspectization of both DII and DSI. We used static invocation interface on the client side for the DSI measurements. Therefore, the client side process times, interval A and interval B, dramatically decrease as compared with DII. However, that change is irrelevant to our AOP implementation. The data show that, as in the case of portable interceptors, aspectizing the dynamic programming interface(DPI) has simplified the control flow and decreased the class size. The average weight of classes in the case of DSI dose not change because server side support for dynamic programming interface is much simpler. No additional methods are used to support DSI in the original implementation. The decrease of the weight of class is more visible in the case of DII due to aspectization of many dynamic request creation methods, as shown by the code snippet. The performance of the ORB with DPI factored out is again equivalent to the original ORB. We therefore conclude that aspectizing DPI simplifies the structure of the system and at least preserve the runtime performance.

5.4 Adding Fault tolerance as An Aspect

We have so far presented implementations of support for portable interceptors, an augmentative aspect, and the dynamic programming interface, a primary aspect. Aspect oriented decomposition can also be used to add a new feature to a legacy middleware implementation in a posteriori fashion without modifying its existing architecture. We follow the same methodology as in the previous aspectization work to discuss conceptually how certain fault tolerance features can be superimposed onto the ORBacus ORB architecture.

5.4.1 Analysis

The requirement for fault tolerance comes from domains that are running mission critical applications, such as public safety systems, medical support applications and avionic control systems. OMG defines the fault tolerant CORBA specification to provide standard interfaces and protocols in order to facilitate the replication of application objects. The fault tolerance services include factories for replicating objects, the fault detection and the notification mechanism, and the recovering mechanism.

Similar to the case of portable interceptors, the support for fault tolerance is also incorporated into the CORBA architecture at a much later time. Consequently, the semantics of the primary ORB objects needs to be augmented in order to support object redundancy. The fault tolerance is a pervasive property and can be regarded as an aspect because, in order to ensure transparency to application objects, fault tolerance support crosscuts the primary functionality of the ORB in the following ways:
a. The ORB must be able to handle additional information in the object reference. The replicas of CORBA objects are managed using groups by the fault tolerance services. Therefore, instead of IOR, fault tolerant CORBA objects use IGOR to publish their identities. IGOR is an extension to IOR by adding information about the group the server object belongs to.

b. The programming interface has additional semantics. It requires modifications to a number of operations of `CORBA::Object`, such as the deciding whether two server objects are equivalent. Those modifications mainly involve dealing with the group information that is added to the object identity.

c. The client side ORB must support transparent re-transmission mechanism to retry requests at alternative destinations, when the primary object in the replication group fails to process

	Structural Metrics				Runtime Interval			
	CCN	*Size*	*Weight*	*Coupling*	*A*	*B*	*C*	*D*
Dynamic Invocation Interface								
Original	5.08	1559	40	40.67	105	59	37	125
Aspectized	4.76	1490	37.67	39.33	108	59	35	124
Dynamic Skeleton Interface								
Original	3.64	274	9.33	14	79	8	43	126
Aspectized	3.46	262	9.33	13.5	76	9	41	119

Table 4: Metric Matrix for the aspectization of DII and DSI

the request due to a fault.

A desired implementation of fault tolerance should avoid making changes directly to the ORB architecture, because otherwise, like in the case of interceptors, it bundles features, which are not needed in application domains other than the ones that need high availability support. We believe fault tolerance support can be composed as aspect programs and should be superimposed onto the ORB, just like that of interceptor support discussed in the previous section. The advantage of doing so, in addition to preserving the ORB architecture, is that we are able to configure the feature in and out either at release time or at deployment time, depending on the requirements of the target application domain.

5.4.2 Implementation

We provide a partial implementation of fault tolerence functionality, which modifies the client side invocation mechanism to handle transparent re-transmission of requests if the primary server object is unable to process the request. We start by inspecting the client side downcall mechanism in ORBacus. The sequence diagram in Figure 13 shows a skimmed version of what happens when a client object writes a number of type long to a remote object. Call No.1 through 6 deal with setting up the invocation with a buffer, `OutputStream`, and the associated request information, represented by `Delegate`. The lower level involving the destination of the request and request sending process starts from call No.7. The `Downcall` object represents the communication details for corresponding server object.

To support transparent re-invocation, we first extend the `Downcall` class to create `RetryDowncall`, which differs from a regular downcall by having the connection information of an alternative server object in the same group, as illustrated by UML diagram in Figure 15. It also contains the special service context to designate the request as a retransmission, as defined in [8]. The modification to the downcall mechanism is shown by the code snippet 14.

The main idea in the code snippet 14 is to repeat sequence No.5 and on for alternative server objects in case of failures. We use the *introduction*(line 4-9) to add new method, which associates the buffer with a downcall containing an alternative server object in the same group. Line 11-15 defines a pointcut to the `invoke` method, which is sequence no.4 in Figure 13. The around advice(line 18-36) use the `proceed` mechanism to try the unmodified method call first. Upon catching a failure exception, the advice repeats sequences starting from No.5 by setting up a new request and recursively retry all other alternatives.

5.4.3 Evaluation

The sample implementation adds the transparent re-invocation mechanism to a legacy ORB implementation with no support for fault tolerance without modifying any of the legacy code. The aspect oriented implementation is much superior to the traditional "open surgery" implementation as it does not introduce extra complexity to the original architecture, in terms of both code size and class coupling. For application domains that do not require fault tolerance support,the underlying ORB is essentially not changed. We can also "weave" the feature into the ORB if it is required to do so. The configurability and adaptability is greatly enhanced.

5.5 Lesson Learned

We have learned the following lessons through our experience of applying aspect oriented analysis and design in solving middleware architecture problems.
1. AOP does not incur much runtime penalty or code bloat. That is because the logic that AOP tries to modularize either already exists inside or needs to be put into the primary decomposition. Our retrofitting work extracts the tangled code from the legacy middleware implementation and collects them in separate modules. The static weaving of AspectJ keeps the overhead of code transformation at minimal.

2. The AOP implementation has a causal relationship with the primary decomposition. That entails the modification of the primary abstraction model leads to corresponding changes in aspect modules. As we explained earlier, we expect the adoption of AOP paradigm to make the primary abstraction much more stable and less influenced by requirement changes. Moreover, through better design of the "weaving" mechanism, aspect implementation can be further decoupled from the primary program. We expect better architectural techniques, such as AOP design patterns, can be used to achieve that.

3. It is still difficult to discover all aspects in middleware platforms, which could bring challenges to more specifically defining what the primary design requirements of middleware are.

4. Runtime adaptation is a very attractive feature in middleware domain. However, there are few mature runtime weaving aspect languages available. The overhead and the optimization of runtime aspect weaving need to be researched further in the context of middleware.

6. RELATED WORK

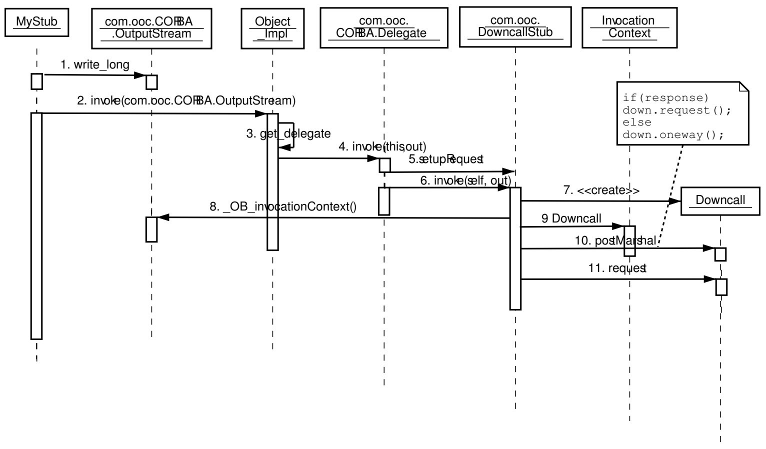

Figure 13: Client Side Invocation Sequence

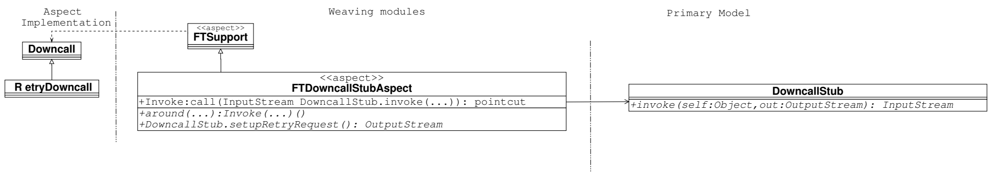
Aspect Implementation
Weaving modules
Primary Model
Downcall
R etryDowncall
<<aspect>>
FTSupport
<<aspect>>
FTDowncallStubAspect
+Invoke:call(InputStream DowncallStub.invoke(...)): pointcut
+around(...):Invoke(...)()
+DowncallStub.setupRetryRequest(): OutputStream
DowncallStub
+invoke(self:Object,out:OutputStream): InputStream

Figure 15: UML Diagram: FT aspect implementation

Related work on the topics discussed in this paper can be broadly classified into approaches that provide customization through static or dynamic policy selection, reflection to adapt middleware internals to changing runtime conditions, and configuration based on various forms of aspect definitions. Much of the discussed projects use several of these techniques. We briefly discuss some of them below.

Astley [19] achieve middleware customization through techniques based on separation of communication styles from protocols and a framework for protocol composition. Further aspects that crosscut the system implementation are not explicitly addressed. Several projects exploit reflective programming techniques to allow the middleware platform to adapt itself to changing runtime conditions. This includes projects such as openCOM [15], openCORBA [13], and dynamicTAO [4]. Recent progress in this area has been summarized in a reflective middleware workshop[6]

LegORB and Universally Interoperable Core are middleware platforms designed for hand-held devices, which allow for interoperability with standard platforms. Both offer static and dynamic configuration and aim to maintain a small memory footprint by only offering the functionality an application actually needs. Customizable functions range from the transport protocol to method dispatching and marshalling. Both platforms do not support the notion of aspects as code cross cutting concerns. Aspects in the sense of LegORB and UIC are functional units supporting application-level requirements.

[6]Reflective middleware workshop April 7th-8th, 2000, http://www.comp.lancs.ac.uk/computing/rm2000/

Similarly, Jonathan constitutes an open middleware framework that can be customized with respect to a large number of functions. Jonathan aims to embrace several standard middleware platforms and offer customization according to application needs. It can be configured to use IIOP or RMI.

The difference between our approach and those listed above is that we analyze the challenges of middleware architecture more fundamentally from the requirements domain. As the result of that analysis, we believe that the aspect oriented approach is more powerful in dealing with architectural issues like configurability, adaptability and evolution. Aspect oriented approach shows more promises to solve crosscutting problems than other approaches, such as those based on reflection and component programming.

Bernard and Putrycz [5] presents an AOP approach of adding load balancing functionality to ORBacus. Our work differs from it fundamentally as we aim at solving a much broader set of architectural problems and developing new middleware architecture methodologies.

This work is inpired by Jacobsen [10], [11] who outlined the use of non-traditional programming paradigms for middleware system design.

7. CONCLUSION

The architecture of middleware platforms has been evolving dramatically due to the necessity of a software layer that decouples applications from the concern of handling the complexity of distributed computing environment. The driving force comes from the effort of incorporating more and more new design requirements from a wide range of ap-

```
aspect POAISAspect extends InterceptorSupport
{
   //an interception point is defined as
  //the upcall creation method of POA
  pointcut UpCallCreation( POA_impl poa,
  // and other arguments of the creatUpcall method):
   args(...) &&
   call(com.ooc.OB.Upcall
  POA._OB_createUpcall (...))&&target(poa);

  //The around call replace the original upcall
  //creation semantics with a new one
  com.ooc.OB.Upcall
  around(com.ooc.OBPortableServer.POA_impl poa,
  // and other arguments of the creatUpcall method
  ):
  UpCallCreation(poa,oid,upcallReturn,profileInfo,
      transportInfo,requestId,op,in,requestSCL)
   {
    com.ooc.OB.Upcall upcall = null;
    //Create an upcall with portable interceptor
    //supoort
    com.ooc.OB.PIUpcall piUpcall =
    new com.ooc.OB.PIUpcall(
      poa._OB_ORBInstance(),
      upcallReturn, profileInfo, transportInfo,
      requestId, op, in, requestSCL,
      poa._OB_ORBInstance().getPIManager());
    upcall = piUpcall;
    return upcall;
   }
}
```

Figure 9: Interceptor Support:Upcall Creation

```
privileged aspect ORBDII
{
    //introduce a new field multirequest sender in
         //ORBInstance. This field is initialized
    //by ORB_Impl, which is executed before ORBInstance
    private MultiRequestSender
         ORBInstance.multiRequestSender_;

    after(ORB_impl orb,
         org.omg.CORBA.StringSeqHolder args,
         String orbId, String serverId,
         String serverInstance, int concModel,
         java.util.Properties properties,
         int nativeCs, int nativeWcs, int defaultWcs):
    execution(private void initialize(
              org.omg.CORBA.StringSeqHolder, String,
              String, String, int,
              java.util.Properties,
               int, int, int))
        &&target(orb)&&args(
         args,orbId,serverId,serverInstance,concModel,
         properties,nativeCs,nativeWcs,defaultWcs)
    {

            orb.orbInstance_.multiRequestSender_=
            new com.ooc.OB.MultiRequestSender();
    }
}
```

Figure 11: DII:Initialization

plication domains that desire the support of the middleware platform. We have identified that it is impossible to obtain a good architectural decomposition using the conventional software engineering approach in the case of middleware, where orthogonality among design requirements is a common phenomenon. Modularity is broken and customization is becoming more and more difficult to achieve.

To address those problems, we believe that the aspect oriented engineering approach is promising in compensating the limitations of the conventional decomposition method. The method of separating the development concerns of cross-cutting properties from that of fundamental system functionalities has given us a better way of developing, evolving and deploying middleware platforms in terms of modularity, customizability and adaptability. In the case study of CORBA, we showed that certain features in the ORB such as interceptor support, fault tolerance, dynamic programming model, logging and tracing, can be treated as aspects and, therefore, can be composed in aspect programs. By doing so, not only is the feature code that scatters around the ORB architecture becoming much more manageable, we also acquire the power of being able of configuring those features in and out depending on the requirements of the application domain. Consequently, we are able to dramatically expand the spectrum of middleware application. Maximum degree of configurability and adaptability should be the nature of middleware platforms. Aspect oriented software engineering approach can greatly help in achieving the design goal.

Though conceptually intuitive, we still need to quantitatively analyze the benefits of aspect oriented approaches in the context of middleware architecture. We have performed the aspect oriented implementation by taking certain features out of an existing legacy ORB implementation, implementing them as aspect programs, and weaving those features back into the ORB. We have presented our preliminary quantitative analysis to inspect the effects on both the structure and also the performance of middleware as the result of applying AOP techniques to the middleware architecture. Our conclusion is that AOP lowers the complexity

```
privileged aspect DelegateDII
{
   org.omg.CORBA.Request
   com.ooc.CORBA.Delegate.request(org.omg.CORBA
  .Object, String)
   {
      return new Request(self, operation);

   }
}
```

Figure 12: DII:Request Creation

```
priviledged aspect FTDowncallStubAspect
extends InterceptorSupport
{
  public OutputStream
  DowncallStub.setupRetryRequest
           (String op, boolean resp)
  {
      //actual code omitted
  }

  pointcut Invoke( DowncallStub stub,
  // and other arguments of the creatUpcall method):
  args(...) &&
  execution(public com.ooc.CORBA.InputStream
  DowncallStub.invoke (...))&&target(stub);


  com.ooc.CORBA.InputStream
  around(DowncallStub stub, org.omg.CORBA.Object
       self, com.ooc.CORBA.OutputStream out
  ):
  Invoke(stub,self,out)
  {
    try{
      return proceed(self,out);
    }catch(FailureException e)
    {
      //Retrieve the invocation context
      InvocationContext ic =(InvocationContext)
      out._OB_InvocationContext();
      String op = ic.downcall.operation();
      String resp = ic.downcall.response();
      setupRetryRequest(self,op,resp);
      return stub.invoke(self,out);
    }
  }
}
```

Figure 14: Client Side Fault Tolerance Support

of the architecture, at least preserves the runtime behaviour, and dramatically increase its configurability and adaptability. In our future work, we will accumulate more experience in applying the aspect oriented analysis and design principles on other variances of middleware technologies. That will greatly assist us in designing a fully aspect oriented middleware platform, which is our long term research goal.

8. REFERENCES

[1] AspectJ. http://www.aspectj.org.

[2] Hyperj. http://www.alphaworks.ibm.com/tech/hyperj.

[3] Gerald Brose. Raccoon - an infrastructure for managing access control in corba. DIAS2001, kluwer, 2001.

[4] Fabio Kon Manual Roman Ping Liu Jina Mao Tomonori Yamane Luiz Claudio Magalhaes Roy H. Campell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb,. *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2000.

[5] Guy Bernard Erick Putrycz. Using aspect oriented programming to build a portable load balancing service.

[6] Robert Filman. Achieving ilities. http://ic.arc.nasa.gov/ filman/text/oif/wcsa-achieving-ilities.pdf.

[7] Object Management Group. Request for information. July 1990.

[8] Object Management Group. Fault tolerant corba draft adopted specification. March 2000.

[9] Object Management Group. The common object request broker: Architecture and specification. December 2001.

[10] Hans-Arno Jacobsen. Middleware architecture design based on aspects, the open implementation metaphor and modularity. Workshop on Aspect-Oriented Programming and Separation of Concerns, August 2001. Lancaster, UK.

[11] Hans-Arno Jacobsen. Re-thinking middleware architecture design. The Sixth Biennial World Conference on Integrated Design & Process Technology, June 2002. Pasadena, California.

[12] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[13] T. Ledoux. A reflective open broker. *Lecture Notes in Computer Science*, 1999.

[14] Robert Kelly Louis DiPalma. Applying corba in a contemporary embedded military combat system. OMG's Second Workshop on Real-time And Embedded Distributed Object Computing, June 2001.

[15] Clarke M. Blair G. Coulson G. Parlavantzas N. An efficient component model for the construction of adaptive middleware. *IFIP / ACM International Conference on Distributed Systems Platforms (Middleware'2001)*, November 2001.

[16] C.A. Constantinides Atef Bader Tzilla H. Elrad Mohamed E. Fayad P. Netinant. Designing an aspect-oriented framework in an object-oriented environment.

[17] OMG. IDL to Java Language Mapping Specification.

[18] Linda H. Rosenberg. Applying and interpreting object oriented metrics.

[19] M. Astley D.C. Sturman and G. A. Agha. Customizable middleware for modular software. *ACM Communications*, May 2001.

[20] Erich Gamma Richard Helm Ralph Johnson John Vlissides. *Design Patterns*. Addison-Wesley, 1995.