

# A Concrete Solution for Web Services Adaptability Using Policies and Aspects

Fabien Baligand<sup>1</sup>  
Ecole des Mines de Nantes  
4 rue Alfred Kastler  
44307 Nantes, France  
+33 (0)6 64 64 86 26  
fbaligan@eleve.emn.fr

Valérie Monfort<sup>2</sup>  
Université Paris 1 Sorbonne  
90 rue de Tolbiac  
75013 Paris, France  
+33 (0)6 74 94 89 17  
v-monfort@mdtvision.com

<sup>1, 2</sup>  
IBM, MDTVision  
31, Avenue de la Baltique  
91954 Les Ulis, France

## ABSTRACT

Traditional middleware is usually developed on monolithic and non-evolving entities, resulting in a lack of flexibility and interoperability. Among current architectures, Service Oriented Architectures aim to easily develop more adaptable Information Systems. Most often, Web Service is the fitted technical solution which provides the required loose coupling to achieve such architectures. However there is still much to be done in order to obtain a genuinely flawless Web Service, and current market implementations still do not provide adaptable Web Service behavior depending on the service contract. Therefore, our approach considers Aspect Oriented Programming (AOP) as a new design solution for Web Services. Based on both WSDL and Policies contracts, this solution aims to allow better flexibility on both the client and server side. In this paper we expose our technical and concrete solution using Axis as the SOAP Engine, WSS4J as the WS-Security handler, and Javassist to weave some non-functional security aspects depending on the policies requirements.

## Categories and Subject Descriptors

D.2.12 [Interoperability]: Distributed objects

**General Terms:** Languages, Experimentation.

**Keywords:** Service, Web Service, Adaptability, Reusability, Aspect Oriented Programming, Service Oriented Architecture.

## 1. INTRODUCTION

Companies are faced with economic challenges which require Information Systems (IS) changes. They are buying new companies, externalizing departments. They are faced with Time to Market constraints and strong competitiveness. Moreover, companies have to communicate with distant IS as partners, suppliers, etc. Consequently, they need to exchange data through workflows in heterogeneous contexts. To illustrate this matter of fact, the Service Oriented Architecture (SOA) concept has emerged and aims to give methodological and technical answers for these concerns [14].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSOC'04, November 15-19, 2004, New York, New York, USA.

Copyright 2004 ACM 1-58113-871-7/04/0011...\$5.00.

Among the different notions gathered in this concept, the Service paradigm leads the spirits, symbolizing the loose coupling that SOA aims to provide. Recently, a new middleware technology, namely Web Service, was born to bridge heterogeneous systems. Even though Web Services are not the only way to model the Service paradigm, they are likely to be one of the major technologies used to achieve both the interoperability and loose coupling required for SOA.

In this context, Web Service technology is asked to handle the same features as components from the DCOM, J2EE or CORBA worlds already handle. These features, such as security, reliability, or transactional mechanisms, can be considered as non-functional aspects. Obviously these aspects are crucial for business purposes and one cannot build any genuine IS without consideration for them.

However, managing these aspects is likely to involve a great loss in interoperability and flexibility. This effect has already been experienced with various middleware technologies. Mostly, middleware delegates these tasks to the underlying platform, hiding these advanced mechanisms from the developer, and then establishing a solid bond between the application and the platform.

Thus, a great deal of work is required to make Web Service fully appropriate for industry. Especially, mechanisms in charge of handling non-functional tasks must preserve seamless interoperability.

Our industrial experiences lead us to model and implement Extended Enterprise. We used Web Services to cross BizTalk Server and J2EE WebSphere platforms. Our conclusion is that none of these two major platforms provide a flawless Web Service model with the ability to adapt seamlessly to non-functional concerns. Our study of Web Services norms based on industrial cases and the feedback we received allowed us to define and implement a new pragmatic solution to handle these aspects with great care to preserve interoperability and reusability.

In this paper, we will first introduce the different Web Service principles and norms, before discussing the issues encountered when developing with current Web Services solutions. Next, we will describe ideas and concepts that can provide an answer for a better interoperability and reusability in Web Service world. Then, our technical solution towards more flexible Web Services will be presented. Finally, we will give an idea about the current limitations of our work and we shall conclude.

## 2. WEB SERVICES PRINCIPLES AND NORMS

### 2.1 Web Service as a Service Implementation

Web Services, like any other middleware technologies, aim to provide mechanisms to bridge heterogeneous platforms, allowing data to flow across various programs. The Web Service technology looks very similar to what most middleware technologies look like. Consequently, each Web Service possesses an Interface Definition Language, namely Web Service Definition Language (WSDL), which is responsible for the message payload, itself described with the equally famous protocol SOAP, while data structures are explained by XML Schemas [13].

In fact, the winning card of this technology is not its mechanism but rather the standards upon which it is built. Indeed, each of these standards is not only open to everyone but, since all of them are based on XML, it is pretty easy to implement these standards for most platforms and languages. For this reason, Web Services are highly interoperable and do not rely on the underlying platform they are built on, unlike many Object Remote Procedure Call (ORPC). According to a vast majority of industrial leaders, Web Service is likely to become the best fitted technology for implementing Service Oriented Architectures. Figure 1 illustrates how this technology can suit to the service layer of Service Oriented Architectures.

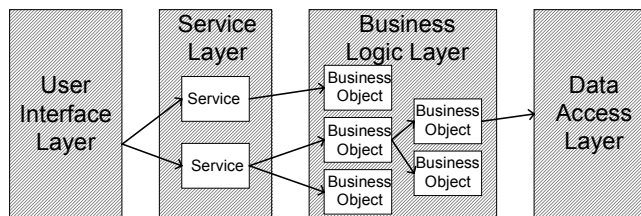


Figure 1. Example of Service Oriented Architecture model.

### 2.2 The Message Contract

Web Services provide a minimalist mechanism to interconnect different applications. But one fundamental point is the importance of the WSDL being the exact interface of the system. As we said earlier, most of ORPC take a great care of hiding the message layer details from the developer. This approach breaks down when the applications involved do not lay on the same middleware infrastructure, and when interoperability becomes a major concern, traditional ORPC fail to achieve this properly. With Web Services, the message contract (WSDL) is the central meeting point which connects applications. The WSDL contract constitutes the design view upon which developers can generate both client and server sides (proxy and stub), as can be seen in figure 2.

### 2.3 The Business Features

Now Web Services have to incorporate new features so they can face any challenge associated with usual business contexts. This reality is translated by tremendous efforts to establish standards concerning each of these non-functional aspects. The roadmap of these efforts is guided by the Web Service Architecture as shown in figure 3.

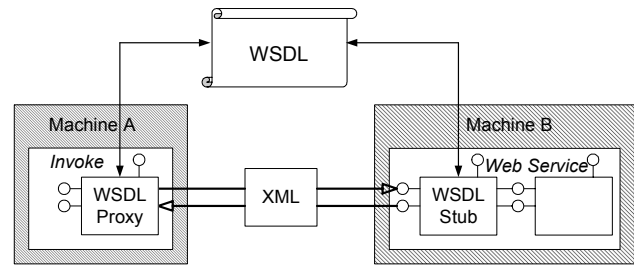


Figure 2. WSDL as a starting point for client and service generation.

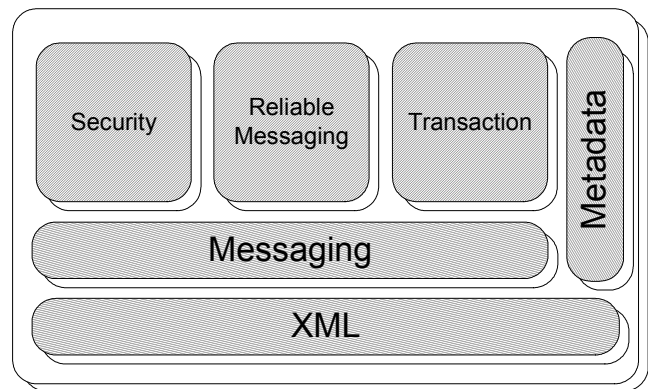


Figure 3. Web Service Architecture.

Works underway aim to establish specifications related to Messaging, Security, Reliability, Transactional and Metadata concerns. These are the main aspects which should provide Web Services with genuine business features. Let us give a brief overview of these main specification domains:

- Messaging specifications deal with message transport properties and especially provide an abstract transport mechanism between Web Services (WS-Addressing). Message attachments are also specified in this area of concern.
- Security is of course the major concern that Web Services are expected to handle. The main capabilities offered by WS-Security are message signatures, message encryption and authentication with token or certificates (X509, Kerberos, etc.). Many other specifications are underway to enable creation of trusted areas and answer other specific situations.
- Reliable Messaging provides mechanisms to take care of the successful reception of multiple messages sent from one end to the other end.
- Transactional domain is responsible for bringing to Web Services the specifications related to business coordination, like the two phase commit.

At this point, it is essential to note that every specification, except those related to Metadata, aim to describe the expected content of the header belonging to messages delivered by Web Services. For instance, if a user needs to be authenticated to use some Web Service methods, he will have to add a token in the header of the SOAP messages sent to this Web Service. Conversely, the Web

Service will have to get the token from the SOAP header and validate the authentication. Both client and service know how to insert and collect the token because the header message is fully explained by an appropriate specification: WS-Security.

Metadata specifications especially tackle WSDL and Policies. As we have already discussed above, WSDL constitutes a message contract which explains the different structures and endpoints involved in reaching any Web Service. However, if we only consider WSDL, there is still a missing piece to make a Web Service work properly. For instance, if a Web Service requires authentication, this requirement has to be fully declared, otherwise the communication will fail at runtime. In other words, we need data whose role is to provide a genuine service contract. Hopefully Policies provide a flexible and extensible framework for expressing the abilities, requirements, and preferences of entities in a Web Services-based system.

Without Policies, there would be no way for the Web Service to express its requirements concerning non-functional aspects. Policies impact on both client and service. They need to be handled in order to achieve real interoperability between the two parties.

Now, let us focus on the issues encountered when using the current toolkits.

### 3. CURRENT FLEXIBILITY ISSUES

#### 3.1 Toolkits Handling Business Features

In order to provide the missing business features required to leverage Web Service technology, a first set of tools has emerged. Built on top of both platforms .NET and J2EE, Microsoft and IBM have implemented their own toolkit with regards to the Web Service specifications.

Web Services Enhancements for Microsoft .NET (WSE) [16] is a supported add-on to the Microsoft .NET Framework providing developers the latest advanced Web Services capabilities such as security, security policy, addressing, routing, and attachments. Instead of using a regular Web Service proxy, a new class, namely Microsoft.Web.Services.WebServicesProtocol, enables the treatment of advanced mechanisms performed by an enhanced proxy. Whenever a message is to be sent or received, it has to go through the enhanced proxy which acts according to the SOAP context of the message.

The Emerging Technologies Toolkit (ETTK) [17] is a software development kit for designing, developing, and executing emerging autonomic and Web Service technologies. It provides an environment in which to run emerging technology examples that showcase recently announced specifications and prototypes from IBM's emerging technology development and research teams. Based on Axis [18], ETTK processes messages through handlers in chain. One particular chain enables developers to insert their own message managers, such as security handlers. A MessageContext object is included in outgoing messages and is extracted from incoming messages, as shown on figure 4. The handlers in charge of the transformations are specified in a Web Service Deployment Descriptor (WSDD) file.

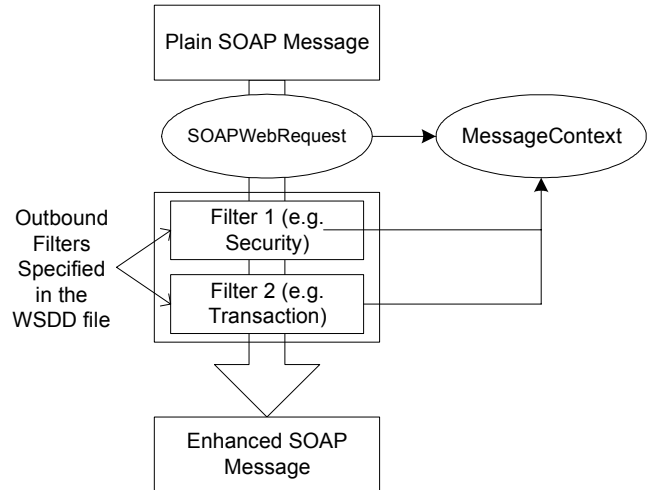


Figure 4. SOAP message filtering.

These toolkits look quite similar in the sense that they operate and compute messages following the same principles that can be seen in figure 5 below. SOAP Engines are composed of filters (SOAP handlers) whose main role is to perform transformations on the SOAP message, depending on parameters included in the header. The SOAP headers are in charge of delivering the context of the message (authentication tokens, reliable messaging properties, etc.).

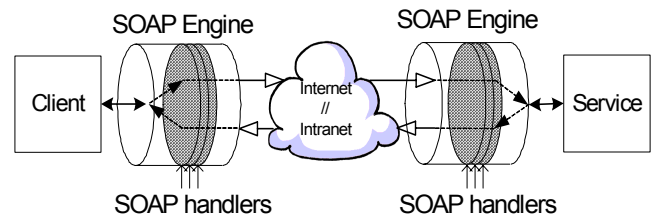


Figure 5. SOAP message processing overview.

#### 3.2 Some Concrete Implementations

In order to understand more precisely the mechanisms and the architecture, let us see some code samples from our Extended Enterprise implementation. This case was originally developed for an automotive company to implement its relationship with partners and distant workshops. In this scenario, we tried to expose two major non-functional aspects often required in business architectures: Security and Routing. The Security aspect is illustrated when a Java JSP client wants to submit to a .NET Web Service some encrypted data with an X509 certificate. To achieve such a mechanism it is necessary to configure the ETTK client. Indeed, as explained previously, ETTK uses handlers to transform the message during the specific treatment layer process. Therefore, we need to add a specific security handler as shown below.

```
<handler name="handler-SS"
type="java.com.ibm.wstk.axis.handlers.SecuritySender">
<parameter name="configPath"
value="services/demos/encclient/deployment/handler-config.xml"/>
...
</handler>
```

Next the configuration of this handler must be specified in the handler-config.xml configuration file (referenced above).

```
<DecryptionKeys>
<KeyStore type="jks" path="..\common/demo.jks"
storepass="password">
<Key alias="demokey" keypass="password"/>
</DecryptionKeys>
```

On the .NET side, the developer needs to create a method whose role is to add the certificate corresponding X509 token to the reception context.

```
SoapContext repContext = HttpSoapContext.ResponseContext;
X509SecurityToken x509token = GetSecurityToken();
if (x509token != null)
{
    repContext.Security.Tokens.Add(x509token);
    repContext.Security.Elements.Add(new Signature(x509token));
}
```

Eventually if the .NET Web Service replies using encryption, it will also have to add this token to the sending context. Communication between client and server will then be totally encrypted and the messages will be successfully processed through the different filters of both platforms thanks to interoperability specifications.

Similar mechanisms occur with the routing aspect. In our case study, SOAP messages are sent to an endpoint and the engine decides, depending on a value written in the SOAP header, which underlying Web Service will have to process the message. Once again the developer has to develop handlers for both the Java and .NET sides and to specify them when the service is deployed. The code below shows the implementation of the .NET handler inheriting from RoutingHandler.

```
public class Route : RoutingHandler
{
    protected override void ProcessRequestMessage(SoapEnvelope message,
    Path outgoingPath)
    {
        string valueX = message.Header.GetElementsByTagName("valueX")
        [0].InnerText;
        if (IsLimitReached(valueX))
        {
            Via ws1 = new Via(new Uri("http://localhost/ws1.asmx"));
            outgoingPath.Fwd.Insert(0, ws1);
        } else ...
    }
}
```

Once implemented, this file must be registered in the web.config file in charge of the Web Service configuration.

```
<system.web>
<httpHandlers>
<add type="ns1.Route, ns1" path="*.asmx" verb="*" />
</httpHandlers>...
```

### 3.3 Issues Encountered With These Designs

As we can see, both solutions do not automatically answer to the service contract wishes. Indeed, there is no mechanism that allows developers to create policy-adaptable Web Services, and this causes a major lack of flexibility. With these approaches, if policies are to change, or if a Web Service has to handle two different policies from different clients then it will fail at runtime.

The reason is: for both platforms, handling business features necessarily implies deploying certain handlers. Thus, Web Services and clients are asked to answer properly to any policy requirement as soon as they are coded and deployed. For instance, if a Java Web Service is asked to support different kinds of security tokens or certificates depending on its clients, it will not be able to deal properly with each of them because specific handlers have already been deployed along with the service. Also, if a .NET client using WSE needs to transmit a Kerberos token along with the outgoing message, it will have to add some non-functional code within its own code, with no consideration for the separation of concern approach [7].

Therefore, current Web Service design cannot help Service Oriented Architectures to accomplish full interoperability. Non-functional features, such as security, routing, reliability, and transactions, cannot be defined once for all when developing or deploying an application. Otherwise Web Services will become as monolithic as previous middleware technologies were. In other words, features that Web Services are asked to provide will become strongly coupled with the application. Designers and developers need Web Services that can automatically adapt to policies.

Knowing these issues, we can now examine how policies and aspects can help to fix them by providing both the data and the mechanisms to achieve adaptable Web Services.

## 4. TOWARDS MORE FLEXIBLE WEB SERVICES

### 4.1 Aspect Oriented Programming

Our technical approach to current Web Service solutions enabled us to notice two major facts which are at the root of Web Service's lack of flexibility. First, there is no dynamic mechanism to bind policies and Web Service handlers. Secondly, there is no clean separation of concerns between the functional and the non-functional code, and also between SOAP logic within handlers and non-functional logic within handlers, as figure 6 shows.

Once the client or service is coded and the handlers are deployed, the Web Service cannot handle new features and, because the different logics are tangled up, it is not easy for another developer to reuse the application in a different context. Consequently, an appropriate way to deal with these crosscutting concerns [9] would be to use different units of modularization to encapsulate these logics [5]. Moreover, if these units of modularization could be managed by a dynamic mechanism, then the whole system would be able to dynamically reconfigure itself depending on the policies [2].

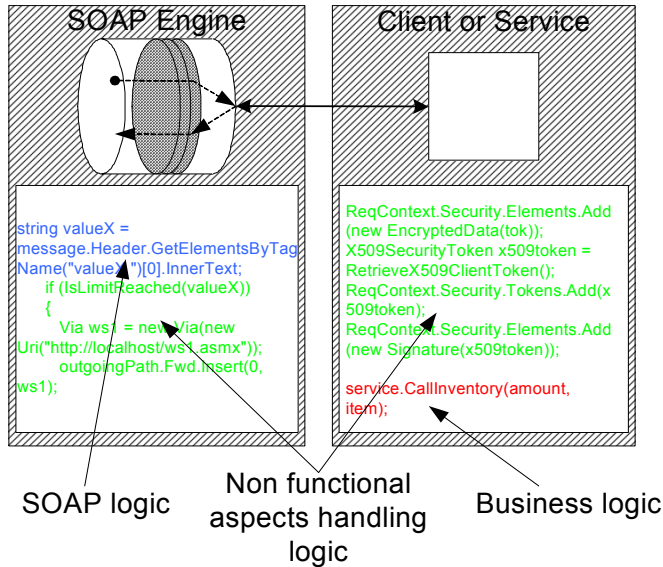


Figure 6. Tangled logics within SOAP Services.

These requirements lead us to consider Aspects Oriented Programming (AOP) as an answer to Web Services reusability issues [3, 4]. AOP is one of the most promising solutions to the problem of creating clean, well-encapsulated objects without extraneous functionality. It allows the separation of crosscutting concerns into single units called aspects, which are modular units of crosscutting implementation. With AOP, each aspect is expressed in a separate and natural form, and can be dynamically combined together by a weaver. As a result, AOP widely contributes to increased reusability of the code and provides mechanisms to dynamically weave aspects.

Considering Web Services, non-functional aspects handling logic should be encapsulated within multiple aspects. Each aspect would be in charge of certain features, such as security, and would deal directly with well-defined objects like Kerberos tokens (security) or Shipping forms (reliable messaging) as shown figure 7.

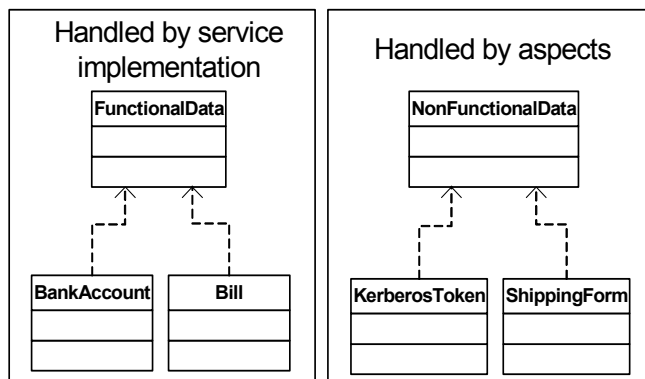


Figure 7. Functional and non-functional data.

Pushing the non functional handling logic inside aspects means that handler's role has to be redefined, as they will only contain SOAP logic then. The idea is to replace the multiple specific handlers, which used to process SOAP messages depending on their own implementations, by a global handler whose role will be restrained

to extracting non-functional data contained in incoming messages, and pushing it inside outgoing messages.

## 4.2 Weaving Process

At this point, we need to define where, when and how the aspects should be weaved. Let us answer these questions by considering the different opportunities for each of them. First, aspects could be weaved to the global handler, to the stub or to the service implementation itself. In fact, considering the global message path and process, choosing any of these entities does not really influence the mechanism. However, we found it more convenient to weave aspects to the stub since it provides a natural meta object to focus on the service itself [6]. Secondly, there are multiple choices for when to weave aspects. It could occur during compile time, deployment time, load time or run time. If the weaving were to happen at compile time or deployment time, it would not be possible to handle policy changes dynamically. Conversely, there is no need to weave aspects at runtime since the policy document will not be most likely to change after the service starts running. Thus, the ideal solution is to weave aspects when the service is loaded to enable one single yet sufficient analysis of the policies document for each new instance [12]. Thirdly, the weaver should be an application capable of reading the policy document, interpreting the policies, selecting the relevant aspects and finally mixing them with the plain stub, as can be seen on figure 8.

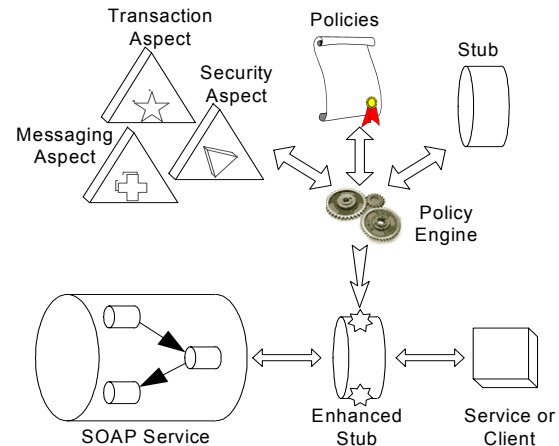


Figure 8. Aspects weaving at load time.

Transmitting non-functional data to aspects weaved to the stub at load time is one fitted solution to achieve genuinely flexible Web Services. This mechanism allows Web Services to be reused more easily since each non-functional aspect is detached from both the service implementation and the handler. The Policy Engine inserts these aspects depending on the service contract requirements [8], which means that interoperability is preserved if, for instance, requirements from different clients vary.

We have seen how AOP can help to gain flexibility through a cleaner separation of logics and which mechanism can help to provide policy awareness among Web Services. We shall now present our concrete implementation of these concepts.

## 5. OUR CONCRETE SOLUTION

### 5.1 Structure of Axis

In our solution we take advantage of multiple open source solutions already available for Java so we modify and assemble them easily. This way, we can start with a ready-to-use platform that we need to complete in order to obtain flexible Web Services.

Thus, the Web Server and the SOAP Engine are constituted by the famous open source duo Tomcat-Axis. Basically, Axis plugs into the Tomcat servlet engine, meaning that it can be considered the same as any other Web Application. Web Services are hosted and managed by Axis in a transparent way for Tomcat as shown in figure 9.

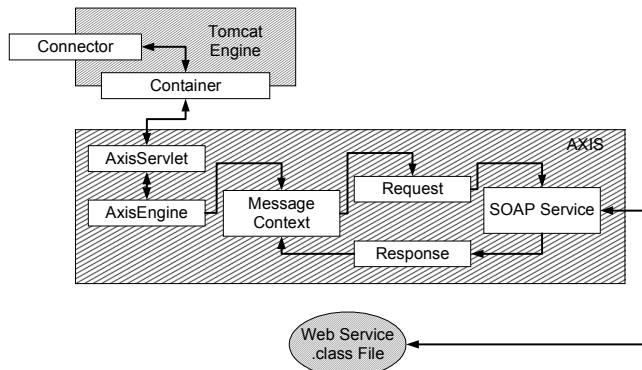


Figure 9. Axis server side architecture.

Axis is based on the concept of a chained message. The MessageContext object is a wrapper object for the request and the responses message and for contextual information about process, request, response, etc. In figure 9, Request and Response are handlers that manipulate the MessageContext. Because these handlers can easily manipulate this object, it is quite natural to select these handlers to act like basic SOAP logic handler. For instance, if an incoming SOAP header contains data that says the body message is encrypted, then the Request handler needs to decrypt the body, as an automatic reflex. But the genuine non-functional logic is hosted by the aspects, and non-functional data used by these aspects is transmitted by the provider. The provider is another handler that, when invoked, calls the stub corresponding to the service invoked. Once processed and transformed into appropriate objects, these data will be passed to the stub weaved with aspects.

### 5.2 Stub Bytecode Modifications

Let us now see how aspects are weaved to the stub. First, we need to understand how class loading works in Tomcat. Indeed, if we can modify the bytecode of the stub object when it is loaded into the Java Virtual Machine (JVM), then it will be possible to weave the aspects at load time. Tomcat uses multiple class loaders, which are java objects aiming to load resources (class or jar files). With Java 2, class loaders follow a delegation model, which means that if a class is asked to be loaded by a class loader, then this class loader will first ask it's parent class loader to do so. If it cannot load the class, the initial class loader will search inside its own resources. All Tomcat class loaders follow this rule except Web Application class loaders, which are responsible for the loading of each class of the Web Application they are in charge of. Consequently, the idea is to modify the class loader in charge of Axis Web Application so we

can reach any Web Service stub anytime it is loaded, as shown figure 10.

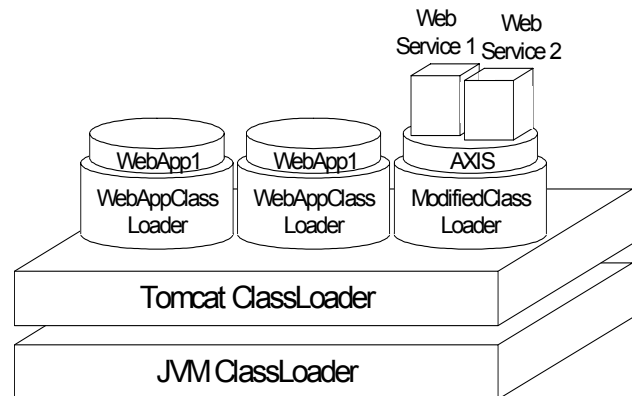


Figure 10. Class loaders hierarchy.

To obtain such a class loader, we just need to reuse the code of the Axis regular WebAppClassLoader and specify that Tomcat has to use the ModifiedClassLoader when it loads Axis Web application, via the server.xml configuration file.

```
<Context docBase="C:\axis-1_1\webapps\axis" path="/axis">
  <Loader loaderClass =
    "org.apache.catalina.loader.ModifiedClassLoader"/>
</Context>
```

The next step is to use a tool which allows both introspection and reflection - the former to inspect the stub code when it is loaded and the latter to achieve the weaving of aspects. One particularly convenient answer to these requests is brought by Javassist [1]. Javassist is a class library for enabling structural reflection in Java, which is performed by bytecode transformation at compile time or load time. In order to modify bytecode at load time, Javassist performs structural reflection by translating alterations of structural reflection into equivalent bytecode transformation of the initial class file. After the transformation, the modified class file is loaded into the JVM by a special class loader.

To bring this mechanism into our solution, the ModifiedClassLoader must adhere to three rules. First, it must encapsulate a Javassist.ClassPool object, which will act as a container for objects containing class files to be loaded [15]. These objects derive from the CtClass class which is a convenient handle for dealing with class files (methods or fields adds or renames, etc.). Next, when the ModifiedClassLoader constructor is called, this ClassPool object must be instantiated with the Web Application class path so it can get the scope of the classes it can handle. Finally, whenever a class is to be loaded, the findClassInternal (String name) method is called and must contain the transformation logic which will affect the stub object anytime it is loaded. The code below shows these modifications inside of what used to be the regular WebAppClassLoader class.

```
public class ModifiedClassLoader extends URLClassLoader {
    protected ClassPool pool = null;
    public WebappClassLoader() {
        pool = ClassPool.getDefault();
    }
}
```

```

pool.insertClassPath(new LoaderClassPath(this));
...}
/* Method called whenever a class is to be loaded */
protected Class findClassInternal(String name) {
    ResourceEntry entry = findResourceInternal(name, classPath);
    Class clazz = entry.loadedClass;
    /* Javassist loader is invoked to get an easily modifiable CtClass */
    CtClass cc = pool.get(name);
    /* Class modifications according to the PolicyEngine */
    if(isStubClass("name"))
        PolicyEngine.Process(cc);
    byte[] b = cc.toBytecode();
    clazz = defineClass(name, b, 0, b.length);
    ...
    return clazz;
}...

```

### 5.3 Policy Engine as a Weaver

Eventually, we shall define how the Policy Engine works. As explained in section 2.3, Policies constitute the Service Contract and thus describe what the requirements to establish communication are. For instance, the <wsse:SecurityToken> element, as shown below, is used to describe which security tokens are required and accepted by a Web service. It can also be used to express which security tokens are included when the service replies.

```

<SecurityToken wsp:Preference="..." wsp:Usage="..." >
    <TokenType>...</TokenType>
    <TokenIssuer>...</TokenIssuer>
    <Claims>...Token type-specific claims...</Claims>
    ... (TokenType-specific details)
</SecurityToken>

```

Once the PolicyEngine.Process(...) method is called, the engine gets a CtClass object containing the code of the stub. Because the name of this class is related to the name of the service itself, it becomes easy for the Policy Engine to locate the Policy contract and thus it can access the policy's requests. The next step for the engine is to fulfill each of these requests by inserting the appropriate aspects within the methods of the stub. This mechanism is almost equivalent for both client and service side. Eventually, the Policy Engine adds fields to the stub so it can obtain and set the non-functional data that the provider manages.

At this point, the new "SOAP messages process" is effective and can be used to dynamically handle each of the functional aspects declared in the Policy document. Figure 11 below illustrates the global mechanism at runtime.

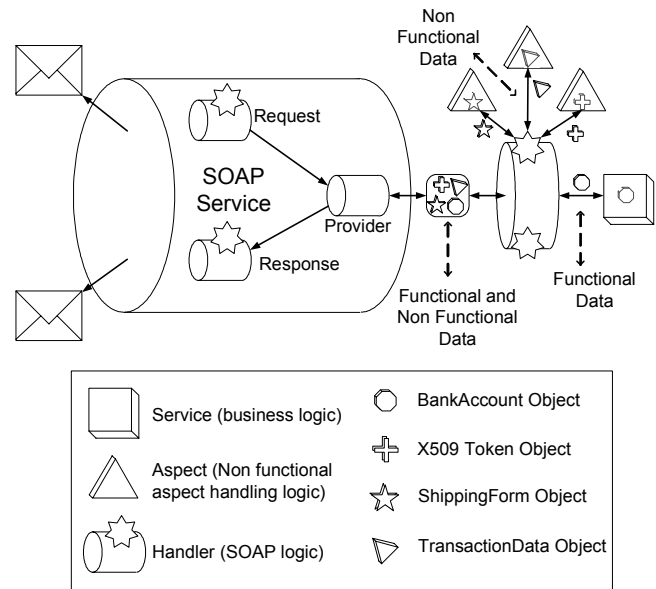


Figure 11. Functional, non-functional and SOAP logics.

### 5.4 Security Scenario

Let us conclude this section by illustrating the whole mechanism with a scenario using security. In our solution, we use Web Service Security For Java (WSS4J) as an implementation of WS-Security, more especially to handle encryption and security token insertion. The WSS4J Axis handlers already support a large number of WS-Security features and their combinations. However, it is not our aim to use WSS4J directly to handle security. Our approach consists of separating its SOAP logic (like the automatic decryption) and its non-functional aspect handling (token management). Then we insert the SOAP logic into the Request/Response handlers while the non-functional aspect handling is left to aspects.

In our case, the policy document specifies that the Web Service requires a Username token and handles 3-DES encryption. At load time, the Policy Engine adds a Username token field to the stub of this service, and weaves the targeted method with the Username token aspect. This code specifies that if the Username is unknown, or if the password is not correct, the service implementation will be skipped and an appropriate message will be returned. At run time, an incoming encrypted message containing a Username token is presented to the SOAP engine of a Web Service. The Request handler will automatically decrypt the body and will transmit the updated MessageContext object to the provider. The provider is in charge of extracting the token data and transmitting an appropriate token object to the stub along with the business objects. When the method of the stub is invoked, the aspect in charge of the token is called and it handles the token with the appropriate logic, as described above. Eventually the implementation of the service is invoked and the result will be returned along with the token of the Web Service. The provider will then fill the MessageContext response object and the Response handler will eventually encrypt the body.

As can be noticed in this scenario, this mechanism enables policies to select an aspect in charge of the security requirement (the Username token). Also, the different logics are cleanly separated



from the others, enabling both Web Service and aspects to be reused easily. If these policies were to change, different aspects would be weaved at load time and the service would become fully compliant with these new requirements.

Let us now conclude this presentation by explaining the current limitations of our solution.

## 6. LIMITATIONS AND FUTURE WORKS

There is still much work that needs to be done before this solution can be fully used in a genuine business scope. Our work allowed us to identify four major tasks that are required to make it happen.

The two first tasks fall on the Web Service community, and especially the WS-I organization, which works on WS-\* norms. There is still a need for these norms to be approved by everyone and we have to see their use in concrete scenarios to fully understand how to deal with them. The second task is also to provide complete policies describing properly each of these norm requirements. Indeed, without a proper explanation of the requirements, it would not be possible to create a dynamic mechanism for handling the multiple non-functional aspects.

Also, we need to define how to handle each norm with both an appropriate SOAP logic and non-functional aspect handling logic. For instance, if an incoming encrypted message containing a token is presented to the SOAP Service, an appropriate decryption logic must be placed within the Request handler while the token must be handled by an aspect. In our case, we have seen that there is an open implementation of WS-Security, namely WSS4J, which brought to us the code we required. However, there is still no open implementation to handle most of the other norms.

Finally, the last task is related to the Policy Engine development. The role of this Engine is to select the appropriate aspects depending on the policies. However, policies are likely to be complex to understand and many requirements may overlap with each other. Building a Policy Engine which can understand and properly respond to each of the policies will be a major task.

These multiple tasks arise from the different bounds that need to tie between each element of our solution, as illustrated on figure 12 below. Eventually, the ultimate result will consist of linking the policies requirements to the appropriate aspects handling them.

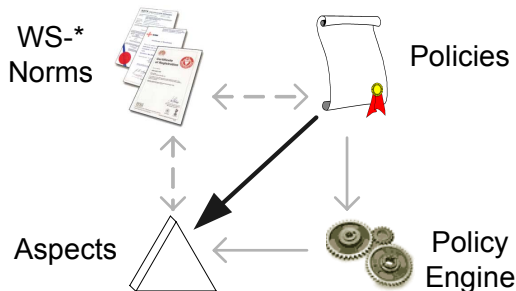


Figure 12. Bounds between each element of the solution.

## 7. RELATED WORKS

The Web Service Management Layer (WSML) [19] is an aspect based platform for Web Services allowing a more loosely coupling between the client and server sides. The idea of this technology is to

transfer the Web Service related code from the client code to this new management layer. The advantages are the dynamic adaptation of the client to find the most fitted Web Service, and it also deals with the non functional properties like Traffic Optimization, Billing Management, Accounting, Security, Transaction. This work looks very similar to the solution we provide in the sense that it aims to gather the scattered code in aspects. However, our solution especially aims to answer to the norms from the Web Service Architecture, which are described in the policies. The Web Services Mediator (WSM) [20] is a middleware layer that sits above standard Web Services technologies such as Simple Object Access Protocol (SOAP) Servers. It aims to decouple an application from its consumed Web Services, and to isolate the application's characteristics (e.g., reliability, scalability, latency).

The Aspect-Oriented Component Engineering (AOCE) [21] has been developed to capture the cross-cutting concerns, such as transaction, co-ordination and security, etc. To achieve this solution, the WSDL grammar was extended by enriching it with aspect-oriented features so that it becomes better characterized and categorized. However, there are no universally accepted standards in the terminology and notations used in AOCE by the various interested parties trying to use it. On the whole, AOCE and our work seem to offer very similar approaches but, although using just policies to select aspects might be restrictive, our strategy does not require developers to understand any vendor specific standard. The Web Service Description Framework (WSDF) [22] consists in a suite of tools for the semantic annotation and invocation of Web Services, by mixing both Web Service and Semantic Web communities. Instead of establishing a hard wired connection between the client and the service, by specifying the Web Services through addresses, WSDF enables the developer to formally specify a service using rules and ontological terms.

## 8. CONCLUSION

Service Oriented Architectures require loose coupling to access the services which will most likely be implemented with emerging Web Service technology. Using current SOAP toolkits, we noticed that interoperability between client and Web Service is damaged by non-functional aspects required by businesses (such as security, transaction, reliable messaging, etc). In fact, they require establishing a strong coupling between the service logic, the non-functional handling logic, and the SOAP logic. On top of this, there is no dynamic adaptation mechanism to bind the service contract requirements to the Web Service and client abilities. These facts significantly reduce Web Service flexibility and affect the loose coupling ability offered by Services.

The solution we provide aims to offer a dynamic mechanism to compute the service contract on the fly, enabling Web Services to become fully aware of the business requirements. The main principle consists of using computational reflection as a means to achieve separation of concerns and dynamic adaptability. Our new SOAP Service design provides a cleaner separation between the multiple logics weaved at load time. After analyzing the policies requirements, a Policy Engine is in charge of selecting the appropriate aspects to handle business mechanism like security, transactions, etc. This mechanism allows Services to gain in loose coupling.

Future works will consist of widening the application scope of this solution and validating the Web Services behavior in concrete



Service Oriented Architectures. The main tasks will be to implement a library to handle the multiple WS-\* norms and then develop a policies fully compliant Policy Engine.

## 9. REFERENCES

- [1] Chiba, S., "Load-time Structural Reflection in Java" in Proc. of ECOOP'2000, 2000, SpringerVerlag LNCS 1850
- [2] F. Baligand, V. Monfort "A Pragmatic Use of Contracts and Aspects to gain in Adaptability and Reusability" The 2004 2<sup>nd</sup> European Workshop on Web Services and Object Orientation, EOOWS'04, ECOOP, June 14-18, 2004, Oslo, Norway
- [3] M. N. Bouraqadi-Saâdani, R. Douence, T. Ledoux, O. Motelet, M. Südholt "Status of work on AOP at the OCM group, April 2001", École des Mines de Nantes, technical report, no. 01/4/INFO, 2001 KW: AOP, execution monitoring, program transformation, interpreter
- [4] Kiczales G. et al. "Aspect-Oriented Programmign", in Proc of ECOOP'97. LNCS 1241, Spinger-Verlag, 1997
- [5] Eric Tanter, Jacque Noyé, Denis Caromel, Pierre Cointe "Partial Behavioral Reflection : Spatial and Temporal Selection of Reification", 18<sup>th</sup> Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003
- [6] Chiba, S., "A Metaobject Protocol for C++" in Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, no.10 in SIGPLAN Notices vol 30, pp. 285-299, ACM, 1995
- [7] F. Baligand, V. Monfort, S. Goudeau "Standards and Web Services: Some Concrete Limitations" The 2004 International Symposium on Web Services and Applications, ISWS'04, IEEE, June 21-24, 2004, Las Vegas, Nevada, USA
- [8] D. Mandrioli, B. Meyer « Applying Design by contract » Interactive Software Engineering Inc editions Prentice Hall
- [9] O. Barais, L. Duchien, R. Pawlak, "Separation of Concerns in Software Modeling: A Framework for Software Architecture" Transformation, IASTED International Conference on Software Engineering Applications (SEA), IASTED, USA, november 2003.
- [10] Eric Tanter, Michael Vernaillen, José Piquer "Towards Transparent Adaptation of Migration Policies" Workshop in Mobile Object Systems, EWMOS 2002, 2002
- [11] Chiba, S. and M. Tatsubori, "Yet Another java.lang.Class" in Proc. of ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, July 1998
- [12] D. Sosnoki "Java programming dynamics: Transforming classes on-the-fly" Feb 2004 <http://www-106.ibm.com/developer-works/java/library/j-dyn0203.html>
- [13] visit web site <http://www.w3.org/TR/SOAP>
- [14] visit web site <http://www.service-architecture.com/>
- [15] visit web site <http://www-106.ibm.com/developerworks/library/ws-polfram/>
- [16] visit web site <http://msdn.microsoft.com/webservices/building/wse/>
- [17] visit web site <http://www.alphaworks.ibm.com/tech/ettk>
- [18] visit web site <http://www.axis.com/>
- [19] Verheecke B., Cibrán M.A., "Aspect-Oriented Programming for Dynamic Web Service Monitoring and Selection," to be published in the proceedings of the European Conference on Web Services 2004 (ECOWS'04), Erfurt, Germany, September 2004.
- [20] visit web site <http://javaboutique.internet.com/articles/WSApplications/>
- [21] Singh, S., Grundy, J.C., Hosking, J.G. Developing .NET Web Service-based Applications with Aspect-Oriented Component Engineering , In Proceedings of the Fifth Australasian Workshop on Software and Systems Architectures, Melbourne, Australia, 13-14 April 2004.
- [22] A. Eberhart. Towards universal Web Service clients. In B. Hopgood, B. Matthews, and M. Wilson, editors, Proceedings of the Euroweb 2002.