RESOLVING INCOMPATIBILITY DURING THE EVOLUTION OF WEB SERVICES WITH MESSAGE CONVERSION

Vadym Borovskiy, Alexander Zeier
Hasso-Plattner-Institut, Potsdam, Germany

Jan Karstens, Heinz Ulrich Roggenkemper
Business Process Renovation, SAP AG, Walldorf, Germany

Keywords: Incompatibility resolution, message conversion, change management, web service evolution.

Abstract: One of the challenges that Web service providers face is service evolution management. In general, the challenge is to ensure the substitutability of service versions, i.e. correct functioning of all ongoing client applications relying on the old version of a service after the version has been substituted with a new one. Unfortunately, no currently available design approach can guarantee a perfectly extensible architecture that preserves full backward compatibility during its evolution. Hence, incompatibilities are very likely to occur if an old version of a service is replaced with a new one. This paper addresses the incompatibility problem and describes a solution to the problem. This solution is based upon the already known design pattern of message translation and the ASP.NET 2.0 Web service platform. Using the platform's API the standard ASP.NET pipeline has been augmented with an additional step of applying XSL transformations to the XML payload of the messages. The solution is then verified against the Electronic Commerce Service from Amazon.com web services suite. Thus, the contribution of the work is a new .NET implementation of the translator pattern.

1 INTRODUCTION

Nowadays software systems are expected to share their data and functionality. Achieving this level of interoperability has required creating custom components to bridge incompatible systems. The main reason of the failure to create truly interoperable systems has been having lots of competitive proprietary protocols based on individual goals instead of a single widely accepted standard (Hall Gailey, 2004). The emergence of Web services (WS) promises to introduce such a standard. Based on platform neutral protocols, WS achieve interoperability at lower cost.

When considering the time factor, WS, however, don't differ from other technologies and show vulnerability with regards to change management (CM). The lack of CM techniques results in the inconsistent evolution of a Web service, which in turn leads to incompatibility between different versions of the same service. In the short run resolving only the latter problem is sufficient, but in the long run a service provider must find a solution to the former problem and improve the substitutability of versions of a service.

In this paper the incompatibility problem is addressed and a possible solution to the problem is presented. Section 2 delves into the nature of the incompatibility problem with regards to WS and builds the foundation for solving the problem. Section 3 advocates message conversion as a solution to the problem and contributes to already known work with a new way of implementing the translator pattern using the .NET environment. The section also presents a prototype that demonstrates the solution in work with a real world Web service. Section 4 discusses the related work. Section 5 concludes the article and outlines the future direction of the research.

2 INCOMPATIBILITY PROBLEM

2.1 Problem Description

To consume a Web service a client application performs a number of necessary steps. Firstly, the client must locate the service. Secondly, the client must ob-

Borovskiy V., Zeier A., Karstens J. and Ulrich Roggenkemper H. (2008).
RESOLVING INCOMPATIBILITY DURING THE EVOLUTION OF WEB SERVICES WITH MESSAGE CONVERSION.
In *Proceedings of the Third International Conference on Software and Data Technologies - SE/GSDCA/MUSE*, pages 152-158
DOI: 10.5220/0001880101520158

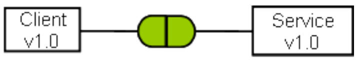

Figure 1: A client consumes a service.

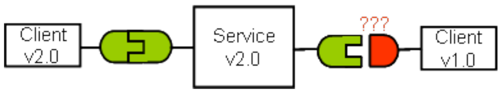

Figure 2: Incompatible interface.

tain a WS description, that is a communication contract that specifies the available functionality of the service and how to invoke the functionality. Thirdly, the client must build a proxy from the WS description and integrate the one into their system. The proxy creates an illusion of seamless integration between the client's system and the remote service. Thus the client's application consumes the interface the service provides (see Figure 1).

Over time a Web service may change, which will cause its description to change as well. In this situation new clients developed from scratch will not experience any incompatibility. However, older clients designed for the previous version of the service might face an incompatible interface and fail to interact with the new version of the service (see Figure 2).

2.2 Roots of Incompatibility

Messaging is the core of communication mechanism of WS. It is based on open standards (XML, SOAP and HTTP) and decouples Web services from their clients (Dahan, 2006). To access a service clients need only the address of the service and the XML schemas of request and response messages. This information forms the description of a service and is included in a *.wsdl* file that is published on the Web. The description is used to generate a proxy via which the clients will interact with the service.

The fact that a *.wsdl* file includes the description of messages to be exchanged with the service, means that a client must comply with communication rules of a service. Such a requirement in turn creates a dependency between the implementation of the client and the description of the service. Because of this dependency the client becomes sensitive to the changes to the WS description. If a change to the description occurs, the client may not be able to interact with the service any longer. The reason for this possible failure is that the client will produce messages according to the older version of the communication contract. The signature of operation calls deserialized from these messages will not match the signature of service's operations. Thus, the roots of the incompatibility problem are on the message level of WS.

2.3 Classification of Changes

Changes of a WS description result in changes of the messages that a service expects and sends. The understanding of the changes and the way they affect communication will definitely help to find a solution to the incompatibility problem. The first step in the analysis of possible changes is to outline all relevant parts of a WS description. The second step is to understand how each of the parts can change.

2.3.1 Content of Web Service Description

WSDL is an XML-based language that is used to define a Web service and the mechanisms to access it. WSDL defines a service in two parts: abstract and concrete. The abstract part describes the interface to the service without any details about how to access the service and includes three elements: *portType*, *message* and *types*. The concrete part builds upon the abstract part to add protocol-specific details and describe how to access the Web service. The concrete part includes two elements *service* and *binding*.

```
<definitions>
 <types>
  <xs:schema>
   <xs:element name="Request">
    .......
   </xs:element>
   <xs:element name="Response">
    .......
   </xs:element>
   </xs:schema>
 </types>
 <message name="RequestMsg">
  <part name="body" element="Request"/>
 </message>
 <message name="ResponseMsg">
  <part name="body" element="Response"/>
 </message>
 <portType name="EcsPortType">
  <operation name="ItemSearch">
   <input message="tns:RequestMsg"/>
   <output message="tns:ResponseMsg"/>
  </operation>
 </portType>
 <binding name="EcsBinding"
    type="tns:EcsPortType">
  <soap:binding style="document"
     transport="http://..../soap/http"/>
  <operation name="ItemSearch">
   .......
  </operation>
 </binding>
 <service name="Ecs">
  <port name="EcsPort"
     binding="tns:EcsBinding">
   <soap:address location="http://..../>
  </port>
 </service>
</definitions>
```

The listing above shows all first-level elements[1] that are related to the *ItemSearch* operation of the Amazon Electronic Commerce Service 4.0[2]. From the listing one can see that to invoke the *ItemSearch* operation *RequestMsg* message must be sent to the service (see *portType* element). The message protocol must be SOAP and the transport must be HTTP (see *binding* element). The message must contain at least one part that consists of an element of *Request* type. *Request* is a self-defined type and described inside *types* element. If *ItemSearch* operation completes successfully the result will be sent back to the client inside *ResponseMsg* message. The payload of the message is defined by *Response* element.

2.3.2 Changes of Web Service Description

Both parts of a *.wsdl* file can change over time. This subsection presents the analysis of possible changes of WS description and their effect on communication between a service and its client.

The change of the *service* element implies the change of endpoints to which messages are sent. The change affects neither the contents nor the format of messages. Storing endpoints in a configuration file will help to safeguard clients from this kind of change.

A change of the *binding* element implies changing either message protocol (SOAP, HTTP-POST, HTTP-GET, etc.) or parameter encoding if SOAP over HTTP is chosen. Such a change alters the structure of the messages, but their content stays the same.

The *portType* element defines the signature of each operation of the Web service. This part of the description is most likely to change over time. The change might happen to either the name of an operation or the *message* attribute of the *input*/*output* elements of an operation. Changing the attribute means that another message will be used to communicate with the service. In addition existing operations may be removed and new ones added.

A change of the *message* element means either changing the type or element that is referenced by any of the message parts or adding/removing a message part. In practice most messages consist of one part, therefore the second type of change is rare.

The last element to consider is *types*. The element includes the definitions of data types used in messages. This element represents the payload of messages. The *types* element is very likely to change: new type definitions can be added and existing ones can be removed or altered. A change of the *types* element most probably implies the semantic change of message payload.

Generally speaking, a change of the abstract part implies a change of service's functionality, whereas a change of the concrete part alters the way this functionality is invoked. If changing the interface or implementation of a service breaks ongoing client applications the new version of the service should be created (Lublinsky, 2007). The most notable versioning approaches for WS are described in (Lublinsky, 2007). According to Lublinsky versioning may take place at three levels: message level, method level and service level. On the message level versioning is applied by placing message elements in a unique namespace that is related to the specific version of a service. This is done by appending a version identifier or a date stamp to the namespace attribute of message elements. On the method and service levels versioning is applied by assigning different endpoints to the versions of a method and a service respectively.

3 MESSAGE CONVERSION

3.1 The Approach

Web services use XML to represent messages flowing between a service and its clients. A client constructs a message encapsulating a function call and sends the message to the service. The service processes the message and constructs an output message if required and sends the message to back to the client. An important detail is that after request and response messages have been sent, they become independent of the client and the service. As shown in the subsection 2.2 the reason of incompatibility is the discrepancy between the service's and the client's communication protocols resulted in incompatible messages. To reconcile this incompatibility message conversion can be used. The main idea of the solution is to transform XML messages using XSL transformation before they reach their destination. Figure 3 shows how the solution fits into the existing interaction process. Note that there is no necessity to use converters on the service and the client sides for each communication direction at the same time.

The solution is based on the *translator* pattern that in turn derives from the *pipes-and-filters* pattern (Hohpe and Woolf, 2004). In their work Hohpe and Woolf introduced the translator as a part of messag-

[1]For the ease of reading and comprehending all namespace attributes and URLs have been omitted.

[2]*ItemSearch* operation searches the Amazon.com product catalog for items that meet a user-specified criterion (e.g. have certain title, manufacturer, etc.).

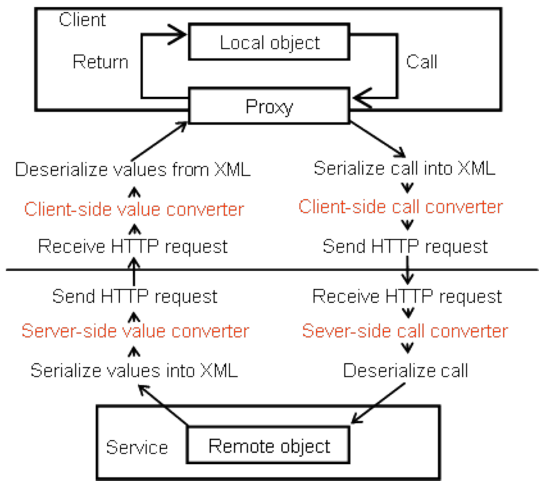

Figure 3: Message conversion.

ing system, that acts as an external entity intervening in communication process between different components. Hence, the translator stands out of WS infrastructure. With the advent of new technologies the message translator can be implemented more efficiently by means of closer integration into the communication process between a service and its client. In its essence the current approach takes an advantage of new technology, ASP.NET 2.0, and suggests a new way of including message conversion into WS infrastructure.

To successfully use the approach one requirement must be met. The semantic relationships between the payload of messages addressed to different service versions must be discoverable. Therefore, the approach assumes that the business logic of the older version is not removed but substituted, meaning that the old functionality stays essentially the same, but is offered in a new way. An important consequence of the assumption is that by using message conversion any incompatibility resulting from a change of the concrete part of a WS description (i.e. the *binding* and the *service* elements) can be resolved if the abstract part of the description stays the same.

To use the approach in practice two tasks must be done. The first one is to create a converter and plug in the one into the existing application. The second task is to develop message transformation scripts in XSLT. A solution to the first challenge largely depends on the concrete situation and first of all the implementation technology used to develop the client and the service. The main factors here are the accessibility and extensibility of message processing of the WS implementation for a given technology. The second challenge is about understanding and discovering semantic relationships between the messages of different formats and creating XSLT scripts out of the relationships.

3.2 Implementing Converter with .NET

To demonstrate the approach described in this article Microsoft .NET platform was chosen. The main reason for the choice was the very flexible customization API that .NET provides. The API allows for building custom extensions on top of the existing ASP.NET development stack (Esposito, 2006). The extensions are plugged in to the ASP.NET pipeline, which is responsible for handling Web service requests and responses. At a certain stage of the processing[3] the custom extension is invoked to perform a user-defined operation on the message (in this case conversion).

Using a .NET development kit an application was built. Powered by Amazon.com ECS 3.0[4], the application implements keyword search against the book catalog of Amazon.com. To simulate the evolution of the service all requests composed by the application were redirected to the newer version of the service, by changing service endpoint. ECS 4.0 is not backwards compatible with ECS 3.0, meaning that the fourth version cannot process requests addressed to the third version and the clients of ECS 3.0 cannot handle responses coming from ECS 4.0. Hence, simply changing the endpoint URL would not work.

On the other hand the functionality of ECS 3.0 is present in ECS 4.0, but provided in a different way. Therefore, to reconcile the incompatibility between the old client and the new version of the service message conversion was used. There were two steps in implementing the conversion. The first one was to develop the converter itself and the second was to plug in the converter to .NET infrastructure.

Creating a converter is a simple task when using .NET. The platform provides a powerful API for working with XML and XSLT. Using the API the converter loads source XML and XSL transformation to memory and then produces result XML by applying the XSL transformation to the source XML.

To plug in the converter to ASP.NET pipeline SOAP extension technology was used. A SOAP extension embodying the converter was built as a separate DLL and is completely independent of the client application. To deploy and invoke the SOAP extension to perform the conversion the DLL must be copied to the client's installation directory and regis-

[3].NET can invoke an extension at the four following processing stages: *before deserialization*, *after deserialization*, *before serialization* and *after serialization*.

[4]Electronic Commerce Service 3.0 was formerly known as the Amazon Web Service. For details on E-Commerce Service see http://aws.amazon.com.

tered with the help of a configuration file. No change of the client application code is needed!

The solution required two converters: one for request and the other for response conversion. Both converters were essentially the same. The only difference between them is the stage of message processing at which they are invoked. The request converter is invoked at *after serialize* stage and the response converter at *before deserialize* stage. Since the service-side processing chain is not accessible, the two converters were plugged in to the client side.

The biggest advantage of the SOAP extension technology of .NET is that the client stays unchanged, which is very important for complex applications because of high costs of retesting and redebugging. With SOAP extensions .NET infrastructure's behavior is changed, but not the application's behavior.

3.3 Generation of Transformation Scripts

Message conversion approach described above requires a developer to create XSLT scripts that define how incompatible messages must be brought to correspondence with a new communication contract. To create XSLT scripts correspondences between source and target message schemas must be found. This operation is knows as schema mapping. Mapping can be specified by a domain expert or by an automated schema matching algorithm. Manual matching even supported by a graphical user interface is a tedious, time-consuming and error-prone process (Rahm and Bernstein, 2001). On the other hand automated algorithms do not perform well in all situations and may produce incorrect mappings. Although there are many algorithms, none of them produces fully reliable results (Legler and Naumann, 2007). A combination of the both approaches, i.e. manual matching supported by automated algorithms, represents a reasonable trade-off (Bernstein et al., 2006).

Once mappings have been found, they must be interpreted as data transformation in terms of XSLT language. During this step developers can be supported by a wide range of software products that create XSLT script for a given mapping, e.g. Altova Map-Force, BEA WebLogic Workshop, IBM WebSphere, Stylus Studio, and Microsoft BizTalk Mapper.

3.4 Achieved Results

This subsection presents the results achieved with the help of the developed prototype. As described in the Section 3.2 the main responsibility of the application was to convert messages from ECS 3.0 to ECS 4.0 format and vice versa.

The listing below shows a request message[5] invoking the *KeywordSearchRequest* operation of ECS 3.0. In response to this message the service would send all books with specified keyword in their title.

```
<Body>
 <KeywordSearchRequest>
  <KeywordSearchRequest href="#id1" />
 </KeywordSearchRequest>
 <KeywordRequest id="id1"
      type="KeywordRequest">
  <keyword type="string">C#</keyword>
  <mode type="string">books</mode>
  <type type="string">lite</type>
 </KeywordRequest>
</Body>
```

The newer service version expects different messages to invoke the same functionality. ECS 4.0 request messages differ in both structure and content. Here is an example of a valid ECS 4.0 request message that gets back the same information.

```
<Body>
 <ItemSearch>
  <Request>
   <Keywords>C#</Keywords>
   <SearchIndex>Books</SearchIndex>
   <ResponseGroup>Medium</ResponseGroup>
   <ResponseGroup>Images</ResponseGroup>
  </Request>
 </ItemSearch>
</Body>
```

From the listings one can see that ECS 4.0 uses document-literal style of formatting SOAP messages, whereas ECS 3.0 uses document-encoding style. Due to this fact the messages have different structure. In ECS 3.0 the parameters are factored out from the operation element. Only a reference to another element containing the values of parameters is included in the operation element. On the other hand ECS 4.0 message has all parameter values embedded in the operation element. So the first step of message conversion is to bring ECS 3.0 message to document-literal style. The second step is to map operation name and input parameters. If direct mapping exists (e.g. *mode* corresponds to *SearchIndex*) then the value of parameter is simply copied into the new message under another name. If no direct mapping exists the transformation script has to compensate the difference by inserting appropriate parameters into the new message. The following XSLT script does required conversion.

```
<stylesheet>
 <key name="requestId" use="@id"
```

[5]For the ease of reading namespace attributes in this and the following XML fragments have been omitted.

```
      match="KeywordRequest"/>
 <template match="//Body">
  <Envelope>
   <Body>
    <ItemSearch>
     <apply-templates
      select="key('requestId',
      substring-after(@href, '#'))"/>
    </ItemSearch>
   </Body>
  </Envelope>
 </template>
 <template match="tns:KeywordRequest">
  <Request>
   <Keywords>
    <value-of select="keyword"/>
   </Keywords>
   <SearchIndex>
    <value-of select="mode"/>
   </SearchIndex>
   <ResponseGroup>Medium</ResponseGroup>
   <ResponseGroup>Images</ResponseGroup>
  </Request>
 </template>
</stylesheet>
```

The response messages of ECS 3.0 and ECS 4.0 have the same structure. Therefore, only parameter name mapping must be performed. This is done in the same way as described above for the request messages: the value of parameter is copied into the new message under another name.

4 RELATED WORK

In (Erlikh, 2000) Erlikh estimated that 90% of software costs are evolution costs. The importance of the evolution requires a systematic approach of managing an evolving software system. This is the task of configuration management discipline (Zeller, 1997).

The most frequently used approach in the area of software evolution is versioning. Versioning is used to distinguish different versions of components and libraries that are simultaneously running at the same machine (Sommerville, 2007). The way how a version is identified and which characteristics are included into the computation of version identifier are defined by a particular versioning model (Conradi and Westfechtel, 1998).

A number of versioning methods has been practically implemented. None of them, however, has solved the challenge of consistent software evolution (Stuckenholz, 2005). Moreover versioning is not the mechanism of incompatibility resolution and does not facilitate software substitutability. It is rather a way to make software changes detectable in client applications. To figure out if two versions of the same component are substitutable an approach offered in (Lobo et al., 2005) could be used.

In contrast to versioning this article offers an approach to incompatibility resolution when the older versions of a service are depreciated before existing clients migrate to the newer versions of the service.

The works of Ponnekanti and Fox (Ponnekanti and Fox, 2004), Hohpe and Woolf (Hohpe and Woolf, 2004) and Kaminski, Litoiu and Mueller (Kaminski et al., 2006) address the incompatibility problem. (Kaminski et al., 2006) suggests to pass calls of older clients through a chain of adapters that compensates the difference between the versions of a service in terms of other operations available in the newer version. This solution is more powerful than message conversion, but is limited to the service side and might result in a serious performance hit in case of long chain. (Ponnekanti and Fox, 2004) suggests a similar technique that reconciles incompatibility inside a client-side proxy. Instead of a standard proxy a ”smart” proxy that bridges the gap between the older client and the newer version must be used. The approach is limited to the client side and requires changing the older application. (Hohpe and Woolf, 2004) presents message conversion as a pattern of enterprise integration. The work is fairly abstract without any implementation guidance.

From the business protocol standpoint the evolution of a Web service is described in (Ryu et al., 2007). The article suggests an approach to manage the protocol instances running according to the old protocol version. Firstly, a protocol manager selects the active instances that can migrate to the new protocol. This is done by analyzing the protocol itself (static analysis) and each individual instance of the protocol (dynamic analysis). All migrateable instances can be safely switched to the newer protocol version. Secondly, for non-migrateable instances an adapter must be developed. In case developing the adapter is not feasible an individual temporary protocol must be introduced to the instance to meet new requirements without cancelling the ongoing instance.

Hence, none of the compared approaches is able to guarantee seamless evolution of software and a Web service in particular. Some of the approaches can help to detect and mitigate incompatibility.

5 CONCLUSIONS

Due to platform-neutral nature based on open standards WS have gained high number of supporters. However, along with benefits WS pose some challenges. One of them is the challenge of change management. The lack of techniques in this area results in incompatibility between different versions of the same service. Resolving this incompatibility is sufficient in the short run. In the long run a service provider must develop methods of ensuring the substitutability of different service versions.

This article focused on the short run problem of resolving incompatibility between different versions of the same service. To find a solution the roots of the problem were investigated. Different types of changes that can result in backward incompatibility during the evolution of a Web service were discussed. If incompatible changes happen, message conversion can be applied. This solution is based on the translator pattern described in (Hohpe and Woolf, 2004). The current work contributes to already known knowledge with a new practical implementation of the translator pattern on the .NET platform. The major challenges and limitations of the approach were outlined. With the help of a .NET prototype the results of the research were checked against the last two versions of E-Commerce Service from the Amazon Web services suite. The future work of the research shall concentrate on developing design techniques to guarantee the substitutability of different versions of a Web service.

REFERENCES

Bernstein, P. A., Melnik, S., and Churchill, J. E. (2006). Incremental schema matching. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1167–1170.

Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282.

Dahan, U. (2006). Autonomous services and enterprise entity aggregation. *The Architecture Journal*, (8):10–15.

Erlikh, L. (2000). Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23.

Esposito, D. (2006). *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics*. Microsoft Press.

Hall Gailey, J. (2004). *Understanding Web Services Specifications and the WSE.* Microsoft Press.

Hohpe, G. and Woolf, B. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Kaminski, P., Litoiu, M., and Mller, H. (2006). A design technique for evolving web services. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*.

Legler, F. and Naumann, F. (2007). A classification of schema mappings and analysis of mapping tools. In *12. GI-Fachtagung fr Datenbanksysteme in Business, Technologie und Web*, pages 449–464.

Lobo, A. E., Guerra, P., Filho, F. C., and Rubira, C. (2005). A systematic approach for the evolution of reusable software components. In *ECOOP'2005 Workshop on Architecture-Centric Evolution (Glasgow, UK, 25-29th July 2005)*.

Lublinsky, B. (2007). Versioning in soa. *The Architecture Journal*, pages 36 – 41.

Ponnekanti, S. R. and Fox, A. (2004). Interoperability among independently evolving web services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware (Toronto, Ontario, Canada, Oct. 18-22, 2004)*, volume 78, pages 331 – 351.

Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *The International Journal on Very Large Data Bases*, 10:334–350.

Ryu, S. H., Saint-Paul, R., Benatallah, B., and Casati, F. (2007). A framework for managing the evolution of business protocols in web services. In *Proceedings of the 4th Asia-Pacific Conference on Comceptual Modelling*, volume 67, pages 49 – 59.

Sommerville, I. (2007). *Software Engineering*. Addison-Wesley, 8 edition.

Stuckenholz, A. (2005). Component evolution and versioning state of the art. *ACM SIGSOFT Software Engineering Notes*, 30(1).

Zeller, A. (1997). *Configuration Management with Version Sets - a Unified Software Versioning Model and its Applications*. PhD thesis, Technische Universitaet Braunschweig.