

White Paper



Service Versioning in SOA

Part I: Issues in and approaches to Versioning

Deepti Parachuri and Dr. Sudeep Mallick

Abstract

Services evolve dynamically to address the changing business context. This evolution leads to multiple service versions. In this scenario, service versioning becomes a priority to minimize / eliminate the impact of changes made to the service, on the service consumer's experience.

Service versioning is an unavoidable reality, as the service provider cannot expect all service consumers to change every time a service changes, at least in the short-term.

This paper, the first of two parts, addresses the problems of service versioning from the perspectives of both the service provider and the service consumer. It (Part I) discusses the change management concerns, highlights the issues involved in service versioning and approaches to accomplish versioning successfully.

Introduction

In real world applications, web services need to evolve to meet changing functional and non-functional requirements, leading to multiple versions of the original service [1]. As change management is the key to successful software development (including web service governance), supporting multiple service versions has become a major concern for SOA practitioners, especially when consumers and providers have independent lifecycles [2].

A service needs to be versioned as it is unrealistic for the service provider to expect all service consumers to change every time a service changes. This begets the question - *How do you make an SOA implementation cope successfully with versioning and change management?*

One obvious answer is loose coupling, an inherent property of an SOA implementation. Loose coupling gives enterprises flexibility, but loose coupling does not mean any change made to the implementation won't impact other parties.

In our versioning framework, we address the problem of service versioning keeping in mind both service providers and service consumers. We also try to answer a few basic questions that are important to understand service versioning requirements:

- What do we actually want to version - *a service contract or a service implementation?*
- What is meant by backward and non-backward compatible changes? How are they handled?
- Should a consumer know whether the implementation behind the interface has changed?
- When should we go for in service versioning? And when for a new service?
- What are the aspects of a service which are liable to change and hence, call for a versioning system?

In our attempts to answer these questions, we identify various aspects of versioning business functionality change, service contract change and service implementation change. Finally, we study the various versioning techniques available and propose the best possible solution for different scenarios.

In Part I (this paper) of this two-part series, issues in service versioning and approaches to handle service versioning have been discussed. Section 2 explains different types of compatibilities. Section 3 covers the different service change scenarios that call for versioning. Section 4 deals with different versioning approaches and schemes for implementing the version changes.

Part II of this series will deal with the different approaches to handling the architectural impact on an SOA caused by versioning and a few other issues related to public web services and SaaS.

Compatibilities

Changes made to services can be of two types - *backward compatible* and *non-backward compatible* [3].

Backward compatible means a newer version of the service can interpret requests from consumers of older versions without breaking the service. Backward compatible changes can be

- Correcting a service implementation, e.g., a bug without any change to the service contract
- Interface modification due to addition or deletion of parameters to support an implementation change or through service enhancement. Fundamentally, this does not involve a change to the semantic contract of the old interface, so long as the changed parameter is made optional
- Service implementation change (with no change to service interface) due to change in the business rules

Non-backward compatible means the newer version of the service cannot interpret the data formats of older versions. Non-backward compatible changes can be

- Removing an operation
- Renaming an operation
- Changing the parameters (in data type or order) of an operation
- Changing the structure of a complex data type.

There is one more school of thought where they talk about *forward compatibility*. Forward compatibility [3] is the ability of a system to accept inputs intended for later versions of itself. Forward compatibility can be achieved by adopting standards designed to ignore unrecognizable elements that may appear in future.

Different aspects of versioning

Changes can happen at different stages in the service lifecycle, such as service identification, design, development, deployment and runtime. The changes can occur in different aspects of the service concept:

- Business functionality change
- Service contract change
 - Interface change
 - Data type change
 - Message change
 - Operation change
 - Binding change
 - Policy change
- Implementation change

1. Business service functionality change

Business service changes deal with changes in functionality. Different types of service functionality changes exist core behavior change, semantics change, orchestration level change etc. In most of these cases, it makes sense to arrive at a new service rather than a version. But if the script embedded in a BPEL changes (or a service it consumes changes) then perhaps it is a simple version level change, rather than a new service. So, a new service is created when it offers a different/distinct functionality compared to existing functionality or its semantics is different or the domain addressed is different.

2. Service contract change

Service contract changes can be classified into interface changes and policy changes (Table 1).

Area of change	Backward compatible	Non-backward compatible
Business service functionality change		
Service contract change (interface and policy changes)	Interface Changes: Adding an operation Adding new optional data structures to the input message Policy Changes: Change in Quality of Service (QoS) Change in agreed domain values in the output message	Interface Changes: Removal of operation, Changing the cardinality of output message Adding extra data structures (parameters/attributes) in the output message Changing the definition of data types Policy Changes: WS-policy changes like changes in security and reliable messaging
Service implementation change (Implementation changes)	Simple fixing of bugs	Changes in implementation which effect the interface

Table 1 Different aspects of service versioning

2.1 Interface changes

We attempt to classify different interface changes as backward compatible (BC) and non-backward compatible (NBC) changes and suggest solutions.

2.1.1 Data type change: Date type of the input/output parameter changes it's an NBC change.

2.1.2 Message changes:

- Adding extra data structures (parameters/attributes) in the output message is an NBC change
- Adding extra data structures (parameters/attributes) in the input message can be considered as a BC change by making the extra parameters/attributes as optional
- Changing the order of parameters/attributes is again an NBC change
- Changing the cardinality of output message is an NBC change

2.1.3 Operation changes:

- Adding an operation is a BC change
- Removing an operation is an NBC change

2.1.4 Binding changes:

Binding changes are protocol changes (SOAP, JMS etc) which are NBC changes, but can be made BC.

2.2 Policy changes

Design rules combined with enforcement are called “policies.” Policies typically govern under what conditions consumers are entitled to access service functionality [8]. *Rules of engagement* between clients and providers are called *policies*. Policies govern who can access a service, what security procedures the participants must follow and any other rules that apply for the exchange between provider and consumer. The difference between a contract and a policy is that - a policy is a set of conditions that can apply to any number of contracts. For example, a policy can range from simple expressions informing a client about the security tokens that a service is capable of processing (such as Kerberos tickets or X509 certificates) to a set of rules evaluated in priority order that determine whether or not a client can interact with a service provider. Here, we try to identify various kinds of changes that can happen in a policy

- Change in QoS
- Change in agreed domain values in the output message
- WS-policy changes
 - Security
 - Reliable messaging

Backward compatible changes

Changes in QoS parameters like response time and availability though backward compatible will have an adverse effect on existing consumer. Service provider may implement some change that does not affect the interface but will have an effect on service quality [5]. Another change which may affect policy is changing the agreed domain value in the output message. Consumers may have an expectation about the domain values of certain fields in the response message. If the service provider sends values beyond the accepted values, it might have an adverse effect.

Non-backward compatible changes

WS-policy changes [6], like security changes and reliability changes, can prevent existing users from continuing their service with the updated service. Now, let us consider security changes, consider an existing web service is using an username token as a security token and the updated web services changes security token type to Kerberos, then the existing consumer will face issues. Now if we take the case of reliable messaging [7], the protocol for delivery assurance gives four basic types, AtMostOnce, AtLeastOnce, ExactlyOnce and InOrder. If in the policy we change the delivery assurance say from AtLeastOnce to ExactlyOnce, the client has to change the implementation at their end.

3. Service implementation change

Implementation changes are mostly backward compatible changes as the service contract is not violated.

Backward compatible changes:

Solutions for tackling backward compatible implementation changes

Silent

One solution for this kind of change is to keep “silent” about the change as there is no impact on the service interface. Simply deploy the changed service implementation and decommission the original service implementation. The modified implementation will then be bound to the original service-interface definition. It is up to the service provider to decide on “whether a consumer needs to know if the nature of the service behind the immutable interface has changed?”

Version identifier

Another solution is to use a change/version identifier in the service interface headers. This approach may be realized by using the identifier in the service interface headers. This identifier can provide a predefined measure of the change in a specific interface. As part of the service definition, a parameter may be defined as follows:

```
<message name="WeatherReport">
  <part name="version_detail" type="xsd:string"/>...
  other parameters...
</message>
```

By predefined, we mean that the service documentation will enumerate the different values that the identifier can take. For example, the parameter “version_detail” may be a concatenation of two numbers, one indicating whether the service has changed or not, and the other indicating the version reference number. Consumers of the service should be capable of obtaining the interface change/version details.

Advantage: Change is communicated to the consumer.

Disadvantage: There is an overhead of an extra output parameter. There is an implicit assumption that the consumer is capable of reading the version details and taking decisions.

Multiple interfaces

Another solution, though sub-optimal, is to create multiple interfaces for every change in service. The polymorphic behavior can be achieved either by having more than one implementation of the interface or by implementing a new service and diverting the requests based on SOAP headers.

Disadvantages: The overhead of maintaining extra interfaces.

Recommended

We recommend the methodology with the version number that will provide information about changes to service consumers. The service can specify a period beyond which the change identifier will not have a relevant value so that this output parameter may be used continuously for multiple changes. This methodology is recommended because it strikes a balance between the amount of additional work required and the need to communicate relevant changes to service consumers.

Versioning approaches

Here, we present different versioning approaches.

1. [Document style versioning](#)

Document style versioning is a primitive style of versioning abstract WSDLs, wherein the version details are stored in the documents and no intelligence is present to manage versioned services. It is used when there are not many services to version and there is no requirement to govern the versioned services. It is the simplest form of versioning.

2. [Namespace versioning](#)

A specific namespace value is sent along with every SOAP message. Based on the namespace value, the service implementation determines what to do with an incoming message. If non-backward compatible changes need to be made to a WSDL document, then make sure that the namespace for the XML elements resulting from that document is unique. To ensure that the various editions of a WSDL document are unique, one way is to append date or version stamp to the end of a namespace definition. This follows the general guidelines given by the W3C for XML namespace definitions.

Advantages: No requirement of a complex registry.

Disadvantages: Interpreting the header and routing the message to the required endpoint is tough.

3. [XML versioning](#)

Though WSDL, in principle, can be based on any schema language, it usually uses XML schema [9] as its schema language. WSDL by itself does not contain any versioning information (nor UDDI or SOAP). Hence, versioning WSDL depends on the user and the capability of versioning WSDL lies in the management of schema information. The design of open and extensible schema concepts can be used to implement versioning. An open schema allows unknown data to appear at certain points without knowing the exact schema of the content. An extensible schema is designed mainly with versioning in mind and provides a way to extend the schema.

[XML schema versioning approaches](#)

There are different approaches for schema versioning [10]

1. Change the (internal) schema version attribute.
2. Create a schema version attribute on the root
3. Change the schema's target namespace.
4. Change the name/location of the schema.

4. [UDDI-based versioning](#)

UDDI is a public registry built to maintain services in a structured way. UDDI is for publishing and discovering information about a business and its services. This data can be classified using standard taxonomies, so that information can be found based on categorization. UDDI also contains information about the technical interfaces of a business's services. Using SOAP messages, one can interact with UDDI at both design time and run time to discover technical data to invoke and use the services. It is a good practice to maintain services using a registry rather than localization of services. UDDI helps in having structured organization and also for maintaining interoperability in discovery mechanism.

If we have to implement a breaking change, the best way is to notify existing consumers about it. A cleaner approach over service broker would be to use UDDI [4] to communicate the change in the metadata used to describe the service. Though UDDI specifications provide guidance for supporting multiple versions of an API, service versioning is not directly addressed. The UDDI tModels are extensible, enabling service versioning to be included. There are two approaches to service versioning in UDDI:

[Adding a version number to the tModel:](#)

tModel contains a structure known as tModelInstanceInfo which in turn contains instanceDetails. The instanceDetails data structure is extensible and allows us to add version number for the service being registered. Adding a version number to instance details enables tModels to communicate service versioning information along with other information used to describe the service. The drawback is that the user needs to be aware of the newly added element and also green page interfaces need to be extended to enable search on service version number.

Having multiple tModels:

Services can expose multiple interfaces supporting more than one version of the interface. Services compatible with different versions will refer to the appropriate tModel in the tModelInstanceDetails collection

The second approach of using multiple tModels is more effective than adding an identifier, version number, in tModel.

Disadvantages of using Tmodels: If several service implementations exist, then several tModel collections must be maintained making the UDDI registry complex and difficult to use.

5. Compound versioning

In real cases, changes in WSDL can happen in multiple ways. Compound versioning [11] assumes simultaneous existence of multiple interfaces and implementations of the same business functionality exposed as a service. The interface provides end points and definitions that consumers use to access the functionality exposed by the application and the implementations of the service adhere to the contract specified by the interface. There could be multiple implementations of the same business service, with every implementation being distinguishable and individually addressable. The possibility of simultaneous changes in the service interface and service implementation, calls for having more sophisticated versioning schemes, such as compound versioning. Based on the two individual versioning life cycles, the idea here is to build a compound service versioning scheme consisting of a combination of version changes of the service interface and the service implementation.

References

1. Ru Fang, Linh Lam, Liana Fong, David Frank, Christopher Vignola, Ying Chen and Nan Du, A Version-aware Approach for Web Service Directory, ICWS 2007, ISBN: 0-7695-2924-0, pp 406-413.
2. Erik Wilde, Semantically Extensible Schemas for Web Service Evolution, ECOWS 2004, LNCS 3250, pp 30-45, 2004.
3. John Evdemon, "Principles of Service Design: Service Versioning", <http://msdn.microsoft.com/en-us/library/ms954726.aspx>, August 2005.
4. Versioning XML vocabularies at <http://www.xml.com/pub/a/2003/12/03/versioning.html>
5. IBM's "Moving forward with web services backward compatibility" at <http://www-128.ibm.com/developerworks/java/library/ws-soa-backcomp/>
6. WS-Policy at <http://dotnet.org.za/stuartg/pages/6277.aspx>
7. Web Services Reliable Messaging Protocol at <http://www.ibm.com/developerworks/library/specification/ws-rm/>
8. Grappling with SOA Change and Version Management at <http://www.zapthink.com/report.html?id=ZAPFLASH-2006519>
9. Fallside, D.C., Walmsley, P., XML Schema Part 0: Primer Second Edition at <http://www.w3.org/TR/xmlschema-0/>
10. XML Schema Versioning at <http://www.xfront.com/Versioning.pdf>
11. Service Versioning for SOA at <http://soa.sys-con.com/node/250503>

About the Authors

Deepti Parachuri is a Junior Research Associate at the SOA Centre of Excellence at SETLabs, Infosys' research group. Her research areas include semantic web, SOA governance and service versioning.

Dr. Sudeep Mallick is Principal Researcher at the SOA Centre of Excellence at SETLabs, Infosys' research group. He has nearly 11 years of experience in development of distributed enterprise applications, technology evangelism and applied research. His current research interests include SOA governance, composite applications and distributed software engineering.



For more information, contact askus@infosys.com

About Infosys

Many of the world's most successful organizations rely on Infosys to deliver measurable business value. Infosys provides business consulting, technology, engineering and outsourcing services to help clients in over 30 countries build tomorrow's enterprise.

For more information about Infosys (NASDAQ:INFY), visit www.infosys.com.