

White Paper



Service Versioning in SOA

Part II: Handling the impact of 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.

In part I, we discussed change management concerns, highlighted the issues involved in service versioning and approaches to accomplish versioning successfully.

This paper, the second of two parts, discusses the architectural impact of versioning and the different techniques required to handle the impact.

For more information, contact askus@infosys.com

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 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].

Part I of this two-part series analyzed the different service change scenarios and the techniques required to implement the changes. This paper (Part II) analyzes the different techniques for managing the architectural impact of change. It discusses techniques and patterns for isolating and minimizing the impact of change from the perspectives of the service provider and the consumer.

This is followed by a discussion on service deprecation which is a key governance concern. Then it addresses versioning techniques for public web services and SaaS, as enterprises are no longer limited to consumption of internal services.

Approaches to handling the impact of version change

1. At server-side

Versioning at the server-side can be handled in two ways:

1.1 Using XML extensions

The XML schema type extension feature is used to handle a few of the non-backward compatible changes in a way that makes them backward compatible. For example, W3C XML Schema provides the wildcards `xs:any` and `xs:anyAttribute` which can be used to allow the occurrence of elements and attributes from specified namespaces into a content model. The wildcard indicates that elements in specified namespaces are allowed in instance documents where the wildcard occurs. This allows for later extension of a schema in a well-defined manner.

1.2 Using service adapters

Service adapters can be created using XSL style sheets to transform the SOAP payload [1]. An older request, say a V1.0 message from a requester is processed by an XSLT engine and transformed into a new version request message say V1.1, which can then be consumed by the new service version (V1.1). The opposite is done for the response message, where the V1.1 response from the service is transformed into a V1.0 response message.

2. Using a middleware

There are various methods of handling versioning using a middleware:

2.1 Using web service brokerage firms

Enterprises can publish their WSDLs in web service brokerage firms [2] like www.salcentral.com and www.xmethods.com. Web services brokerage firms like

www.salcentral.com can dynamically scan WSDL schemas looking for changes in structures. Once a change is found, the service e-mails or sends an SMS stating that a change has taken place. But, it may not use the standard way of receiving notifications because different service brokerages will implement different ways of notifying changes.

Disadvantages: There isn't one accepted, standard brokerage firm. Security is also a concern.

2.2 UDDI subscription-based approach:

UDDI v3 has a built-in feature called *Subscription* [3] which can be utilized to notify service changes to subscribers. Until UDDI v3 came up with this feature, there was no standard way of notifying consumers about changes in WSDL. According to the UDDI V3 specification, “*Subscription provides clients, known as subscribers, with the ability to register their interest in receiving information concerning changes made in a UDDI registry*”.

Recommendation

A UDDI-based solution is a neat and standard way of handling versioning in services. When there are many services (say more than 30) then UDDI is the correct choice. When there are only a few services (under 30) then handling at the server side or client side is the right choice.

2.3 WS-addressing for version management

WS-addressing can be used for tackling issues related to versioning in web services [4]. WS-addressing can also be used to address scenarios in which the endpoint URL of a web service changes. WS-addressing can be used to solve the issue by adding a custom tag to indicate changes in implementation. For example, we can add a tag named <implementation> in a response message to indicate a change in implementation logic. Whenever the implementation changes, information pertaining to the specific change is included in this tag for a certain period of time. When there is change, an empty implementation tag is provided. Similarly, the consumer can include an implementation tag that indicates the last-accessed implementation version. If the implementation has changed since the last client request, the service can send the required information. Similarly, when there is an interface change and the consumer request does not match the required interface definition; WS-addressing can be used to handle the change by including some custom header tags that indicate an interface change.

Disadvantage: Implementation is a custom-defined tag. Hence this tag must be used only after an agreement between the client and the service provider. The SOAP engine should also support WS-addressing.

Up to this point, we have dealt with situations where a consumer is directly talking to the provider. Now, we look into situations where the provider uses a broker or registry to publish the WSDLs.

Methodology	Advantages	Disadvantages
Forward compatibility using extensions [Part I - #2]	Service is compatible with a newer version of itself by ignoring unknown tags and extensions	We cannot predict future changes
Namespace versioning [Part I - #3][12]	No requirement for a complex registry. CORBA and DOTNET have APIs for naming resolution and binding [13]	If SOAP engines do not have support, an intermediate router is required to redirect the SOAP message to the new end point. Clients do not have knowledge of the service implementation version
Service versioning covenant	No requirement of a complex registry	“If, else” covenant can grow out of proportion and it may become un-maintainable Clients do not have knowledge of the service implementation’s version
WS-addressing for version management [4]	No requirement for a centralized registry or even an intermediate router	The SOAP engine should support WS-addressing
Web service brokerage firms [2]	Can automatically send notification to the client if a change occurs in the registered service	There is no agreement on one standard brokerage firm among consumers.
UDDI-based versioning [Part I - #1] [12][3]	When many services exist, a registry makes maintenance of services easier. Using the Subscription API, notifications can be sent when a service changes	If several service implementations exist, then several tModel collections must be maintained. UDDI registry becomes complex and difficult to use.

Table 1 Summary of different service versioning approaches

2.4 Using policies

2.4.1. Service versioning covenant

A covenant is an “If Then Else” agreement. It is a much more realistic expression of how software actually works. The service is saying that “if you send me X, I will do Y. The service should agree that when we send it a message with a specific schema, it will perform a certain action. If we send it a different schema, a different action will occur.

Disadvantages: The “If Then Else” covenant can grow out of proportion and it may become un-maintainable. Soon, we will have an application that is tailored to one customer. Also the client does not have information about the version of the service component behind the interface.

2.4.2 Policy enforcement mechanisms

Policy enforcement mechanisms [5] enable policy-based version selection at runtime. Policies are enforced at runtime by a Policy Enforcement Point (PEP) which ensures that the defined rules have been obeyed. A policy enforcer can be used as a middleware to interpret messages and route messages to the required version of the service. A policy enforcer supports routing of messages.

2.5. Enterprise Service Bus (ESB)

ESB is a traditional way of handling versioning [6]. The advantage of using an ESB is that the client sees only the virtual endpoint on the ESB. The original service endpoint is hidden. Clients send requests to this virtual endpoint hosted on the ESB, allowing the ESB to route the message to the appropriate version of the service. This approach requires that there is

sufficient information in each message to perform the routing. Version number of the required service can be included in the body of the message or in the SOAP header. For example, a WS-addressing end point reference could be used to contain this information. Messages will be routed independently, so the version information needs to be included in each and every message. When more than one version of a service is available then the routing decision can be done based on the policy or versioning strategy defined at the time of creating services.

2.5.1 Versioning of CIM

Another issue when using ESB is versioning of CIM (Canonical Information Model). CIM is a controlled and governed, centralized data model that defines the message model inside an ESB. It is possible that the data model (domain-level data or application-level data) may change after the CIM is developed and deployed [7]. The versioning of the CIM impacts the web services server and client sides in terms of message transformation changes. This also possibly impacts the service interface and web services client side code leading to both upstream and downstream versioning issues. Migration strategies need to be created to make the data compatible with CIM.

3. At client side

Clients should be able to switch between versions freely, and invoke any version of the service without rebuilding their code. Version-transparency can be achieved through usage of service proxies. Proxies are generally bound to a selection policy/strategy and update their target service version when there is a better match than their current service version. Proxies are responsible for mediating between the user-provided data and the expected input of the target service version. Change types affecting only optional WSDL parameters, as well as added operations or changed output parameters (backward compatible changes) can be handled transparently, while non-backward compatible changes are generally non-transparent. DAIOS [8] is one such framework which decouples clients from the services by abstracting service implementation issues such as encoding styles, operations or endpoints. But this can handle only backward compatible changes. For non-backward compatible changes client stubs need to be rebuilt.

Version deprecation

Deprecation [11] of the old version of a service is as important as the deployment or publishing of it as different versions evolve over time. Deprecation is not as simple as deactivating the service; it is a process of notifying consumers in advance about the deactivation, not accepting requests for such services or rerouting the request to a new service, acknowledging incorrect client users, suspending the service for a stipulated period, running the deactivated service for a period and acknowledging the state.

Solution for version deprecation

To retire an older version of a service, we should intimate users about the expiry date of the older version and the availability of newer version sufficiently in advance. This can be implemented by using a version identifier as suggested in section 3.3; for instance a tag named <serviceExpiry> can be included in a response message to notify users of version expiry. Even subscription and notification APIs of a centralized registry (UDDI or Brokerage firms) can be used to inform existing clients of service deprecation.

Versioning in public web services and SaaS

Versioning is a common problem faced by public web services and SaaS applications. Public web services like Amazon Web Services insert their API version into the URL [9]. Backward compatible changes, meaning changes that do not affect customers using the API are done by simply showing the version within the XML. However, for larger version upgrades customers can choose the calling URL (api1.domain.com, api2.domain.com etc).

SaaS deployments are not “deploy once and forget” applications [10]. SaaS applications are likely to be upgraded and changed over time to provide new functionality. Customers will need to upgrade their contracts when new functionality is introduced. To use a SaaS application, a customer needs to subscribe to the SaaS application. The customer is granted a subscription (contract that authorizes the customer to access the application). Once a customer has purchased subscriptions, processes should be provided to update the contracts when the SaaS application introduces new functionality or users want to buy additional blocks of features. To handle the above scenarios, products like SaaSGrid provide mechanisms for application and version management, customer contract management etc.

Conclusion

Service evolution/versioning is an integral part of the service lifecycle. In service-oriented computing, versioning is an important issue as changes in services directly affect service consumers. This two part series identified various issues and aspects of service change management and more specially service versioning. The first part analyzed the different scenarios of service change in an SOA and techniques for implementing those changes. The second part (this paper) discussed the ever important topic of techniques and patterns to handle (isolate and minimize) the impact of service change on the solution architecture.

References

1. Designing and versioning compatible web services http://www.ibm.com/developerworks/websphere/library/techarticles/0705_narayan/0705_narayan.html
2. Romin Irani , “Versioning of Web Services”, <http://www.webservicesarchitect.com/content/articles/irani04.asp>
3. UDDI subscription based approach at <http://www.developer.com/services/article.php/3374631>
4. Sriram Anand, Krishnendu Kunti, Mohit Chawla and Akhil Marwah, ” Best practices and solutions for managing versioning of SOA web services” at <http://soa.sys-con.com/node/143883>
5. HP policy enforcer at https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-15-27^1408_4000_100
6. Binildas C. A Service Oriented Java Business Integration, PACKT publishing Ltd, March 2008.
7. SOA - Common Information Model (CIM) - Part 2 at <http://it.toolbox.com/blogs/the-real-soa/soa-common-information-model-cim-part-2-18444>
8. DAIOS at <http://www.vitalab.tuwien.ac.at/~florian/papers/TUV-1841-2007-01.pdf>
9. Versioning web services at http://kalsey.com/2006/02/versioning_web_services/
10. Versioning for Your SaaS Operations at <http://www.apprenda.com/SaaSGrid/?content=versioning>
11. Grappling with SOA Change and Version Management at <http://www.zapthink.com/report.html?id=ZAPFLASH-2006519>
12. Kyle Brown, Michael Ellis, “Best practices for Web services versioning”, <http://www.ibm.com/developerworks/webservices/library/ws-version/>, Jan 2004
13. Meijer E., Szyperski C, “Overcoming Independent Extensibility Challenges”, Communications of the ACM, Vol. 45 No. 10, October 2002.

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.