RESERVING IMMUTABLE SERVICES THROUGH WEB SERVICE IMPLEMENTATION VERSIONING

Robert Steele, Takahiro Tsubono
Universit of Technology, Sydney, PO Box 123, Broadway 2007, Sydney, Australia
Email: rsteele@it.uts.edu.au, ttsubono@it.uts.edu.au

Keywords: Web Services, versioning, version control

Abstract: Widespread adoption of a Web services-based paradigm for software applications will imply that applications will typically have potentially many dependencies upon Web services that they invoke or consume. These invoked services might typically be available from a remote site and be under the administration of third parties. This scenario implies a significant vulnerability of a Web services-based application: one or more of the services which it consumes may become altered, hence potentially "breaking" the application. Such alterations might be such as those that alter the WSDL signature of the service or could be changes to the underlying service implementation that do not change the WSDL signature. In this paper, we will focus on the second of these two cases and will introduce a versioning system that can detect changes to service implementations and that can avoid the breaking of applications that call services in the face of changes to the implementations of those called services.

1 INTRODUCTION

If the Web Service or Service Oriented Computing paradigm is to attain widespread adoption there will need to be a solution to the potential fragility of Web service-based applications arising through their dependencies on called Web services. This is particularly true for shipped applications that can no longer be modified to accommodate any changes to Web services that they call.

There are two types of changes that can occur to Web services. One type of change involves a change to the service's WSDL signature: that is, a change to the service's set of operations or to their parameters or parameter types. A second type of change involves changes to the service's underlying implementation that leaves the WSDL signature unaffected.

Although there are many different causes that may break the client applications, changes to the service implementations are significantly risky which directly influence behaviour of the services. DLL Hell (Eisenbach, et.al., 2002) was a major issue that broke other applications in response to application installations and updaters replacing shared resources. The problem was not only caused by inherent software entities, e.g. libraries and executables, being swapped with their old versions, but also by new versions producing backward-incompatibility or new bugs (Anderson, 2000).

The solution of DLL Hell was to provide each application with the infrastructure of reserving dependent software entities that are never updated by other applications. Having learnt from this experience, the Web service implementation versioning proposed here reduces risks of "service hell" for both service providers and requestors by means of allowing client applications to continue to use the same versions of services.

1.1 Motivation

A new version of an existing service often replaces the old version rather than leaving two versions of the same services operational simultaneously. This is due to various reasons such as added cost of maintenance. The old service requestors assume that service providers never break the existing service calls. However, such an assumption cannot be guaranteed in the scenario where existing service calls use new service implementations that client applications have not been built and tested against.

Regardless of dynamic or static binding, if the previous version of the service is unavailable,

service requestors do not have a choice other than to upgrade to the new version. However, even when multiple versions are made available, the service calls are redirected to the one specified by service providers, not by service requestors. There is no standardized infrastructure for service requestors to have intervention into the choice of versions.

The business needs vary across service requestors. They may not be willing to upgrade to the new services simply because the current version provides sufficient functionality (Irani, 2001), while some other service requestors may. As the number of service requestors increases, it becomes complicated to make a modification to the existing services in a way all those service requestors involved agree. Leaving the current version of the Web service allows the service provider to be released from receiving unexpected claims resulting from the arbitrary decision to update an existing service.

Currently UDDI (OASIS, 2004) does not support finding a version of services based on the versioning semantics, e.g. meaning of major numbers. The service requestors need to know the relationship between versions prior to choosing the appropriate version of the desired service. However, as the format of versions and the versioning scheme are currently not standardized, thus at the worse case, it could be unique to each service provider. While Web service orchestration grows in popularity in response to its capability of SOA (Service Oriented Architecture), such a non-standardized versioning scheme increases the complexity of configuration for service binding. As an example, if a service endpoint was dynamically chosen from a pool of services that all claim a conformance to a standardized service interface definition in a given industry, a client application may need to apply different version selection mechanisms to each service provider.

Even though a service requestor may see the same WSDL signature, changes may be opaque to service description. As the service implementation is basically invisible to client applications, there is no means for client applications to detect or verify changes to the target service implementation.

On the other hand, service modification also adds to the work of developers by enforcing the backward compatibility when an existing service is to be updated. However, the backward compatibility issue is not only on the service interface description, but also on the semantics of defined data types and the immutability of the returned set of data against all possible formation of inputs on a given state. In this scenario, testing applications in a way functions retain the same output is full of complexity. Moreover, to keep the service interface backward compatible, new service interfaces constrain available modifications to fall into some types such as the addition of operations or new optional parameters to existing operations (Brown and Ellis, 2004; Butek, 2004; W3C 2003). As time goes by, if these constraints are followed, the service interfaces grow in their complexity. For instance, when a deletion of a service operation is denied, modification continues increasing service operations. At some point, the maintenance of all operations and maintenance of the consistency of some of their attributes/policies such as naming convention, becomes complex.

To reduce the number of versioned services in operation, service providers have to own the responsibility for the consequence of the service upgrade. Web service implementation versioning aims to guarantee that service providers do not dispatch existing service calls to a different version of the same service without client agreement.

1.2 Paper Outline

Firstly, the current Web service versioning methods and relevant technologies are described in the next section. In Section 3, an overview of our Web service versioning approach will be presented. The architecture and the procedure for the use of the version-aware Web service server are then described in Section 4. The forthcoming issues are discussed in Section 5. Finally, Section 6 concludes the paper by wrapping up the presented idea and providing proposals for the future work.

2 BACKGROUND

The topic of maintaining the association of an application and its interdependent components is not new. Some researches and vendors addressed the solutions using a versioning strategy. Plasil et. al. (1998) presented SOFA/DCUP (SOFtware Application, Dynamic Component UPdating) - a way to define a set of nested versioned components and to update a specific component at runtime. .NET Framework (Microsoft, 2005) tightly couples applications and the required libraries based on the assembly versions. Java Product Versioning Specification (Sun Microsystems, 2003) defines the method of comprising version details in a package. To the extent of our knowledge, none of the research papers have targeted the versioning of Web services at the time of writing. However this issue is currently active within the Web Service Description Working Group (2005). The relevant technologies and issues are described in the following subsections.

2.1 Version Identification

Where the discussion of Web service versioning is seen, two approaches are often considered to resolve the issues of which elements and layers of application architecture carry version information: an inclusion of the version in (1) namespace (W3C, 1999) of WSDL elements or (2) service endpoint URLs.

The former utilizes a namespace in WSDL that uniquely establishes the logical scope for a group of WSDL elements. Thus, this versioning strategy defines a unique namespace for each version of a service or an operation, for example. As the namespace does not directly deal with the service implementation, it needs to be mapped to an identifier of software entities, which a SOAP (W3C, 2004a) engine can recognise. Web services for J2EE specification (IBM, 2003) defines to map a namespace to a Java package (Gosling, et. al., 2000) that dispatches a service call to an instance of the class drawn from the associated package.

A version appended URL is often seen on the Internet. While the format of a version differs in the purpose of its use, the date-based format is sometimes preferred than a tuple of incremental integers. A benefit of using the date format is that clients can intuitively find the release date, while no indication of version implications hides the relationship between different versions.

Regardless of the formats, URL-based versioning for Web services causes an infrastructural issue for Web services architectures. The problem is in the layer at which URL is evaluated. As for HTTP, a web server is in charge of evaluating incoming HTTP requests and forwards them to wherever the URL is linked to. Thus, if a version selection mechanism exists in this scenario, the web server is in charge of parsing a version. Moreover, as such a server is beyond the control of a SOAP engine, the versioning issue is forced up to the transport protocol. Thus, the version-parsing mechanism must be implemented for every transport protocol. Tasks of version identification could be delegated to a layer below the URL in order to insulate a service endpoint from the versioning issues and, therefore, service endpoints remains abstract representation of service location.

2.2 Change Notification

Apart from the versioning method, clients need to know the occurrence of changes to the service interfaces and implementation. Some proposals have been available for informing the client interested parties of such events.

NSPF (Kalali, et. al., 2003a) and later SOMR (Kalali, et. al., 2003b) are proposals of frameworks for notifying service requestors about the status of services such as availability and changes to service interfaces. Web Services Eventing (Box, et.al, 2004) and Web Services Notification (Akamai, et.al, 2004) are the followers for standardizing such notification architecture.

The latest UDDI version 3 (W3C, 2004b) also implemented Subscription API, which notifies changes in the UDDI registry to its subscribers. However, the UDDI registry maintains a service endpoint and the location of the WSDL rather than the actual WSDL document. As changes in service implementation are not necessarily reflected to the service endpoint or WSDL location, the notification is not invoked without service providers deliberately altering the registered service description. Thus maintaining a WSDL repository as suggested by NSPF and SOMR is still beneficial in terms of detecting changes in service interfaces.

Unfortunately, service providers are still free of restrictions when changing implementations behind the scenes without altering the WSDL. Furthermore, even though non-changed services might be maintained by a service provider, a part of their system may start accessing altered software entities from third parties, should the dependency of those entities be disregarded.

2.3 Version Semantics

A typical format of a version number conveys the interpretation rule and version semantics, by assigning meaning to each segment of a version and by reserving specific letters. Although, a number of well-defined numbering schemes are available nowadays, to the extent of our knowledge, nothing is practically accepted as the global standard. However, many have commonality in ruling the semantics.

From the perspective of those who are affected by version changes, the interest is often in the backward compatibility with the old version. Many publications on versioning distinguish numbers into major and minor portions for this purpose. Typically, major enhancements imply a possibility of backward incompatibility, whereas minor enhancement is backward compatible (Brada, 2000; W3C, 2003).

Vendors concerned with the unit of low-level software entities equip their published programming languages with the ability of expressing precise changes such as revision and build numbers. .NET Framework (Microsoft, 2005) tightly couples applications and the required software entities based

on the assembly versions. Java Product Versioning Specification (Sun Microsystems, 2003) defines the method of comprising a version in a Java package. In either case, the version details are attached to a set of software entities, a Java package or .NET assembly. The relationships between those packages or assemblies are comparable in relation to the version semantics. However, these schema come with a manual configuration of a version and therefore, there is nothing to bar stating incompatible changes as compatible.

3 A VERSION-AWARE WEB SERVICES SERVER OVERVIEW

The server-side system needs to enforce non-change of version unless the consumer explicitly chooses to change. This implies that a service-based application will not unexpectedly break due to changes in the service implementation. Apart from this basic motivation, Web service implementation versioning yields benefits for improving manageability, reliability, flexibility and visibility of the versioned services. The major features are summarized as follows:

- Modification events are enforced to be reflected from implementation to services. Thus, service-level version management become more reliable.
- Inter-service compatibility for service orchestration is maintainable in relation to version dependencies (Andrews, et. al., 2003).
- SCM (Software Configuration Management) systems on the client side can manage dependencies of their applications and remotely implemented Web services in relation to the versions.
- Developers are allowed to implement both backward compatible and incompatible changes to the same endpoint. The URL of the service endpoint maintains its representation as of service marketing time.
- Version management details such as a history and the version graph can be made available to client applications.

At the time of writing, many extensions to the Web service architecture e.g. WS-Security and WS-Reliability are emerging. The SOAP header and its handler provide a flexible way of adding pre- and/or post-message processing functionality to Web services. A version-aware Web service server utilizes this mechanism by simply incorporating the version information into the SOAP header. The considered architecture here is the component based deployment of service implementations. The constituents are summarized below.

3.1 Change Detection

For the version-aware Web service server to monitor the service implementation for changes, some changes in the component repository and the configuration of the components' container should be transparent to the version management systems. Therefore, version management systems and application servers need to collaborate closely with each other. Resulting benefits: (1) separating roles of resource management from container specific tasks, (2) making version management systems independent of a particular application server as well as other systems/servers, and (3) enabling version management of fine-grained software entities such as packages and classes. Application servers should detect changes that have occurred to the relevant component by either push or pull semantics, otherwise the use of modified components should be denied. The implementation of the change detection functionality depends on the SCM systems and will not be discussed further in this paper.

3.2 Service Immutability

In order to ensure immutability of service implementations in the component repository, the version-enabled Web service server uses a digest - a fingerprint of the service implementation. The hashing functions, MD5 (Rivest, 1992) and SHA-1 (Eastlake, 2001), are two major candidates to generate such digests. If a digest is sent with a SOAP message, the service implementation of requested version is verified against this. A digest mismatch will always be a result if the implementation changes. A digest is generated and stored for each version of a service. The SCM system needs to know the managed (and potentially unmanaged) software entities.

A version-aware Web service server provides a built-in function for service requestors to retrieve and preserve the digest of a particular service version. Service requestors can then verify non-changes to the service implementation by matching the local and remote copies of the digests. To avoid modification to the functions of digest generation, this function needs to be kept built-in.

3.3 Version Parser

Web Service Implementation Versioning is based on the idea that multiple versions of service implementations are available simultaneously. Therefore, the version-enabled Web service server should be able to parse a version out of a service request message. Specifically, this is done by a Version Handler.

The version number is included in a header of SOAP messages. When the Version Handler detects the existence of a version header named Service Version Header (SVH) in a SOAP request message, it verifies the existence of the requested version number against the version repository.

Unlike the case of attaching a version to the URL, the version-aware Web service server can differentiate errors due to incorrect endpoint and the non-existence of a version number. Moreover, the SVH is also extensible to support an intentional versioning that takes account of user preferences specified in the message. This is described in the next section.

3.4 Version Selection

As seen by current practices, the version format may not be simple especially where it is concerned with a wide variety of applications or industry specific variants. If the new version is not an evolved component of its baseline, this is not even applicable to a version graph (Conradi, 1998). However, we take into account that the client's concern is not the format itself but the derived impact to the client application from changes - compatibility.

While the format may vary among applications, the interested version group (Bendix, 1996; Gergic, 2003), aka version sets (Conradi, 1998) which is a versioned item, on the level of product unit is often as simple as described by major and minor numbers in this situation. Therefore, the selection of software entities can be performed by two means: (1) providing the complete version identifier or (2) specifying a baseline (partial version identifier) which leaves the selection of revision and build numbers, for instance, to be derived from the client preferences and the up-to-date version graph. Taking the benefit of XML schema languages, the SVH can constrain acceptable options of such client preferences. On the other hand, if a version is totally missing, then the choice will be either the latest up-to-date version or a SOAP fault.

4 ARCHTTECTURE

A version-aware Web service system provides a way to decouple the versioned service implementation from its interface description. Typically, changes to service implementation occur more frequently than to its interface. However, if an endpoint URL is to indicate a version identifier, the number of endpoints increases by each release of an existing Web service. More precisely, regardless of non-change to the abstract interface definition, the concrete implementation definition may need to specify every possible version formulation in WSDL. Here, the version formulation also implies the possible use of change-based systems for SCM. In such a case, the possible versions of a service increase exponentially by the number of versions of the underlying software entities behind WSDL. This causes a number of issues: (1) confuses clients with a wide range of URLs, (2) may multiply dependencies of WSDL files (3) complicates management of service endpoints, (4) makes URL non-abstract (5) confuse s the faults that arise from version mismatch and incorrect endpoints.

As noted earlier, the solution to these issues we propose here is to separate a version identifier from the endpoint URL and the namespace exposed by WSDL. Thus, the clients configure their application with a version-free endpoint and namespace. This reduces the need of re-configuration or re-deployment of client applications when they want to bind to new versions, which also help the URL to be kept meaningful to humans. On the other hand, this requires a way for the clients to retrieve the same version of the WSDL document from the same endpoint as well as for clients to preserve a version identifier to which their applications are bound.

The following subsections describe procedures for the service configuration, deployment and invocation that occur for an installation of the version-aware Web service server.

4.1 Implementation Deployment

The deployment of software entities go through a SCM system. While not violating the container functionality such as security and transactions, actual service implementations are stored in a component repository in the SCM. This is necessary because the maintenance, storage and retrieval of software entities are more efficient by using SCM Figure 1 illustrates the procedure of implementation deployment. The detailed steps follow the figure.

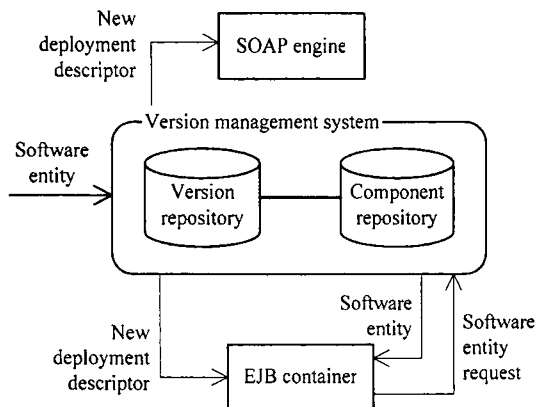

Figure 1: Component deployment procedure

1. A software entity is added or checked-in to the component repository.
2. The version management system generates a new version for related services and an associated digest.
3. The version dependency is updated in response to the semantics of the new version.
4. The version management system creates a new component name derived from the one in the base deployment descriptor. Then, the new deployment descriptor is generated with the new component name for deployment.
5. The version management system notifies the application server that the new version of the existing components is available (or the application server detects it by itself).
6. The application server configures itself with the new deployment descriptor. The new deployment does not affect other components in use. The detailed procedure of deployment follows the instruction from application servers so that the architecture does not defeat the benefits of current deployment automation.
7. The version management system notifies the SOAP engine of the new deployment descriptor for new service configuration.
8. The details of the new version are added to the UDDI registry if applicable.
9. If the service requestors subscribes for the notification of changes in the UDDI registry for the relevant services, the UDDI communicates with the subscribers for new versions available.

4.2 Client Application Configuration

Client applications are configured in much the same way as for non-version-aware Web services. The only difference is the existence of the SVH in messages that contains version information. The procedure is simply:

1. A service requestor obtains a WSDL file.
2. The latest version of WSDL file is returned from the server.
3. Client applications are implemented in a way that sends a SOAP message containing version preferences in the SVH.

4.3 Service Invocation

An additional handler in the SOAP engine means the selection and verification of the correct version is performed before the invocation of the service implementation. The selection procedure utilizes both a namespace of the requested services in the SOAP message and the SVH. More specifically, whereas the namespace in the message merely specifies a version group, the SVH carries a specific version local to the group.

For the service invocation, the version identifier means a unique identity for a collection of software entities that implement a service. When incorporating the complete version identifier in the request message, it is guaranteed to be dispatched to the desired version of software entities. The Figure 2 illustrates the interactions of clients, SOAP engine, components container in the application server and version management system. For brevity, other necessary SOAP handlers are omitted from the figure. The detailed procedure is as follows:

1. A SOAP message arrives to an intermediary or the ultimate destination, either of which is capable of handling the SVH.
2. When the SVH is detected, it searches for a matching version identifier from the version repository. If it is found, then the Version Handler replaces the namespace of the requested service in the message with a version identifier. If it cannot be located or multiple version identifiers are located, then the request is rejected with an appropriate SOAP fault.
3. If the SVH is missing, the service call is either rejected or dispatched to the most up-to-date version of the requested service.
4. The return message carries the original namespace that is identical to the one in the

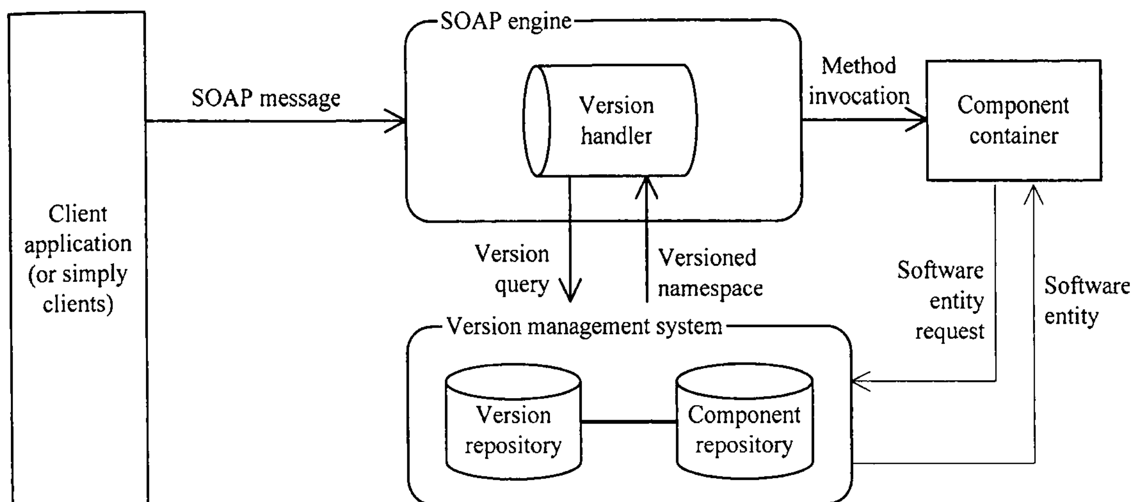

Figure 2: Service invocation procedure

request message, and a version identifier of the service executed.

Finally, a version identifier and a digest are the required ingredients for the verification of service immutability. The verification of implementation immutability occurs as a consequence of incorporating a digest into the SVH; otherwise it would not be performed. This is due to the performance concern that the verification puts an overhead to a service invocation. As well as allowing clients to verify the service implementation immutability, on the server side, this provides a basic protection against the violating interaction that attempt to alter the existing software entities without creating a new version.

5 DISCUSSION AND FUTURE WORK

The main drawback of this approach is the maintenance complexity of different versions of components deployed on the same application server. As the version management is not the primary role of a component container, it is not attractive to implement version management functionality into an application server.

Moreover, in the case of dynamic binding, the newest service implementation is still only a choice for a given industry-specified interface

Additionally, a slight performance latency can be expected due to the process for the version verification. However, this would not be a major issue since such a comparison cost is lightened with the help of caching.

6 CONCLUSION

The versioning issue is a long examined (and potentially never ending) issue in software development and management. The collaboration of SCM systems and SOAP engine simplifies tasks of both client applications and service developers by providing automated support for version selection. This also enables the SOAP engine to spot the exact place of faults and return a more informative guide to client applications. This paper has described the fundamental requirements and methods of versioned service deployment, version configuration, version look-up, version identification and change detection, which is sufficiently feasible to implement with the help of currently available standards and software products. The future work will target the evaluation of architectural resistance to malicious modification of version identification and service implementation.

REFERENCES

Akamai Technologies, Computer Associates International, Fujitsu Laboratories of Europe, Globus, Hewlett-Packard, IBM, SAP, AG, Sonic Software, TIBCO Software, 2004, *Web Services Notification*, , http://www-106.ibm.com/developerworks/library/specification/ws-notification.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I. & Weerawarana, S. 2003, *Specification: Business Process Execution Language for Web Services Version 1.1*, http://www-128.ibm.com/developerworks/library/ws-bpel/.

Anderson, R. 2000, *The End of DLL Hell*, Microsoft Cooperation, http://msdn.microsoft.com/library/en-us/dnsetup/html/dlldanger1.asp.

Bendix, L. 1996 'Fully Supported Recursive Workspaces ', in, *Proceedings of the SCM-6 Workshop on System Configuration Management* Springer-Verlag, pp. 256-261

Box, D., Cabrera, L.F., Critchley, C., Curbera, F., Ferguson, D., Graham, S., Hull, D., Kakivaya, G., Lewis, A., Lovering, B., Mihic, M., Niblett, P., Orchard, D., Saiyed, J., Samdarshi, S., Schlimmer, J., Sedukhin, I., Shewchuk, J., Smith, B., Weerawarana, S. & Wortendyke, D. 2004, *Web Services Eventing (WS-Eventing)*, IBM et.al, http://www-106.ibm.com/developerworks/webservices/library/specification/ws-eventing/.

Brada, P. 2000, *SOFA Component Revision Identification*, Department of Software Engineering, Charles University, Prague.

Brown, K. & Ellis, M. 2004, *Best practices for Web services versioning*, IBM, http://www-106.ibm.com/developerworks/webservices/library/ws-version/.

Butek, R. 2004, *Make minor backward-compatible changes to your Web services*, IBM, http://www-106.ibm.com/developerworks/webservices/library/ws-backward.html.

Conradi, R. & Westfechtel, B. 1998 'Version models for software configuration management ', ACM Comput. Surv. , vol. 30 no. 2 pp. 232-282

Eastlake, D. & Jones, P. 2001, *US Secure Hash Algorithm 1 (SHA1)*, RFC Editor.

Eisenbach, S., Jurisic, V. & Sadler, C. 2002, 'Feeling the way through DLL Hell', In *The First Workshop on USE '02*, http://joint.org/use2002/, Málaga, Spain.

Gergic, J. 2003, 'Towards a versioning model for component-based software assembly', In *Proceedings of ICSM 2003*, pp. 138-147.

Gosling, J., Joy, B., Steele, G. & Bracha, G. 2000, *The Java Language Specification*, viewed 19 Jan 2005 http://java.sun.com/docs/books/jls/.

IBM 2003, *Web services for J2EE Specification*, http://jcp.org/aboutJava/communityprocess/final/jsr921

Irani, R. 2001, *Versioning of Web Services - Solving the Problem of Maintenance*, http://www.webservicesarchitect.com/content/articles/irani04.asp.

Kalali, B., Alencar, P.S.C. & Cowan, D.D. 2003a 'NSPF: Designing a Notification Service Provider Framework for Web Services ', in, *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems* Springer-Verlag, pp. 73-90

Kalali, B., Alencar, P. & Cowan, D. 2003b 'A service-oriented monitoring registry ', in, *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research* IBM Press, Toronto, Ontario, Canada pp. 107-121

Microsoft 2005, *.NET* Framework, http://msdn.microsoft.com/netframework/.

OASIS 2004, *UDDI Version 3.0.2*, http://uddi.org/pubs/uddi-v3.0.2-20041019.htm.

Plasil, F., Balek, D. & Janecek, R. 1998, 'SOFA/DCUP: Architecture for Component Trading and Dynamic Updating', In *Proceedings of ICCDS '98*, IEEE CS Press, Annapolis, Maryland, USA, pp. 43-52.

Rivest, R. 1992, The MD5 Message-Digest Algorithm, RFC Editor.

Sun Microsystems 2003, *Java Product Versioning Specification*, http://java.sun.com/j2se/1.4.2/docs/guide/versioning/spec/versioning2.html.

W3C 1999, *Namespaces in XML*, http://www.w3.org/TR/REC-xml-names/.

W3C 2003, *Versioning XML Languages*, http://www.w3.org/2001/tag/doc/versioning.

W3C 2004a, *SOAP specifications*, http://www.w3.org/TR/soap/.

W3C 2005, *Web Services Description Working Group*, http://www.w3.org/2002/ws/desc/.

W3C 2004b, *Web Service Description Language (WSDL)*, http://www.w3.org/TR/wsdl.