Versioning of Web Service Interfaces

by

Anamika Agarwal

Bachelor of Technology, Civil Engineering (2002)
Indian Institute of Technology, Bombay

Submitted to the Department of Civil and Environmental Engineering
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

at the

MASSACHUSETTES INSTITUTE OF TECHNOLOGY

June 2004

Author ____________________
Anamika Agarwal
Department of Civil and Environmental Engineering
June 29th, 2004

Certified by [illegible]
John R. Williams
Associate Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by [illegible]
Heidi M. Nepf
Chairman, Committee for Graduate Students

Versioning of Web Service Interfaces

by

Anamika Agarwal

Submitted to the Department of Civil and Environmental Engineering
on June 29th, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis investigates the problem of "design for change" in the context of Web Service based information systems. It describes the current status of architecting Web Services, an implementation of the Service Oriented Architecture. It also discusses the availability and support for higher level specifications, such as security, that leverage the baseline standards like SOAP (Simple Object Access Protocol), WSDL (Web Service Description Language), and UDDI (Universal Discovery, Description and Integration). This thesis examines versioning as an additional specification. It discusses the Web Service enabled programming environment by addressing the various causes of change. Two levels of versioning are identified to deal with this problem - Schema level and Interface level versioning. For Schema level versioning, the extensibility models and their versioning support are discussed. Other techniques of using distinct namespaces and custom versioning tags are presented by means of samples. The second level of versioning, the Interface Level Versioning, discusses implementation of versioning logic using the current standards of WSDL 1.1. In addition, the ongoing recommendations and efforts in this field are stated.

Thesis Supervisor: John R Williams

Title: Associate Professor

Acknowledgement

First of all, I would like to thank Prof. John R. Williams for his constant support academically and otherwise. His words of encouragement and assistance made me explore a really new field. I am grateful to him for the freedom of thought he had given while pursuing the thesis.

Special thanks to Abel without whom this thesis was not doable. He was a guide as well as a friend. His innovative ideas and problem solving skills were really inspiring and kept me on track for the thesis. He has a major role to play in shaping my experience in this field.

Sakda, another senior member of the IESL group, helped me with his knowledge and great collection of books. The long discussions with him really helped me to come up with some good insights about the topic.

I am also thankful to Sivaram and Hari, two graduate seniors in the department, for all their help and guidance over the years. My fellow graduates, Andy and Deepak, were a great company to work with. The presence of other members of the group like Hai, Stephen, Scott, George, Sudarshan, Ching-Huei and Ying-Jui Chen kept me motivated to work and I am thankful to them for the same. I would also like to thank Joanie, Cynthia and Jeanette for helping me out with all administrative issues.

My friends Vijay, Yamini, Smeet, Shobhit, Vikrant, Sushil, Manu, Saikat and Udai made me feel home away from home during my stay at MIT. I would like to thank my parents and relatives for all their love, support and encouragement all over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"No program is an island" – Don Box

Don Box's quote [5] is realized in the distributed software programming world where every program fundamentally needs the support of other programs. However, with time the individual programs change independently breaking the existing compatibility. Designing for change is a critical problem which has no satisfactory solution at present. This thesis examines "design for change" in the context of Web based distributed systems. Web based distributed systems are usually organized using a layered model as shown in *Figure 1*. Layering of an application allows decomposition of into component tiers. This thesis illustrates the layered application architecture and discusses the relevance of versioning in the software lifecycle of a Web based distributed system.[1]

1.1 Background on Application Architecture

Most software application architectures are layered. A layered architecture categorizes the application into logical units, or tiers known as application components which are standalone entities and communicate with the adjacent tiers. *Figure 1* shows the typical

[1] The Web based distributed system discussed in the thesis is based on Web Services. Web Service is the latest approach to the problem of developing software that is scalable, extensible and maintainable. It builds on the concept of OOP, interface based programming, COM+, and CORBA. This thesis assumes the reader is familiar with these techniques [9, 10, 11, and 13].

tiers in application architecture: client tier, presentation tier, business logic tier, integration tier and data tier. The client tier represents how the user interacts with the system. The presentation tier is devoted to the dynamic creation of display code (HTML) that is sent to the client tier. The business logic layer contains the implementation and processing of the actual operation. The integration tier encapsulates the functionality required for an application to communicate with the data tier or other business logic tiers. The data tier deals with information repositories like data in databases and file systems [3]. However, this architecture can be extended to n-tiers by identifying other logical units as well.

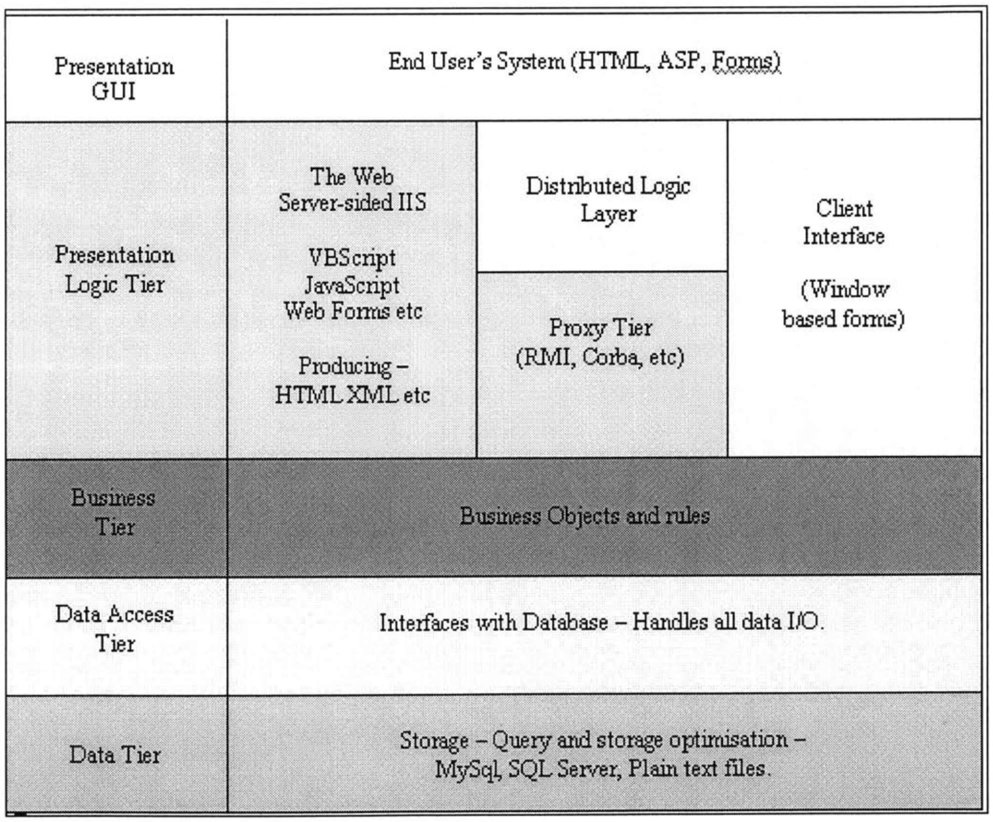

Presentation GUI
End User's System (HTML, ASP, Forms)
Presentation Logic Tier
The Web Server-sided IIS
VBScript JavaScript Web Forms etc
Producing – HTML XML etc
Distributed Logic Layer
Proxy Tier (RMI, Corba, etc)
Client Interface
(Window based forms)
Business Tier
Business Objects and rules
Data Access Tier
Interfaces with Database – Handles all data I/O.
Data Tier
Storage – Query and storage optimisation – MySql, SQL Server, Plain text files.

Figure 1 - Tiers of an Application [4]

A tiered architecture hides the implementation details within the tier and forces all interaction to occur via explicit interfaces as shown in *Figure 2*. Hence, these logically and functionally separated tiers can be designed independently. Changes within a tier are encapsulated and do not affect other tiers unless the changes are reflected in the exposed interface.

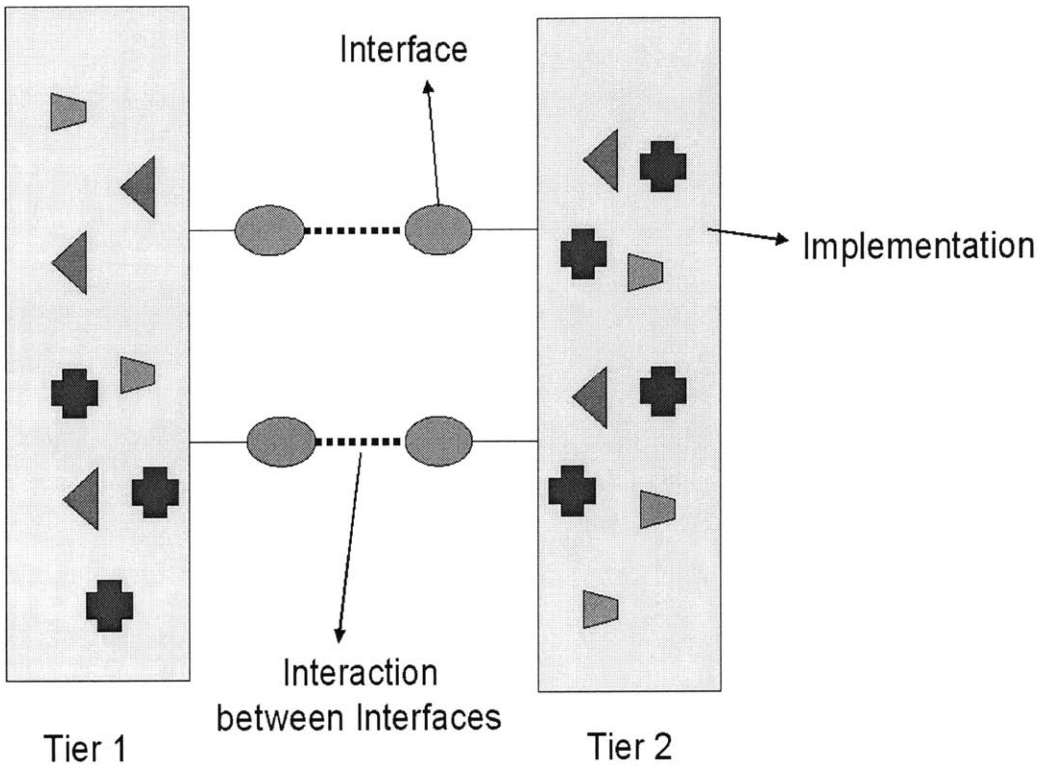


Figure 2 - Interaction of tiers via Interfaces

These tiers are deployed in different network configurations, leading to different application architectures. They come in two broad categories:

- **Monolithic Applications** in which all the tiers are placed within the same computer system where they are virtually or conceptually separate from each other for example, applications running on mainframes.

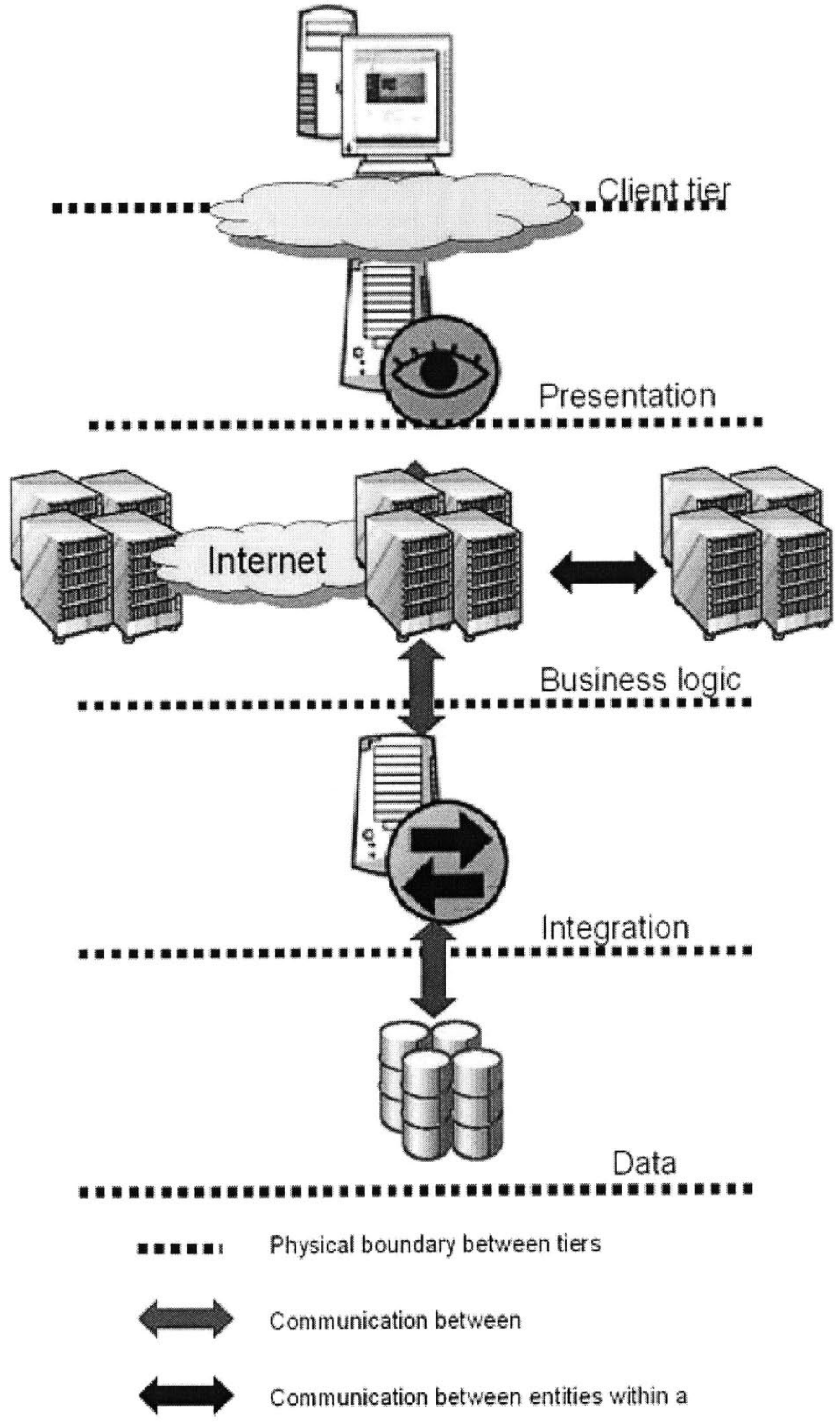


Figure 3 - Workflow and Interaction between the tiers via Interfaces

- **Distributed Applications** in which the tiers are spread over several computer systems which are connected. *Figure 3* shows an example of a distributed system in which the tier components are located at different physical locations and interact with each other via standard Application Programming Interfaces (API's) exposed by the standalone functional units. Separating the responsibilities of an application into multiple tiers makes an application scalable and its components reusable.

1.2 What is versioning?

Applications are changed over time in response to the changing conditions around them and as a result, the exposed service API's also change. The kinds of changes expected in application development are characterized into three types:

- Addition of new functionality to an existing application
- Modification of existing operations in an application
- Bug Fixing

Good quality control procedures require managing, tracking and accessing the history of the various phases the application has undergone. This is achieved by versioning. Versioning tracks how the software has changed over time. Traditional versioning involved wholesale replacement of the software with no guarantee of compatibility between versions. This replacement approach didn't have a structure to control and manage the change, and hence any software system of any useful size eventually degenerated into a chaos. Gradually, compatibility (harmonious existence of the various components) became a key concern between the various versions of an application. In broad terms, the approaches to versioning fall into a number of classes ranging from "none" to a "big bang" [2]:

- *None*: No distinction is made between versions of the language. Applications are expected to cope with any version they encounter.
- *Compatible*: Designers are expected to limit changes to those that are either backward or forward compatible or both. *Backward compatible* changes allow applications to behave properly if they receive an "older" version of the application request, language, or an instance document. *Forward compatible* changes allow existing applications to behave properly if they receive a "newer" version of the language, application request or an instance document.
- *Big Bang*: Applications are expected to abort if they see an unexpected version.

Versioning strategies at the extreme ends (none or big bang) might be appropriate for autonomous or monolithic systems but in the world of distributed systems they pose problems. For instance, compatibility issues arise when a new version of a component interface interacts with an interface of an older version of another component or vice versa.

1.3 Research Motivation

This thesis makes the argument that the versioning support should be considered at the design phase of the Web based distributed systems because it is an important factor in the reduction of costs, resources and complexity. The decisions made during the design period of the software development cycle determine 80% of the product's costs as shown in *Figure 3*. The absence of any versioning support in the architecture results in increase of complexity and cost. A lot of resources in terms of cost and efforts are spent in interfacing, sustaining and running the existing capabilities instead of providing any sort of value creation to such systems [1]. So, it is required to have an inbuilt support at the design phase to track the version changes which is lacking in the current architecture of Web based distributed systems.

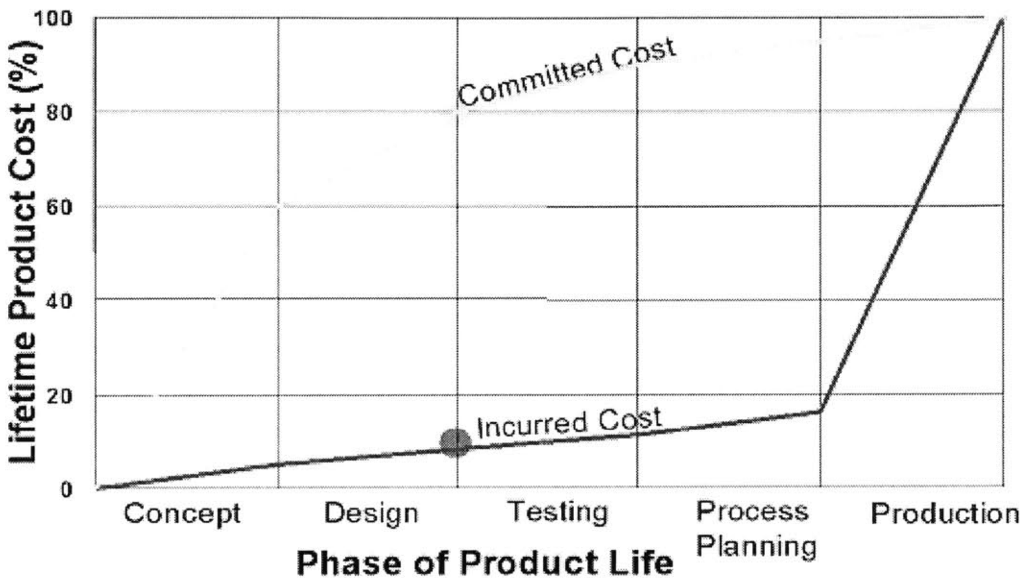


Figure 4 - Software life Cycle [1]

This trend of incurred cost (as shown in *Figure 4*) is observed in the component technologies which have evolved till date say RPC, DCOM, CORBA, and RMI [13]. XML Web Services - the latest paradigm in distributed computing, have evolved from the past lessons learnt by earlier distributed technologies and has helped alleviate the problems like interoperability, loose coupling and reusability. The baseline standards for integrating two heterogeneous applications exist today but there are a number of issues to be addressed which are discussed in Chapter 2.

This thesis specifically addresses the issue of versioning of XML Web Services. The technology itself is relatively new, and so very little work has been done on the versioning aspect of it. However, there are working groups within W3C and companies like IBM, Microsoft, BEA etc., who realize the promise of this technology, and are collaborating to develop standards and specifications.

1.4 Problem Statement

This thesis investigates the versioning strategies that could be implemented in the existing Web services model. These strategies involve adding some versioning logic at the Web service interface, which is the point of interaction with the outside world.

Interfaces are connectors through which applications or components interact and are exposed in the form of service API's (Application Programming Interface). The component software environments are largely based on distributed platforms and, due to natural software evolution, typically contain multiple versions of components that need to communicate with other components. Correctly handling API versioning has been one of the most difficult issues faced by developers of distributed systems. New evolving component API's need to be compatible with the API's of the existing components within an application. The evolution should be such that the existing API's are able to consume the versioned API in a seamless manner without any changes and recompilation at their end. All existing applications need to retain their functionality as the individual pieces evolve.

In a distributed environment, developers have to deal with distributed set of components that aren't necessarily under their control. The current XML Web Services architecture does not provide any support for versioning of services. A robust versioning strategy is needed to support multiple versions of Web Services in such a way that upgrades and improvements are made to a Web Service; while continuously supporting previously released versions. In this thesis, the above discussed problem is outlined followed by some template solutions and strategies, along with the discussion of architecture for addressing the problem through the application of patterns and best practices. It investigates the various issues of API versioning and uses .NET based techniques for implementation of the proposed solutions.

1.5 Research Organization

In accordance with research objectives, scope, and approach already discussed, the organization of the thesis can be described as follows:

- **Chapter 2:** This chapter summarizes the conceptual models of Service Oriented Architecture and discusses the present Web Services Stack. This is followed by the reasons for the Web Services hype and the issues which are given a blind eye in the current specifications.
- **Chapter 3:** This chapter discusses the problem of maintenance and versioning from a business perspective that an organization would face when it exposes its applications as a collection of Web Services for Enterprise Application Integration and Business to Business Integration by considering an example business scenario.
- **Chapter 4:** This chapter focuses on versioning and defines standard definitions to maintain a consistency with the usage of certain key words. It talks about the various scenarios which lead to change in services and identifies the versioning schemes/ levels in the lifecycle of Web Services.
- **Chapter 5:** This chapter discusses the problem of versioning from the XML schema perspective. Techniques to implement flexibility and loose coupling within the schema by using certain XML tags are discussed which is followed by the introduction of extensibility and open content models to extend the schema. The subtle difference between versioning and extensibility is highlighted. The chapter concludes by proposing some best practices for schema versioning.

- **Chapter 6:** This chapter illustrates the versioning at the Interface level. For the same, the WSDL standard specifications are analyzed. It presents the anatomy of WSDL 1.1 and WSDL 2.0. This is followed by a sample .NET Web Service which is considered as the base example to illustrate the possible changes within the service and techniques to implement those changes with some versioning logic.
- **Chapter 7:** The thesis concludes by presenting recommendations for versioning Web Services and discusses ongoing efforts and future direction in this field.

Chapter 2

Evaluation of the Current Web Service Architecture

The object oriented model is insufficient to address the issue of integration of multiple programs in a distributed environment. The attempt of using the **object** metaphor designed for the in memory case (object oriented model) - the local case, and stretch in the distributed world did not work. This is illustrated by the example of COM objects which worked extremely well within the same address space but ran into real production issues when tried to implement the same object technology and stretch it to a distributed level known as DCOM. This led to the search for another level of abstraction – service oriented model.

2.1 The Service Metaphor

The metaphor of **service** is considered in the service oriented model which treats program as *well-defined, self contained* software units *independent* of the *context* and *state* of other services. It is considered as a promising solution to bring together programs of all types running on different platforms and even programs from the local box. The fundamental premise is defined on a concept of **boundary** which means separation of service interactions from service implementation [5]. It states that systems are modeled in such a way that **there exists a very strict boundary between any two programs.** These boundaries can occur in various forms like geographical distances, trust authorities, distinct execution environment; and are to be respected by other programs. In the service orientation model, each program shares a contract with other programs which describes the behavior of the given program in the form of messages (discussed later). These small programs with explicit boundaries and contracts could be used to build a system of distributed programs.

Before discussing the components of the service oriented model it is important to understand what a message is and how does it differ from any other mode of communication. A **message** is defined as a collection of data fields sent or received together between software applications. It is composed of two parts- header and payload. The header stores control information about the message and the payload contains the actual content of message. Messaging use these messages to communicate with other systems to perform some kind of function. Such communications are referred as message-oriented because it would send and receive messages to perform the operation.

2.2 The Building Blocks

The **building blocks** of a service oriented model are **clients, services and systems.** *Figure 4* shows how the service and client interact within a system [5].

- A service is a program to which one could communicate using messages.
- A client is another program that uses the service by sending messages.
- A system is analogous to applications in the object oriented world. In the world of services, a system is an environment where a lot of programs have to coexist with other programs, for example real world business processes. Coexistence means not just building the system piece by piece but also maintaining it as it evolves. New pieces are created and old pieces are discarded over a period of time.

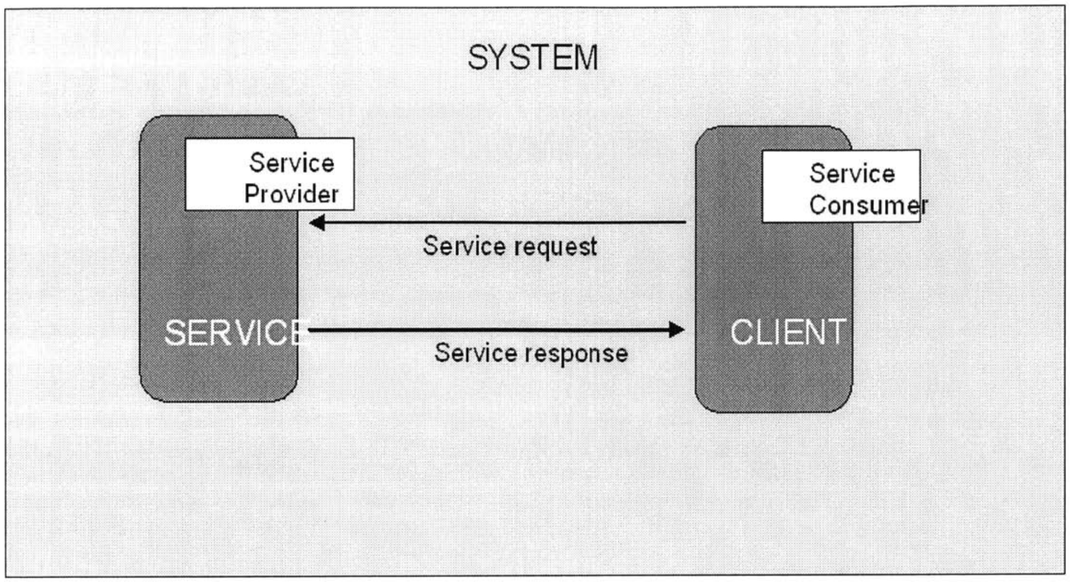


Figure 5 - System definition in Service Oriented Architecture

Contrasting this service oriented architecture with the object oriented architecture, one would conclude that both were built to cater different programming environments and hence there are some key differences in the two architectures:

- Object Orientation assumes a homogeneous platform and execution environment in contrast to Service Orientation which assumes a heterogeneous platform and execution environment.
- Object Orientation share types and not schemas whereas Service Orientation shares schemas and not types.
- Object Orientation assumes cheap, transparent communication in contrast to Service Orientation which assumes a variable cost and explicit communication.
- In Object Orientation objects are linked and the lifetime of the objects is maintained by the infrastructure whereas in Service Orientation services are autonomous and the security and failure isolation is at a service level instead of a system level.

2.2.1 Tenets of Service Oriented Architecture (SOA)

The Service Oriented Architecture stands on some core principles. There are four tenets which uniquely define this architecture [5]:

- **Boundaries are explicit**. Explicit boundaries allow the formal expression of implementation independent interactions. These interactions do not depend on the platform, middleware, or coding language choices used to implement the various services. As a programmer one should define the limits of a program and explicitly expose the service based functionality. Unlike object oriented model, the service oriented model keeps the surface area of a service definition as small as possible. Methods within a service are only accessible via the channel of standard messages. External services cannot touch the class definitions and the methods of a service directly.
- **Services are autonomous.** In the service oriented world, every program has to ensure its functionality even in the absence of other cooperative programs. Being a part of such a system in which one doesn't have control over the other, every service has more responsibilities. There is an expectation of reliability and availability between programs. If a service is published then it is assumed that it would be available over time and the availability would be in a predictable way which would be mentioned in the contract. Independent deployment, versioning and security are the primary areas of concern for these services. Security infrastructure has to be solid as packets can travel freely in such a system between the programs. Services are deployed independently and the issue of how the evolution of one service affects the rest of the system is of prime concern. In the service oriented system there is no centralized unit or repository which stores all the interfaces and hence there is no way for the propagation of the change to other connected services.

- **Share schema and not class.** Integration is based on message formats and exchange patterns, not classes and objects. Schema is all about structural representation at the wire level and contract tells about behavior. Contract describes a set of message exchanges between services in terms of behavioral interactions. In fact those interactions are messages which are schematized.
- **Policy based compatibility.** Service autonomy means types and service will be versioned independently. So, one cannot assume a central common type definition. This would mean using the same XML schema globally for a given set of data which is not possible. The schema definition and validation rules at one end are distinct from the validation rules and schema used at other end to verify a given data structure. The validation rules might be tighter or looser, the set of default assumptions might be different and the way it is mapped to technologies might be different at the two ends. Policy associates stable global names based on URI's with ideas, concepts. The compatibility between any two services is not done by matching interface definitions rather by identifying what capabilities the service offers and what are the requirements at the other end which is done by stable global names.

2.3 The present Web Service Stack

Web Service is yet another implementation of the service oriented architecture. XML Web Services is a new standardized way of integrating disparate systems and applications connected through an Internet protocol (IP) backbone [6]. The standard relies on XML as a language for tagging data; Simple Object Access Protocol (SOAP) for transferring that data; Web Services Description Language (WSDL) for describing the services available; and Universal Description, Discovery, and Integration (UDDI) for listing what services are available. Following table lists the basic Web Service Stack.

- **Transport layer**: The transport layer is the foundation of the Web services stack. The Internet protocols that can be supported by this stack are HTTP and, to a lesser extent, SMTP (for electronic mail) and FTP (for file transfer).
- **XML-based messaging layer**: SOAP is the messaging protocol for XML. It is built upon the lower layer, the transport, so that messages could travel along with the transport protocol. All SOAP messages support *publish, bind,* and *find* operations in the Web services architecture. SOAP comprises of three parts: an envelope to describe the content of a message, a set of encoding rules, and a mechanism for providing remote procedure calls (RPCs) and responses.

Tool	***Layer***
Static-> UDDI	Service Discovery
Direct -> UDDI	Service Publication
WSDL	Service Description
	Service Implementation
	Service interface
	Secure Messaging
SOAP	XML based Messaging
HTTP, FTP, SMTP, MQ, RMI over IIOP	Transport

Table 1 - the basic Web Service Stack

- **Service description layer**: Service description provides the means of invoking Web services. WSDL is the basic standard for defining, in XML format, the implementations and interfaces of the service. This means that WSDL divides a service description into two parts: **service implementation** and **service interface**. A service interface is created before the implementation of WSDL.

- **Service discovery layer**: Service discovery relies on service publication. If a Web service is not or cannot be published, it cannot be found or discovered. The service client can make the service description available to an application at runtime. The service client, for example, can retrieve a WSDL document from a local file obtained through a direct publish. This action is known as *static discovery*. The service can also be discovered at design or run time using either a local WSDL registry or a public or private UDDI registry.

These layers within the web service stack serve as critical foundation elements for the implementation of the service oriented architecture. However, there are real production and deployment issues associated with this proposed stack. Additional features and functionalities have to be considered to stretch it to a commercial and business level from a prototype model.

2.4 The Web Service Overstatement

Web service has also undergone the *technology adoption hype cycle* as any other technology [7]. It has been exaggerated since its emergence. The key selling points of Web Service- centric technologies has been their simplicity and ease of implementation. It addresses the core capability of messaging within service-oriented architecture in a consistent, ubiquitous, internet centric manner. This hype is centered into two broad areas [6].

One category of proponents place emphasis on the capability of lower level Web Services API's but ignores or understates the lack of consistent higher level specifications like versioning, security and transactions. According to them, higher level API's would evolve within a short period of time.

As Web Services mature over time, opportunities and vendor disagreement would increase preventing the adoption of any particular vendor solution. So, every vendor is trying to have a foot in the door even in the absence of specifications and standards.

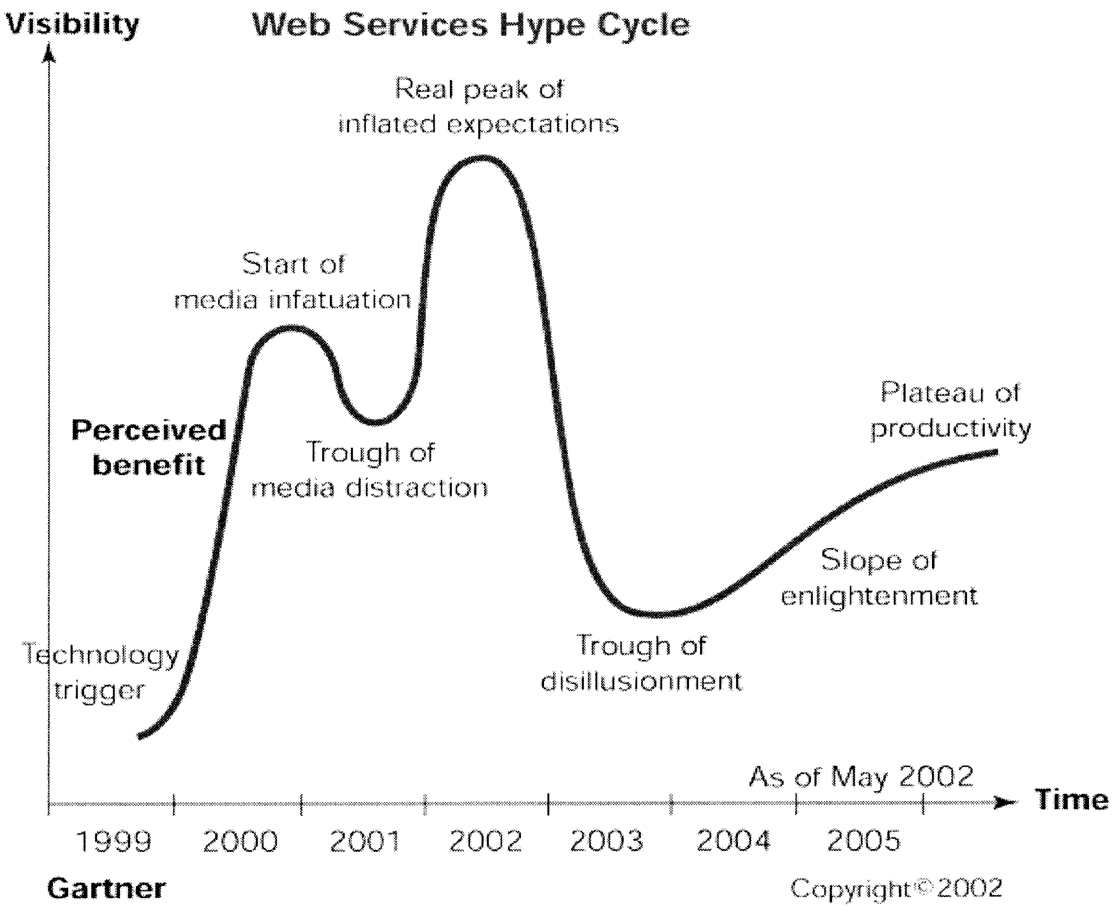


Figure 6 - Web Service Hype Cycle [6]

However, the potential of the Web service promise and its impact on e-business has made the hype overwhelming. Currently, Web Services are positioned at the massive peak of inflated expectations and is due for a serious slide into a trough of disillusionment. It has not been successful in real-world scenarios because the strong points were highly exaggerated and the weaker aspects were looked down upon and given a blind eye. As a result, higher expectations were created which were never realized. Developers and vendors tried to implement technology support upfront at a very early stage in the absence of specifications which led to proprietary solutions from each vendor. Before the technology matures in terms of specifications, implementation, stability, it is required that the underlying business model, best practices are created and stabilize over the period of time. However, this is not the case with this technology. The architectural evolution and the technology support are running going hand in hand. This leads to compatibility issues between the proprietary solutions defeating the idea and motive of this architecture. At the same time, it is important to realize that this simple infrastructure of Web Service cannot be extrapolated from prototype models to long term feasibility and practicability.

2.5 Pitfall of Web Services

Although Web Services does a great job at solving certain problems, they have certain limitations or bring along issues of their own. Although vendors agree widely on the core foundation elements of a web services software strategy, they are not taking the same approach towards higher-level APIs as they are added over time [8]. Some of these pitfalls are inherent to the technological foundations upon which Web Services are based, others are based on the specifications themselves and certain issues have not been taken care of till now. Hence, it is imperative to understand the issues, so that appropriate actions could be taken [8]. Following are to mention a few:

- **Availability** – Web Services are built on the same infrastructure as Web Sites. It does not guarantee the 100% availability of web sites. The spread of autonomous and independent services around the network fade the trust element between any two parties which is very important for the realization of such architecture. In addition, HTTP is not a reliable protocol, in that it does not guarantee delivery or a response. So, additional logic has to be inserted to confirm the successful execution of the request.

- **Matching Requirements** – Web services are envisioned as a "one size fits many customers" technology. However, individual customers wish to customize the general service as per their specialized requirements. So either, one has to maintain multiple versions/ instances of the service or other solutions have to be considered.
- **Immutable Interfaces-** One of the major advantages of this architecture is everything is loosely coupled. Changes at one end are not propagated automatically to the other end. So as the services evolve, it is extremely important that backward compatibility is maintained to run the existing client which is the focus of this thesis.
- **Performance Issues** – The performance will vary wildly from one request to another in such architecture. Every time to make a request, there are overheads of establishing and terminating the connection and travel of packets from one point to another. Another performance consideration is the conversion to and from XML during the communication process which is encoded within the SOAP envelope. Such a feature is definitely costly and slow compared to the transfer of data in binary format. XML takes more bytes to encode data than the equivalent binary representation. The overhead is bearable for small transactions, however it would become conspicuous for interactions involving huge amount of binary XML data transfer.
- **Lack of Standards-** Web Service is relatively a new technology and centers around incomplete and non finalized standards. Although the deficient features are implemented in some or the other way, they are unique to each vendor's

implementation. The guiding principle of Web Services is to create an open standardized system for remote program execution. Hence, the utilization of vendor-specific solutions should be avoided. The current Web Service specifications need to be complemented with specifications and standard models in the following areas:

1. *Security and Privacy*: The messages are sent over the internet in the form of packets which are viewable by anyone. Different levels of security are required for establishing a secure interaction between two parties which is lacking in the current standard. However, HTTPS is one way of safeguarding information, but it is not able to cater all the requirements.

2. *Non repudiation and Authentication*: Features like authentication (who is communicating with the service) and the proof for a particular communication took place are ignored by the standard specifications and are currently delegated to the Web Service container which kills homogeneity.

3. *Transaction*: Activities that are well suited to Web Services are very complex. They involve dozens of smaller actions that must be complete or rolled back. However, specifications for the proper coordination between the services, for example maintaining a particular sequence of activities, are not currently supported.

4. *Billing and Contracts*: The current architecture lacks a way for pricing contracts to be negotiated and maintained. Because the specifications don't have an agreed upon mechanism for handling these issues, vendors provide their own solutions. This can lead to problems when moving from one vendor's tools to another or getting two different vendor's tools to talk. This is the reason for most Web Services posted on public UDDI registries are still free to use.

5. *Provisioning*: Addition of valid user accounts is important to track the consumers of a given service and also to establish an element of trust between the two ends.

6. *Testing:* The Web Service architecture decouples the clients from the server. This makes the testing of a Web Service based solution difficult.

2.6 Current Initiatives

Efforts are ongoing to realize this architecture. A set of interested businesses like IBM, BEA, Hewlett-Packard, Intel, Microsoft and SAP have collaborated to form the **Web Services Interoperability** (WS-I) organization [6].

WS-I's goal is to facilitate the development and deployment of interoperable Web services across a variety of platforms, applications and programming languages. There are three phases pursued by this organization to achieve this goal.

- The first or "connection" phase involved laying out the core, baseline standards: the XML Schema, SOAP, WSDL and UDDI.
- The second phase focuses on "security and reliability". In this phase, critical Web services specifications like versioning and maintenance; XML Digital Signature, XML Encryption, HTTP-R, SAML and XACML are being worked upon.
- The third, or "enterprise," phase, will address provisioning, transactions, workflow and systems management.

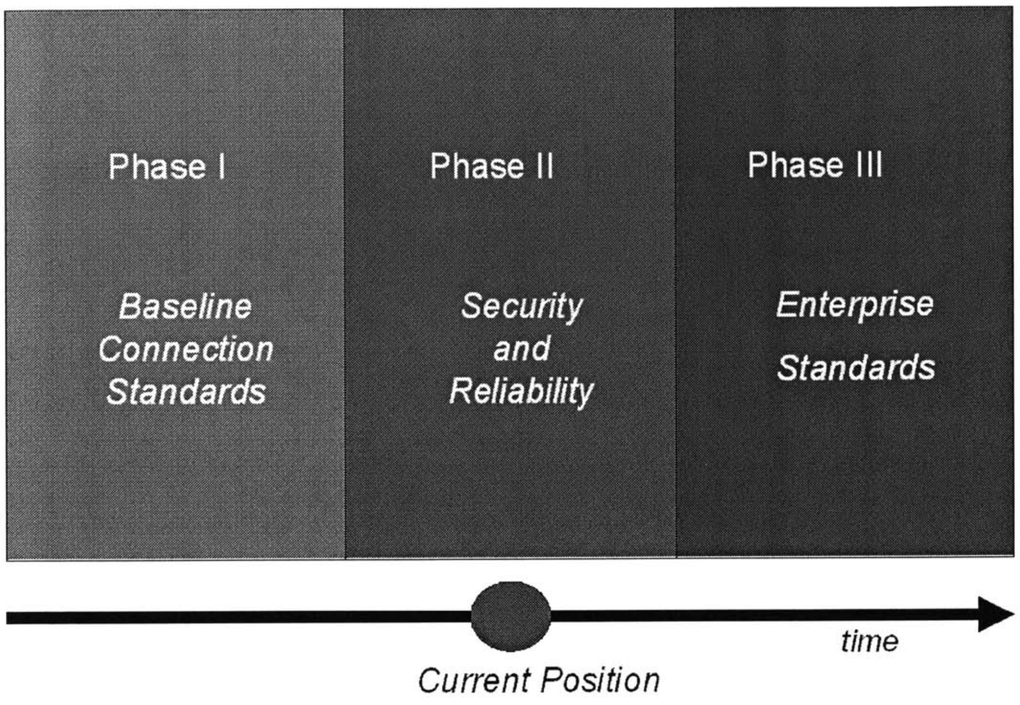


Figure 7 - Current position in the evolution of Web Service

Currently, specifications for the second phase are evolving. The standards are fresh and are changing very frequently. So, to leverage the promise of Web services it is more suitable to use it in pilot projects inside the firewall where interaction outside the corporate boundaries is required. After having discussed, the concerned issues at large we would now concentrate our focus on one such aspect which is required to establish reliability between services, which is versioning. Compatibility and existing interactions should sustain along with the evolution of the service.

Chapter 3

Versioning of Web Services – A Business Perspective

This chapter would concentrate on how this technology can be leveraged in a business scenario. One such hypothetical situation is discussed and the affects of the versioning problem are stated and the pros and cons of such an environment are presented.

3.1 Business Process Automation

The need of business process automation framework within and across enterprises had the following goals:

- Lower costs
- Streamlined and more efficient processes
- Increased operational views
- Gain real time views of information
- Faster time to market
- Ability to monitor and track process executions
- Ability to detect and manage exceptions
- Strengthen relationships within an organization and with partners and customers

Business process automation can happen within the subsystems (capital management, supply chain, customer relationship management) of an organization (EAI) or across distinct organizations (B2Bi) [30]. The process of creating an integrated infrastructure for seamless linking of disparate systems, applications, and data sources to share and exchange information within the corporate enterprise is known as *Enterprise Application Integration (EAI). Business to Business Integration (B2Bi)* is an extended form of EAI which is the secured coordination of information among businesses and their information systems. All these services act like black boxes which could receive and emit messages and the implementation details are hidden from each other. They are an extended version of the class objects but operate at a much larger scale i.e. at the enterprise level. The service would comprise of the business logic and data tier which is hidden from other services.

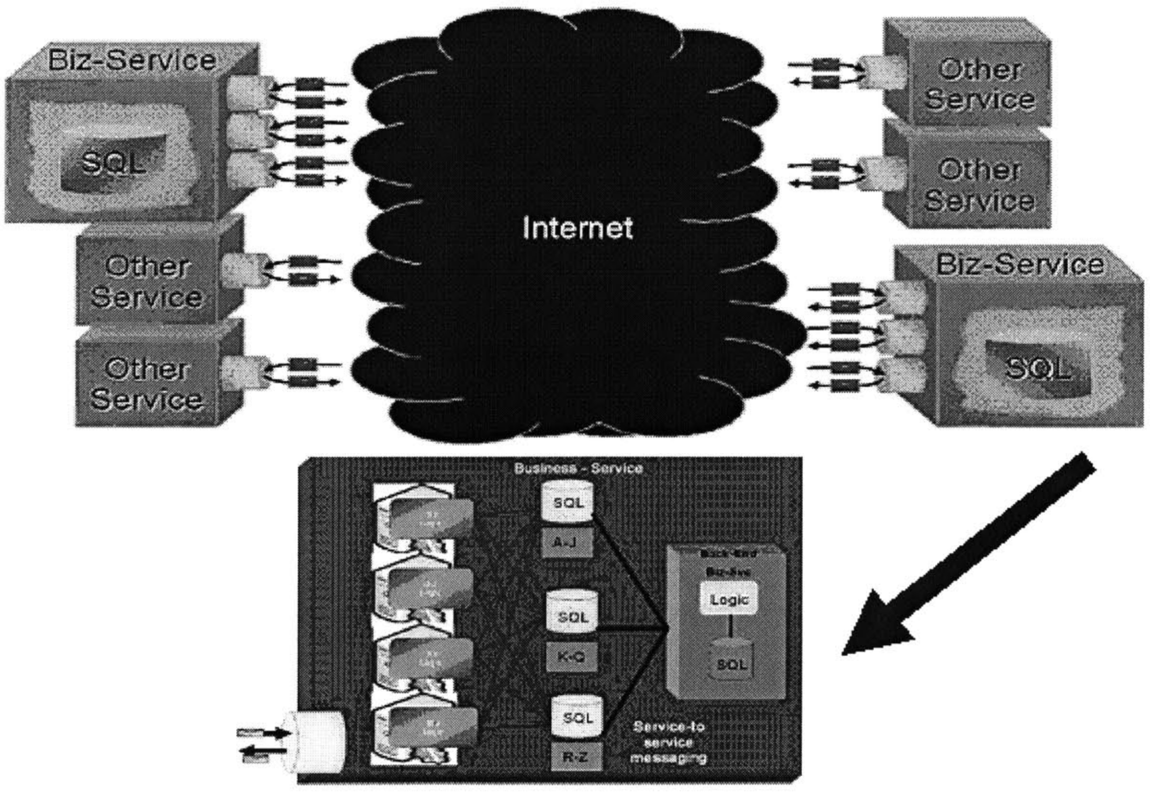


Figure 8 – The interaction of various services in an enterprise environment [33]

Web Service programming model offers a platform neutral approach for integrating diverse applications within and across organizations in a way supported by standards rather than proprietary systems. The major drive for the adoption of Web Services is the ability of the architecture to offer faster and more cost-effective solutions to the integration and interoperability problems. The Web services approach also eliminates the complexity and expense of traditional point-to-point application integration methods. Many businesses are already benefiting from Web services through reduced development and maintenance costs and faster deployment of new applications and services. Following are few live examples of web services in business:

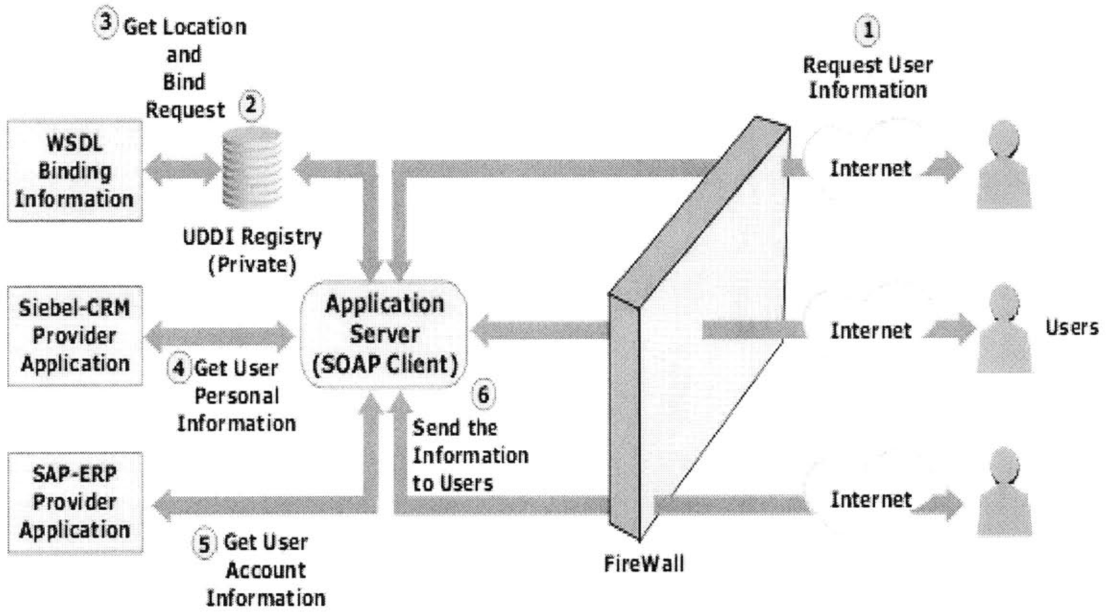
- Leading content management vendors like Documentum, FileNet, Interwoven, Stellent, and Vignette, are exposing their products' full APIs or a predefined set of content management functionality (workflow, search, administration, check-in/ check-out) as Web services.
- Amazon exposes a set of Web Services that can be used to allow any application to find and display Amazon product information. The Web services are compatible with any language capable of posting and receiving data via HTTP and consuming either HTML or SOAP-based XML.

3.2 Web Service Model in a EAI scenario

Figure 35 shows a scenario in which an organization leverages the potential of web services. The portal application running on an application server provides a single point of entry to the internal business processes (like supply chain, CRM, ERP) spread across the organization. The internal business processes expose their functionality in the form of service API's registered within the private UDDI or brokerage firm of the organization. The Web services loosely integrate portal with the internal processes like CRM and ERP

[32]. Following are the sequence of steps describing how the different units would talk to each other:

- Once the user logs on to the organization portal, users request information about the various internal applications.



3 Get Location and Bind Request
2
1 Request User Information
WSDL Binding Information
UDDI Registry (Private)
Internet
Siebel-CRM Provider Application
Application Server (SOAP Client)
Internet
Users
4 Get User Personal Information
6 Send the Information to Users
SAP-ERP Provider Application
Internet
5 Get User Account Information
FireWall

Figure 9 – Scenario for Enterprise Application Integration using Web Services [32]

- The application supporting the portal framework gets information about Web Services made available by the internal applications (CRM and ERP in this figure) by searching in the private UDDI registry.
- The UDDI query gives the location and WSDL binding information of Web Services to the application server.
- The application invokes the required Web Method of each of the applications in ERP and CRM and retrieves the required information.
- The information is then formatted and sent to the user.

3.3 Versioning- Setback in the realization of the model

Building an ecosystem of integrated web services is much more complex than the examples often suggest. There are many characteristics like versioning, security, trust that must be accounted for which are not addressed when describing web services as discussed in Chapter 2.

Version control, one of the major real – world integration problems, has to be solved before standalone web services are deployed. As services evolve with time, the service providers change their interfaces and work flow. The change in the service could be due to:

- Modifying existing services or adding new operations to the existing service.
- Customizing the service as per the requirements of specific customers will force in the maintenance of multiple versions by the vendor.

3.3.1 Web Service Provider market today

The market of the Web service providers is explained in context to the example mentioned in Section 3.2. In the above scenario, the web service exposed by the CRM application modifies one of its existing operations by adding new parameters to the web method. The service provider will expose a new interface and would not want keep the old interface for long as their databases, business logic, have all changed and it won't make sense spending extra resources in running unnecessary multiple versions.

The client, who was hooked to the old service, will break in such a situation. There would exist an incompatibility between the service provider's and the client's interface as the client (portal application) has not upgraded the software to the new interface and would be still consuming the old service which is no longer supported by the service provider.

This is the way the web service provider market operates today. Depending on the complexity, the consumer's service may be down for an unknown amount of time, perhaps weeks or months. The probability of such scenarios makes the service provider unreliable and untrustworthy. No client would ever want to subscribe to such a service which could endanger the functionality of their application.

3.3.2 The Web Services DLL Hell

The situation described above seems a lot similar to *DLL hell*. In *DLL hell*, the installation of a new application on the desktop would install a new DLL as a result of which some of the existing applications would break [21].

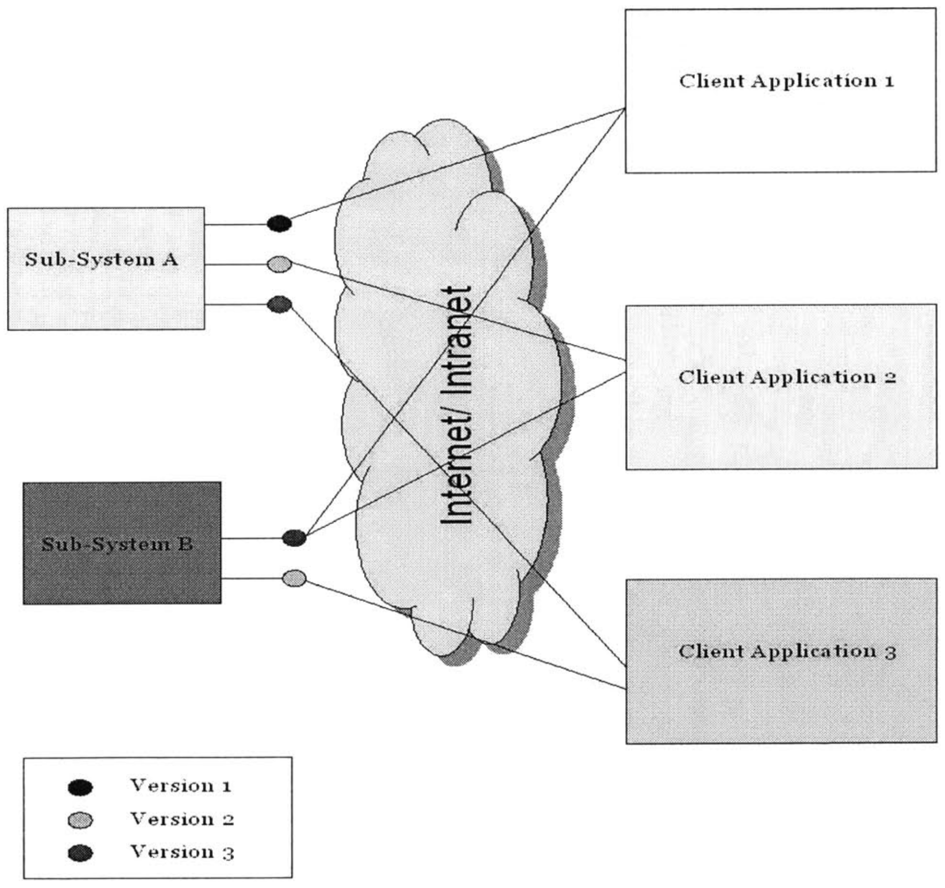


Figure 10 – Scenario showing how different client applications are relying on different versions of a Service

In a web services *DLL hell,* a service upgrade breaks an arbitrarily large number of services that are on many separate and different machines. The consumer of the service can't download an update, uninstall or reinstall. It is required for the consumer to code against the new interface to make the application up and running. There is no technical solution or standard supported by the web service architecture and the problem of upgrading services and the affect on existing client software is not treated in any way.

3.3.3 Solution to the problem?

With the current architecture, the only solution today is through the use of contracts. It would be required to use some sort of versioning strategy to run multiple versions of the service until all the existing clients are upgraded to the new service. Even then, it would be required to maintain versions because of customized interfaces [29, 30].

The contract between the consumer and the provider should explicitly state that the service provider should give some fixed time of notice for the interface change. But the downside of such a contract is the client is forced to change with the changes happening at the service side which is an overhead for the client end.

Chapter 4

Versioning of Web Services- A Technical Perspective

In a service oriented environment, a system is composed of autonomous services which are independently deployed, versioned and secured. A service, like any other program, goes through a common lifecycle of iterative development. Hence, the issue of how the service evolves and how it affects the rest of the system is of prime concern. The changes can take place at two levels:

- XML document types (schema) for applications grow and change. Hence, multiple versions of the schema for a document type are created.
- Old code of an application is hard to eradicate if it requires any change at the interface level.

Before we discuss further about versioning, it is required to define some terminologies to understand the context of the discussion in further chapters [2].

4.1 Definitions

- **Extensibility**: It is a property that enables the evolution of software and is a major contributor to provide loose coupling to system. It provides the independent and potentially compatible evolution of languages.

- **Language**: A language is an identifiable set of vocabulary terms that has defined constraints. For example, Web Service Description Language (WSDL) which comprises of a set of tags to define the service.
- **Content**: Content is data that is a part of an instance of a language.
- **Component**: A component is the realization of a term in a language. XML elements and attributes are components.
- **Instance**: An instance is a realization of the language. Documents that are sent in by clients to activate an operation of a Web Service are an instance of the language.
- **Sender/ Client**: A sender creates or produces an instance for processing by another application.
- **Receiver/ Service**: A receiver consumes an instance that is obtained from a sender.

These terms and the relationships are shown below in the figure.

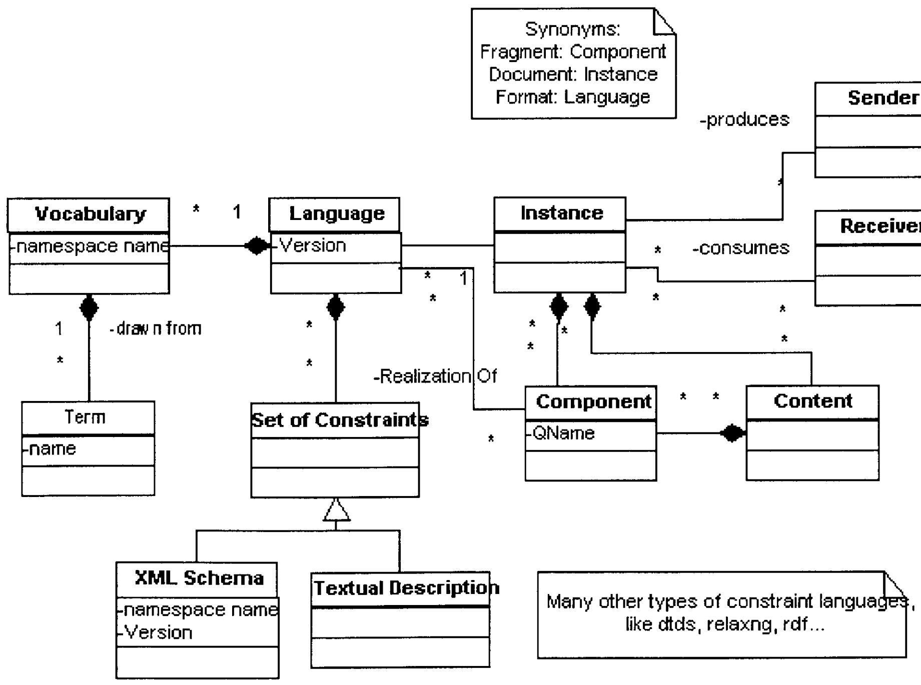


Figure 11 – Relationship between the definitions mentioned above [2]

4.2 Understanding Change Types

Before discussing about the versioning strategies, it is important to understand the types of changes that can take place in a service. There are two types of compatibility changes which software may undergo - backward compatible changes, forward compatible changes [2].

Backward compatibility: A software change is backward compatible if new versions of the software can consume files and data created with an older version of the same program. It is important because it eliminates the need to start fresh again when the software is upgraded to a newer version.

Forward compatibility: A software change is forward compatible if older versions of the software can process data formats and contents created by a newer version. Forward compatibility is harder to achieve than backward compatibility, since, in the backward case, the data format is known whereas forward compatible software needs to cope gracefully with unknown features.

Types of changes that are backward compatible in a Web Service:

- Addition of new operations to an existing service. If existing requestors are unaware of a new operation, then they would be unaffected by its introduction.
- Suppose a new set of complex data types are created for a newly added operation. As long as those data types are not contained within any previously existing types, this type of change will not affect an existing requestor.

Types of changes that are non- backward-compatible are:

- Removing an operation

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

Versioning can only be implemented for backwards compatible changes in a service. When a service evolves in a way that is non-backward compatible, then a new service is created. It is no longer a version of the old service but entirely a new service is generated. However, in the following chapters we will consider the maintenance of services undergoing both types of changes.

4.3 Versioning Schemes

Three levels are identified within a service where versioning could be implemented [9].

- **Interface Versioning**: The problem of interface versioning is not unique to Web Services. Any software that integrates with other independent softwares or components does so by utilizing an interface description which explains how the external environment could interact with the given software. The description of the interface is separated from its implementation. Interface contracts are of great advantage as they allow a developer to make the entire changes to the implementation without impacting the users as long as the published contract is honored. However, non-backward-compatible changes force the versioning of the interface. In the case of Web Services, it would mean versioning of Web Service Description Language which is an XML based interface contract to describe the Web Service and its operations.
- **Versioning of implementation**: Another possible way in which a service could evolve is change in the business logic of the actual implementation in such a way that

the changes are not reflected at the interface level. Such kind of versioning would automatically be forward and backward compatible with the clients.

- **Versioning of data types**: The receiver and sender usually interact via exchange of messages which are defined as a collection of data fields sent or received together between software applications. These data fields can evolve over the period of time. For instance, a service operation might use some user defined types (UDT) which might evolve with time. This would result in the versioning of the type system say the XML schema which would require establishing compatibility with the older versions.

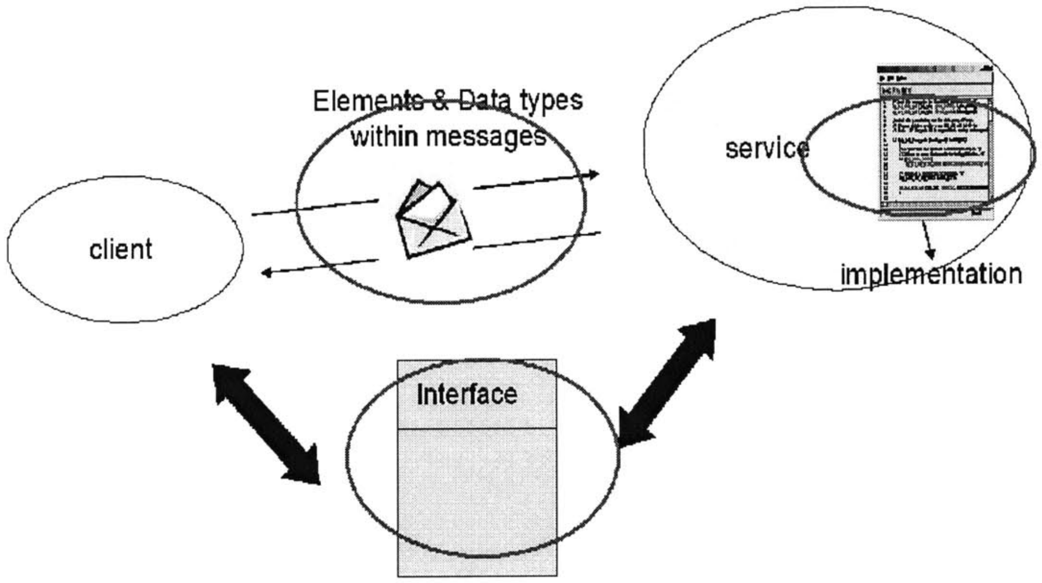


Figure 12 – Versioning Schemes and service- client interaction

4.4 Analysis of versioning scenarios

There are two dimensions of the versioning problem [19].

- **Active versus Passive versioning**. In active versioning, the author explicitly prepares for versioning by implementing some versioning strategy and this thesis is an attempt for the same by discussing versioning techniques and issues. In passive versioning no such explicit preparation is required. For example, XML Schema extensibility requires active insertion of something in the schema document by the author to predict where extensibility needs to occur for touch less evolution (to be discussed later in the thesis) whereas web technology has extensibility passively built in to the system as the default.

- **Engaged versus Disengaged versioning**. In engaged versioning there is a close coupling between the developer and users and regular upgrades of schemas are done at both the sender and receiver ends whereas in disengaged versioning all the components within the system are loosely coupled and there are no frequent schema upgrades.

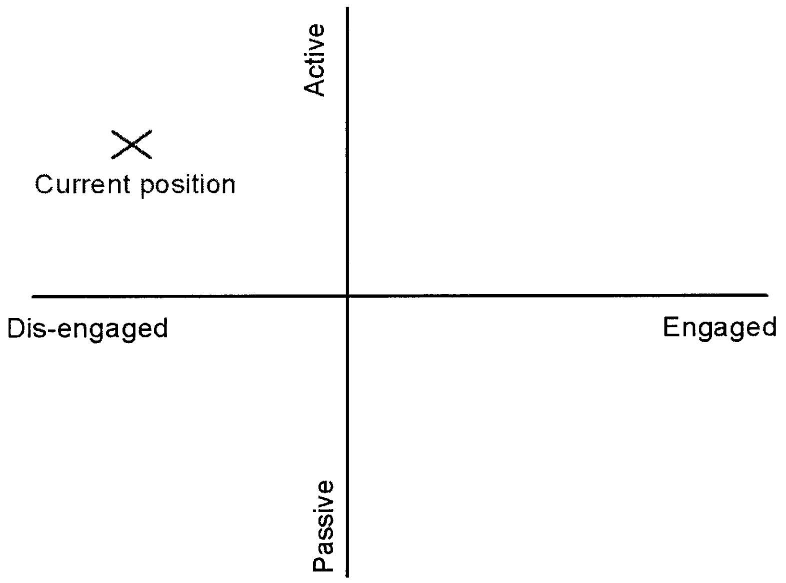


Figure 13 – Position of the Web Service from versioning perspective

To solve the versioning problem it is required to move from the disengaged and active quadrant to disengaged and passive quadrant where no explicit versioning support has to be provided but it is in built within the infrastructure. The following chapters would go deeper into the versioning schemes.

Chapter 5

Versioning – Schema Level

Compatibility is required at the interface level as well as the schema level for distributed systems to interact in a seamless manner. In this chapter we look at schema level versioning followed by interface level versioning in the next chapter.

Operations defined within the service evolve with time. This result in the change of message formats and sequences exchanged between the sender and the receiver. Hence, it is required to incorporate some sort of flexibility at the messaging level so that the messages could be versioned with time. This leads to schema level versioning because schema is the place which reflects the structure, syntax and semantics of the data exchanged between sender and receiver. In addition, the content of a message is required to change in compatible manners. This need leads to the concept of loose coupling at the message level which is defined as [12]:

- openness of the wired contract in terms of data types and parameters expected for operations
- the ability to extend the contract in future

In the following sections, we would discuss the ways of achieving flexible implementation within the schema. This would allow incorporating some versioning strategy.

5.1 Mechanisms to implement Flexibility

Before we discuss about flexibility, it is important to understand the concept of serialization. **Serialization** is a facility that maps a local class definition to an XSD type. Every message part in the doc/ literal resolves to an element in the programming model and varying degrees of coupled ness exists between the element types [22]. With time, the class definitions change and hence would the corresponding messages. Distributed software should be designed in such a way that changes are implemented without having to update on both sides for every change. Following are the desired goals to be achieved:

- Schema should be open in terms of data types of the elements
- Schema should be extensible
- Messages should be able to version/ evolve without invalidating the schema

The changes should be backwards compatible to maintain integrity with the system. Also forward compatible in some cases is desired.

5.1.1 Degree of Openness of Message Types

Openness within messages could be defined as the extent of flexibility in the element types which are exchanged in the message sequence, for instance, method signature of a service. These data types can vary from being very open to very restricted. The degree of openness should not be to an extent that it does not even explain what is being expected over the wire and hence defeating the purpose of a description language and schema. The goal is to develop a wired contract which has all the implementation details and at the same time is open in terms of structure so that it can be extended and versioned. Following are some of the ways to incorporate flexibility in the message schema. The first two strategies focus on the built-in primitive data types (pre-defined in the specification and the programming model) and the third one focus on the flexibility of the user defined data types (they are defined by individual schema and class designers) [22].

5.1.1.1 Defining the most generic parameter type

The most open nature of the schema is to define the data type of the input and output parameters, as the root of the **type system**. For instance, *System.Object* is the root of the CLR type system or *xsd: anyType* is the root of the XSD type system.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://iesl">
        <element name="Name" type="anyType"/>
</schema>


public string AddName( [XmlElement(Namespace="http://iesl")]object Name)
{...}
```

Figure 14 - Sample schema and class definition to define object of the root type

The schema is very open and flexible as one could send any type of data across the wire, until it got serialized. It offers the least coupled ness with respect to the type definition of the input and output parameters of the service methods. However, the downside of this mechanism is the inability to find what is expected or sent through the wire. The semantics of the contents required for the operation have to be negotiated out of band. The contract does not explicitly explain what it wants in the form of an object. There is no implementation coupling. In addition, the object bypasses all the XML validation features which is an inappropriate use of SOAP stack as it defeats the purpose.

5.1.1.2 Defining the parameter type as string

Another mechanism to attain flexibility is to have data in the form of xsd:string or the corresponding String type of the implementation type system. This is possible as anything and everything can be cast as string.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://iesl">
        <element name="Name" type="string"/>
</schema>


public string AddName( [XmlElement(Namespace="http://iesl")] string Name)
{...}
```

Figure 15 -Sample schema and class definition to define object of string type

This mechanism is slightly less open than *anyType*, but has many of the same issues as discussed in the previous bullet.

5.1.1.3 Flexibility in user defined types (UDT's)

User defined types are very restricted in type definition. The members of such data types have a definite sequence and type (can be primitive or user defined) associated with them. This type definition is validated against the schema in scenarios where message contain UDT's. Consider the following user defined type *Name* with the following type definition.

```
<schema targetNamespace="http://iesl" xmlns="..." >
   <element name="Name">
       <complexType>
         <sequence>
             <element name="firstName" type="string"/>
         </sequence>
        </complexType>
     </element>
</schema>


public string AddName( [XmlElement(Namespace="http://iesl")] Name name)
{...}
```

Figure 16 - Type Definition of a User Defined Type - Name

So the message would fail against validation in situations where it doesn't comply with the schema. Tight coupling is observed between the two ends as the validation rules are rigid. Looking form another perspective, user defined types have all details about the implementation of the method as they have appropriate and specific description of what the method signatures is and what is expected over the wire. Flexibility in such cases could be achieved by adding special XML tags and attributes to the schema and class definition. Some of the elements defined in the specification are discussed below:

1. *User Defined Types and Optional Data.* The instance documents of XML schema might not contain each and every element defined in the schema. However, the schema should validate such instance documents. This can be achieved by inserting attributes in the element definition which categorizes elements as *must-haves* and *nice-to-haves*. For example, attribute *"minOccurs=0"* marks a particular element as optional. In the following example, the *middleName* element is marked as optional in the *Name* type definition.

```
<schema targetNamespace="http://iesl" xmlns="..." >
 <element name="name">
   <complexType name="Name">
      <sequence>
          <element name="firstName" type="string"/>
          <element name="middleName" type="string" minOccurs="0"/>
          <element name="lastName" type="string" />
      </sequence>
   </complexType>
 </element>
</schema>
```

Figure 17 - The Name type definition along with the minOccurs attribute

2. *User Defined Types and Unions.* This XML tag allows having different combinations of elements in the instance documents. The *choice* element allows only one of the elements contained in the *<choice>* declaration to be present within the containing parent element. This element allows the validation of various combinations of elements in XML instances with the same schema and hence provides some amount of flexibility.

```
<schema targetNamespace="http://iesl" xmlns="..." >
 <element name="name">
   <complexType name="Name">
     <sequence>
         <element name="firstName" type="string"/>
         <element name="middleName" type="string" minOccurs="0"/>
         <element name="lastName" type="string" />
          <choice>
            <element name="ssn" type="string"/>
            <element name="driverslicence" type="string"/>
            <element name="passportnumber" type="string"/>
           </choice>
     </sequence>
   </complexType>
 </element>
```

Figure 18 - The Name type definition along with the choice element

In the above example, the instance document for the schema can contain only one of the three elements namely *ssn, driverlicense* or *passportnumber*.

5.1.2. Open Content Model

The content model describes the content structure of elements and attributes in the XML document. To describe the content structure, attributes like the *model, minOccurs, maxOccurs, order, content, minLength, maxLength, default,* and *type* are attached to the element definition. It can take two forms: **Open** and **Closed**. In an open content model, all required elements mentioned in the schema must be present in the instance document, but it is not an error for additional elements to also be present. On the other hand, a closed content model describes all and only what may appear in the content of the element [20]. By default, content model of an element is open in nature. Consider the following schema snippet:

```
<xsd:element name="Book" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Title" type="xsd:string"/>
      <xsd:element name="Author" type="xsd:string"/>
      <xsd:element name="Date" type="xsd:gYear"/>
      <xsd:element name="ISBN" type="xsd:string"/>
      <xsd:element name="Publisher" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Figure 19- Schema of a Book element

The schema enforces that the instance documents should contain the *Title, Author, Date, ISBN* and *Publisher* in the *Book* definition.

For scenarios where it is required to make this schema flexible and extensible, the extensibility feature of the open content models could be leveraged. This is achieved by the following ways explained in the subsections [20].

5.1.2.1 Derive a new type from the element type definition

Extensibility can be achieved by **type substitution** in which new types could be created by using the existing user defined types. In Figure 15, a *BookType* definition was created. Another element type, *BookCatalogue* is created which is a collection of elements of type *BookType*.

```
<xsd:complexType name="BookType">
  <xsd:sequence>
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Author" type="xsd:string"/>
    <xsd:element name="Date" type="xsd:string"/>
    <xsd:element name="ISBN" type="xsd:string"/>
    <xsd:element name="Publisher" type="xsd:string" />
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="BookCatalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Book" type="BookType"  maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Figure 20 - Extension of Book element using Type Substitution

The contents of the *BookType* can be further extended by using the *extension* element with the *base* attribute as mentioned below.

```
<xsd:complexType name="BookTypePlusReviewer">
  <xsd:complexContent>
    <xsd:extension base="BookType" >
      <xsd:sequence>
        <xsd:element name="Reviewer" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 21 - Sample showing usage of the extension element in the schema

The element *BookTypePlusReviewer* has been defined with the same base schema as *BookType* indicated by the *base* attribute. In addition, other schemas can import/ include the *BookType* and define types which derive from *BookType*. This type substitution mechanism is a powerful extensibility mechanism. However, it suffers from two problems.

- *Location restricted extensibility*: The extensibility is restricted to appending elements onto the end of the content model i.e. after the *Publisher* element in the *Book* type definition. It is not possible to add elements at the beginning or middle of the existing content definition.
- *Unexpected Extensibility*: With the above mentioned mechanisms it is easy to forget the fact that someone could extend the content model using type substitution mechanism. Extensibility is unexpected as it is not explicitly stated where extensibility may occur. The other technique of using wild cards <any> gives such capabilities.

5.1.2.2 Using wild cards

The concept of wild cards allows the addition of extra elements and attributes from other or same namespaces to the instance documents. This is typically achieved with the *<any>* element tag [17]. It allows insertion of elements from any namespace.

```
<xsd:element name="BookCatalogue">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Book" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Title" type="xsd:string"/>
            <xsd:element name="Author" type="xsd:string"/>
            <xsd:element name="Date" type="xsd:string"/>
            <xsd:element name="ISBN" type="xsd:string"/>
            <xsd:element name="Publisher" type="xsd:string"/>
            <xsd:any namespace="##any" minOccurs="0"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Figure 18 - Sample showing the addition of <any> element in the schema

In the above example, an explicit element *<any>* is added to the schema definition of the *BookType* element. It means any well-formed XML element may occur in place of the *<any>* element and that XML element may come from any namespace.

Suppose the author also wants to add a declaration for a *Reviewer* element in the document instance. The addition of this *Reviewer* element in the instance document does not invalidate the above schema because it allows addition of new elements from any namespace. Hence, we see how an instance document author can enhance the instance document with an element that the schema designer may have never even envisioned.

```
<BookCatalogue xmlns="http://www.publishing.org" xmlns:rev="http://www.Publishing.org"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation= "http://www.publishing.org/BookCatalogue.xsd">
  <Book>
    <Title>Past, Present and Future</Title>
    <Author>Anamika Agarwal</Author>
    <Date>2004</Date>
    <ISBN>94303-12021-43892</ISBN>
    <Publisher>McMillin Publishing</Publisher>
    <rev:Reviewer>
      <rev:Name>
        <rev:First>Abel</rev:First>
        <rev:Last>Sanchez</rev:Last>
      </rev:Name>
    </rev:Reviewer>
  </Book>
</BookCatalogue>
```

*Figure **22** - Instance document containing Reviewer element instead of Any element*

With the *<any>* element, one can have a control over where, and how much extensibility one wants to provide. This can be achieved by using *maxOccurs* attribute in the *<any>* element definition which limits the number of elements that could be added.

```
<xsd:any namespace="##any" minOccurs="0" maxOccurs="2"/>
```

The wild card allows to do the following in terms of extensibility:

- It allows to put the *<any>* element specifically where we desire extensibility
- If extensibility is desired in multiple locations, multiple insertions of the *<any>* element is possible.
- With *maxOccurs* one can specify "how much" extensibility to allow.

However, there are certain namespace concerns with the *<any>* element [16].

Scenario 1 - If the wildcard is used with the *##other* namespace (all namespaces except for the target namespace). The problems with this approach are observed when the examining elements and wildcards are structured together as siblings. The problem could be summarized as *the namespace author cannot extend their schema with extensions and correctly validate them*. This would prohibit the schema author to change or extend the document hence the versioning using *##other* namespace is not possible as the author cannot modify it.

Scenario 2 - If the wildcard is used with ##any or ##targetnamespace [16]. Such a wildcard is not possible due to the *Unique Particle Attribution rule* which does not allow a wildcard adjacent to optional elements or before elements in the same namespace. The following schema is illegal.

```
<xs:complexType name="nameType">
        <xs:sequence>
                <xs:element name="first" type="xs:string" />
                <xs:any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="last" type="xs:string" minOccurs="0"/>
        </xs:sequence>
</xs:complexType>
```

Figure 23 - Schema invalidating the Unique Particle Attribution Rule

This is one significant problem with the XML schema and its evolution using wildcards in the current specification i.e. XML Schema 1.1. The XML schema has to undergo changes to overcome this restriction on wild card insertion. For example, a "priorty" wildcard model, where an element definition is given more priority than the wildcard element could be suggested. Getting into the details of the schema constraints is out of the scope of the thesis and could be further read at this reference.

5.1.2.3 Using Extension Elements

Having discussed the pros and cons of the current specifications, the following mechanism is the best proposed solution for implementing extensibility [18]. It allows backwards and forwards compatibility, and validation using the original or the extended schema. In this technique, a wildcard is inserted along with an *ExtensionType* element, so that the element definitions and the wildcards are not at the same level within the structure.

```
<xs:complexType name="nameType">
        <xs:sequence>
                <xs:element name="first" type="xs:string" />
                <xs:element name="extension" type="tns:ExtensionType" minOccurs="0"
                maxOccurs="1"/>
                <xs:element name="last" type="xs:string" minOccurs="0"/>
        </xs:sequence>
</xs:complexType>
<xs:complexType name="ExtensionType">
        <xs:sequence>
                <xs:any processContents="lax" minOccurs="1" maxOccurs="unbounded"
                namespace="##targetnamespace"/>
        </xs:sequence>
</xs:complexType>
```

An extended instance is

```
<name>
        <first>Anamika</first>
        <extension>
                <middle>U</middle>
        </extension>
        <last>Agarwal</last>
</name>
```

Figure 24 - Schema showing the addition of wild card along with the extension element

However, the tradeoff in this solution is each subsequent version will increase the nesting by 1 level. But it is not a major issue as compared to backward and forward compatible evolution along with validation. In addition, other issues to be considered with the existing wild card model:

- *Allowing of undefined elements only.* The wild card should not just create a hole for anything to be inserted. In addition, another constraint which has to be considered is that a wildcard should allow only those elements that have not been defined- effectively from other namespaces plus anything not defined in the target namespace – is another useful model.
- The wildcard construct requires the author to place wildcards in every possible extensible spot, or to guess where the extensibility will occur.

Having discussed about flexibility and extensibility it is required to understand how it could be leveraged in implementation of versioning. Before that it is important to understand how versioning differs from extensibility. The next section, discusses the same.

5.2 Extensibility Vs Versioning

Extensibility provides a partial solution to achieve versioning in a scenario where one needs to make the schema extensible so that it could be extended further in the future. Following points highlight the differences between extensibility and versioning [18].

Observations about extensibility:

- Extensibility is about evolution across space
- It is done in a decentralized manner
- The goal is that other parties could plug their extension in

- Multiple extensions can be authored at any given point of time
- The namespaces are unpredictable and extensibility is expected not to break backwards compatibility

Observations about Versioning:

- Extension of schema construct is a type of change that can lead to versioning. However, there are other types of changes as well that might lead to versioning
- Versioning is done by a single authority. Any new version is controlled by the same authority as the first version
- The evolution over time happens in a linear fashion. V1 is followed by V2 etc
- The evolution may introduce significant new features, including incompatible changes
- Namespaces could be maintained by a single authority in a linear progression manner

5.3 Schema Changes

The schema constructs can change in a number of ways. Following are the main reasons that bring about a change in the schema:

- New concepts are added (e.g. new elements or attributes added to format or new values for enumerations)
- Existing concepts are changed (e.g. existing elements & attributes should be interpreted differently, added elements or attributes alter semantics of their parent/owning element)
- Existing concepts are deprecated (e.g. existing elements & attributes should now issue warning when consumed by an application)

- Existing concepts are removed (e.g. existing elements & attributes should no longer work when consumed by an application)

In addition to leveraging the extensibility techniques, there are other versioning approaches that could be implemented to achieve all the above mentioned changes.

- Change the internal schema version attribute supported by the schema
- Create a schema version attribute on the root element
- Change the schema's targetNamespace
- Change the name and location of the schema

Each of the above mentioned approaches are explained in the sub sections below [15]:

5.3.1 Change the internal schema version attribute

Versioning of schema constructs can be achieved by changing the internal schema version attribute. This approach utilizes the optional version attribute at the beginning of the XML schema. One could simply change the version number in this version attribute to track the changes within a schema.

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema“ elementFormDefault="qualified"
attributeFormDefault="unqualified" version="1.0">
```

Figure 25 - Sample schema element using the version attribute

The advantages to this approach are:

- It is easy to implement and also is a part of the specification
- Instance documents would not have to change if they are still validated with the new version of the schema

- By iterrogating the version attribute, the application could identify if the schema has changed and accordingly appropriate action can be taken

However, one major disadvantage with this approach is the validator ignores the version attribute. Hence it is not a required constraint.

5.3.2 Create a schema version attribute on the root element

In this approach, a new versioning attribute is introduced in the schema element definition. namely, *schemaVersion* in the below examples. This attribute could be used to capture the schema version. There are two ways to leverage this attribute to implement verisoning logic.

- *Option 1*: One way is to make this attribute required and the value fixed. Then each instance that uses this schema would require to set the value of the attribute to the value used in the schema. This would make *schemaVersion* a constraint that is enforced by the validator.

```
<xs:schema xmlns="http://www.ieslSchema"
targetNamespace="http://www.ieslSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
<xs:element name="Name">
<xs:complexType>
....
<xs:attribute name="schemaVersion" type="xs:decimal" use="required" fixed="1.0"/>
</xs:complexType>
</xs:element>
```

Figure 26 - Sample schema using a custom created versioning element

This technique enforces a required constraint to the schema and hence instances would not validate without the same version number. However, the downside is the schema version number in the instance has to match exactly. This does not allow an instance to indicate that it is valid across multiple versions of the schema.

- *Option 2:* This approaches uses the *schemaVersion* attribute in a diferent context altogether. Instead of using this attribute to capture the version of the schema, it is used in the instance to declare the version(s) of the schema with which the instance is compatible.

The value of the *schemaVersion* attribute could be a associated with different context to define how the attribute has to be used. For instance, the convention might be that the *schemaVersion* attribute declares the latest schema version with which the instance is compatible. This would mean that the instances should be valid with schemas with version number equal to the value of the attribute or earlier.

Sample Schema (declares it's version as 1.3)

```
<xs:schema xmlns="http://www.ieslSchema"
targetNamespace="http://www.ieslSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified" version="1.3">
<xs:element name="Example">
<xs:complexType>
....
<xs:attribute name="schemaVersion" type="xs:decimal" use="required"/>
</xs:complexType>
</xs:element>
```

Sample Instance (declares it is compatible with version 1.2 (or 1.2 and other versions depending upon the convention used))

```
<Example schemaVersion="1.2"
xmlns="http://www.iesl"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.iesl.com/ MyLocation/iesl.xsd">
```

Figure 27 - Sample Schema showing the use of Schema Version attribute as in Option 2

In the example mentioned above, the sample schema indicates the version as 1.3. However, the instance shows the value of the *schemaVersion* as 1.2 which explains that this instance is compatible with version 1.2 and earlier. So, in this case, the above schema won't be able to validate the instance document. Following are the advantages and disadvantages with this mechanism.

1. *Advantages:*

- Instance documents are not required to change if they remain valid with the new schema versions.
- An application would receive an indication that the schema has changed.

2. *Disadvantages*

- Requires extra processing by an application. In the above example, an application would have to pre-parse the instance to determine what schema version is compatible and compare this value to the version number stored in the schema file.

5.3.3 Change the schema's targetNamespace

Changing the targetNamespace of the schema is another way to incorporate some versioning information. The schema's *targetNamespace* could be appended with some verison information in the form of version number or a date stamp. So, that part of the *targetNamespace* could be used to reflect that the version of the schema.

```
<xs:schema xmlns="http://www.iesl.mit.eduSchema/V1.0"
targetNamespace="http://www.iesl.mit.eduSchema/V1.0"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
```

Figure 28 - Schema Sample showing the use of targetNamespace for version information

The pros and cons of this technique are the following:

- It is required to assure that there are no compatibility issues with the new schema. At the least, it is required that the instance documents that use the schema must change to direct to the new *targetNamespace*.
- Any schema that include this new schema would have to incorporate the change of the targetNamespace to maintain compatibility between the two schema.

With this approach, instance documents will not validate until they are changed to designate the new *targetNamespace*. All instance documents are forced to change, even if the change to the schema is really minor and would not impact an instance.

5.3.4 Change the name/location of the schema

This approach changes the filename or location of the schema. It uses the naming convention very similar to the naming convention of *targetNamespaces*. It enables to identify which version is the most current by iterrogating the version number or the date stamp associated to the name. This strategy has more disadvantages than advantages.

- As with the earlier stated option, this approach forces all instance documents to change, even if the change to the schema would not impact that instance.
- Any schemas that import the modified schema would have to change since the import statement provides the name and location attributes of the imported schema.
- An application receives no hint that the meaning of various elements/attribute names has changed.

The schemaLocation attribute in the instance document is optional and is not authoritative even if it is present. It helps the processor to locate the schema. However, relying on this attribute solely is not a good practice.

5.4 XML Schema Versioning – Best Practices

Number of ways were discussed to make the schema flexible, extensible and versionable. Some of the salient features or practices which could be adopted are the following [15]:

- Capture the schema version somewhere in the XML schema
- Indentify the instance document, meaning what versions(s) of the schema with which the instance is compatible.
- Make the previous versions of the schema available so that applications could use the previous versions in cases were backwards compatibility is not supported. It allows users to migrate to new versions of the schema in a seamless manner.
- Another promising solution is to pre-parse the instance and choose the appropriate schema based on version number present in the instance document. For example, one could have schema location URI point to a document that includes the list of all the available versions of the schema. A routing service or a tool could be used to appropriately channelize the instance documents to the corresponding schemas which could validate those instances. The trade off is the consumption of additional resource as it requires two passes of the XML instance. First one, to get the correct version of the schema and the other one to validate that instance.

Chapter 6

Interface Versioning – WSDL

Chapter 5 focused on the how to handle changes of data types and definitions which are the integral part of messages in the service oriented model. Having discussed that, we now move one step ahead in this chapter and talk about the versioning of the interface of the service. The interface of Web Service is described by a standard, platform neutral XML based language known as Web Services Description Language (WDSL). It is the key in providing a universal API which allows the description of remote services in a standard, language-agnostic way and could be consumed by any arbitrary processor. There are two different versions of the WSDL specifications to be considered:

- WSDL 1.1 released in 2001, has been deployed by vendors, and for a limited set of services is known to be interoperable.
- WSDL 2.0 is more precise in its language, reducing ambiguities and has the potential to enable a larger range of services to be supported. But it is still in the working phase and has not approved by W3C as a standard. However it is one of the potential recommendations made to the forum and has not been deployed as yet.

While proposing techniques of versioning at the WSDL level, the scope is confined to:

- Specifications of WSDL 1.1
- Implementation samples use the .NET framework.

- SOAP 1.1 over HTTP is used as the network protocol for the transfer of messages.
- Literal encoding of messages is used.

6.1 Anatomy of WSDL 1.1

WSDL 1.1 describes a Web Service in two parts-*abstract definition* and *concrete definition [26].*

The *abstract definition* contains a description of the Web Service that is independent of the platform and language, with no mention of how messages are transported between the client and the server. The abstract definition specifies interface of the service but not how to access that interface. The *types* element holds data definitions relevant to the messages. WSDL prefers the use of XML Schema Data (XSD) types as the canonical type system. The versioning at this level have been discussed in the earlier chapter. A *message* consists of one or more logical parts which are associated with a type from the type system. A *portType* is equivalent to an interface and is a named collection of abstract operations and messages involved.

The *concrete definition* adds the network and protocol details describing how to access the Web service. A *binding* defines a message format and protocol details for operations and messages defined by a particular portType. A *port* defines an individual endpoint by specifying a single address for a binding. It is the network location where the web service is implemented. A *service* is defined as a collection of multiple ports.

The framework and elements of WSDL are not explained in much detail in this chapter as the focus of the thesis is to look at the versioning aspect. Hence, only a brief introduction is provided for both the versions of WSDL.

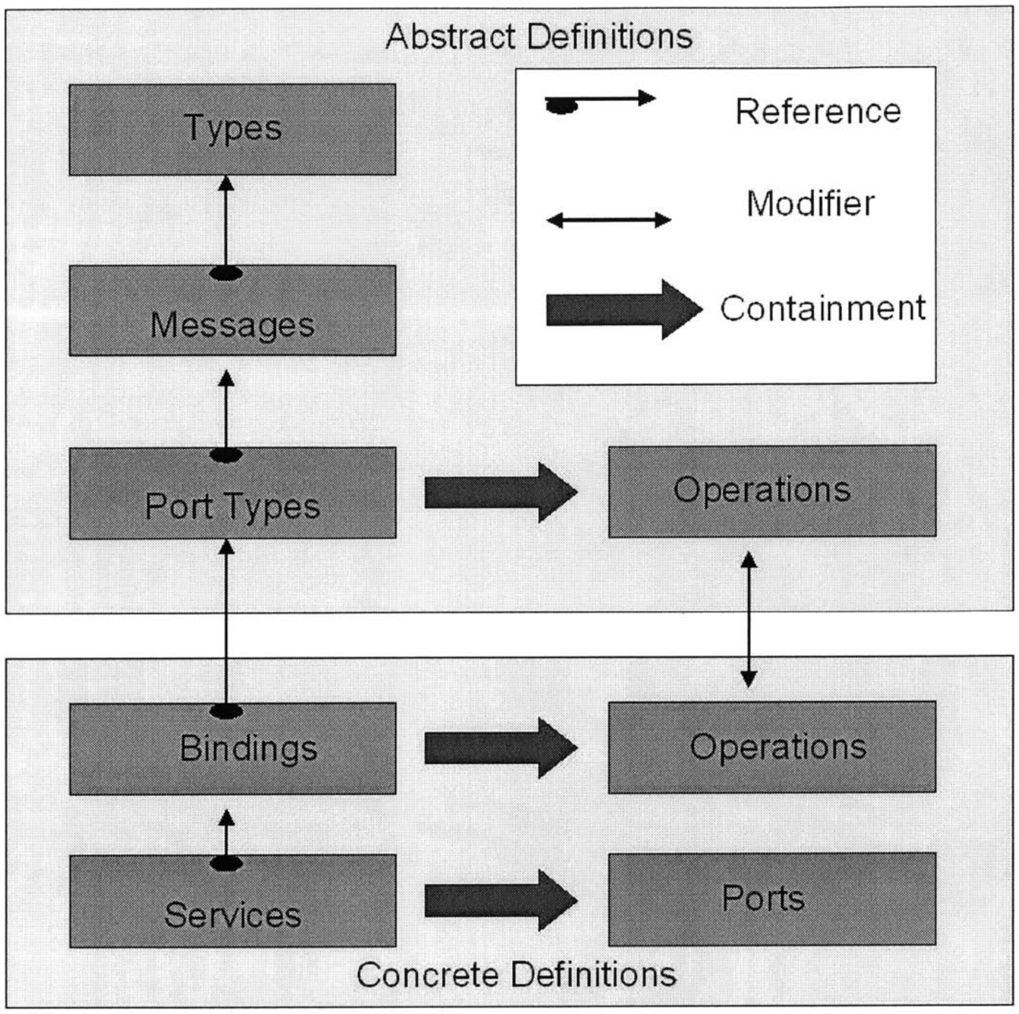


Figure 29 - Anatomy of WSDL 1.1 showing the component interaction [14]

6.2 Anatomy of WSDL 2.0

In comparison to portType in WSDL 1.1, WSDL 2.0 defines a collection of operations as an *interface* which describes a set of messages that a service can send/ receive. It does this by grouping related messages into *operations*. An interface is bound to concrete protocol and message format via one or more bindings. A *binding*, and therefore an *interface*, is accessible via one or more endpoint, each endpoint having its own URI. A *service* component is a collection of endpoints which are linked to one and only one *interface* that the service supports. Further, WSDL 2.0 considers that an *interface* represents the behavior of a *resource* on the World Wide Web [27].

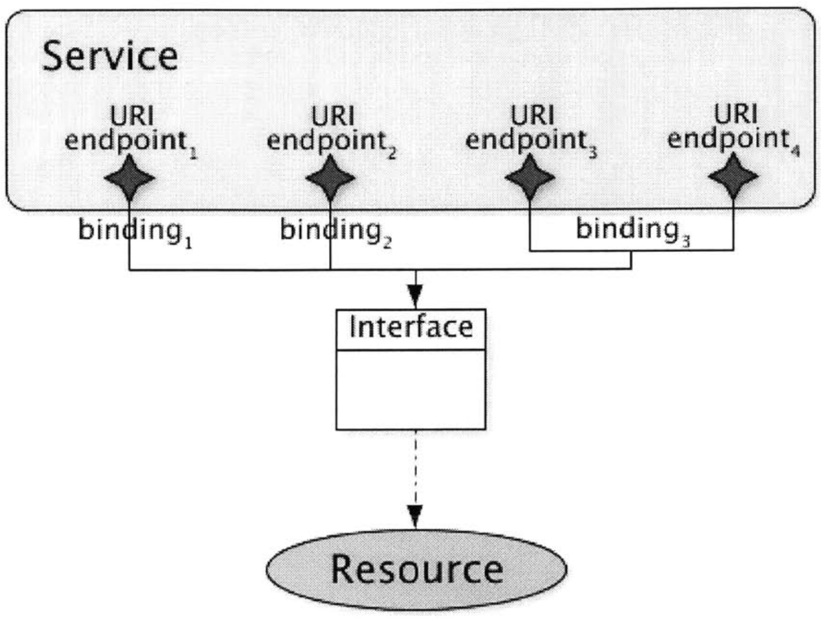


Figure 30 - Anatomy of WSDL 2.0 [27]

WSDL 2.0, looks at the same concept of service description with a different terminology and perspective. However, it is still in the working phase and the final draft has not been decided as yet.

6.3 Execution steps for a sample .NET Web service

Let us consider a very simple .NET web service as the starting point on which the versioning techniques would be built. This web service named '*BasicMathOperations*' handles messages for two basic math operations *Add* and *Subtract*. It is a .NET Web Service.

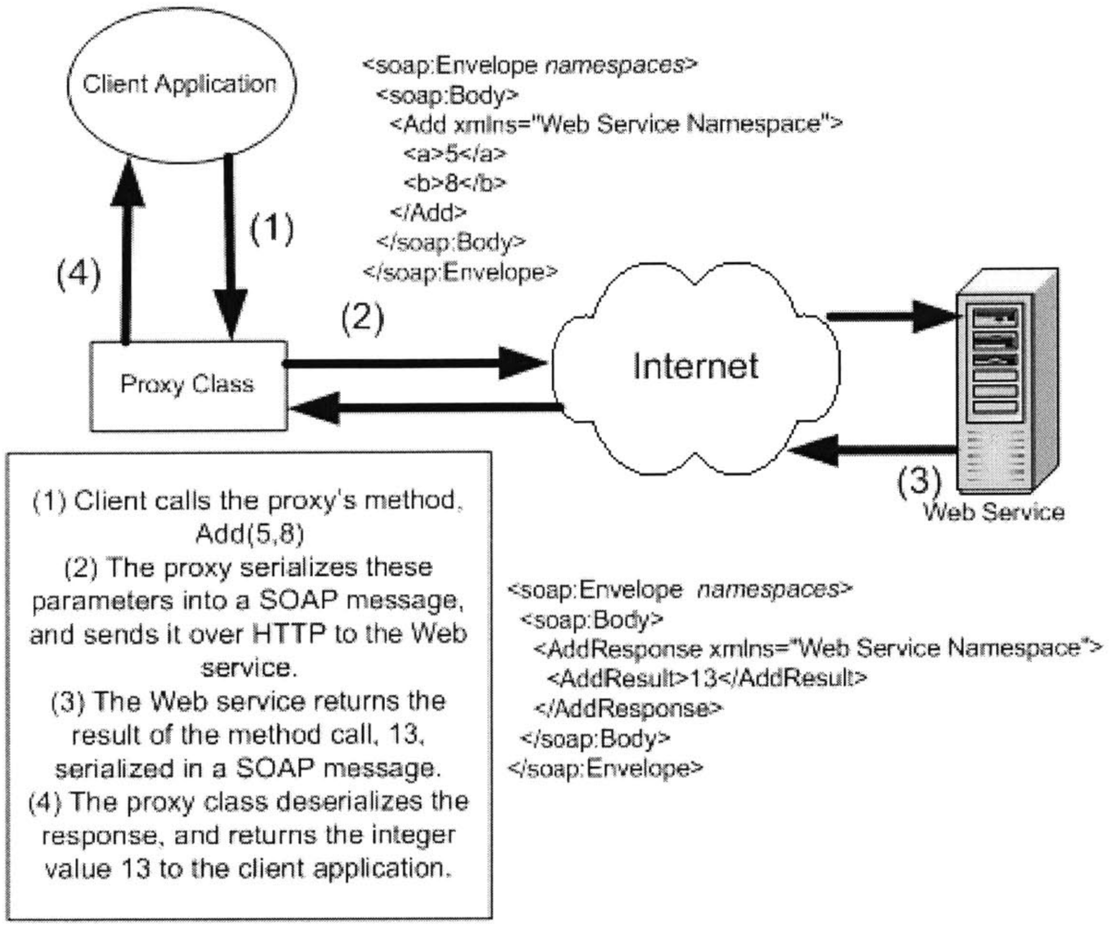
```
[WebServiceAttribute(Name="BasicMathOperations",
Namespace="http://iesl.mit.edu/MathService/2004/05/09")]
      public class Mathv1 : System.Web.Services.WebService
      {
            public Mathv1()
            {
                  InitializeComponent();
            }


            [WebMethod]
            public int Add(int a, int b)
            {
                  return a+b;
            }
            [WebMethod]
            public int Subtract(int a, int b)
            {
                  return a-b;
            }
      }
}
```

Figure 31 - Sample .NET Web Service named BasicMathOperations

There are two ways of creating a Web Service.

- **Code first** – Class implementation of the service is created first which is used to generate the WSDL interface.
- **WSDL first**- WSDL interface of the service is created first which is used to generate the implementation class.

In the above example, we have adopted the *Code first* approach. The Web Service class (*Mathv1*) is implemented in .NET as a *.asmx* file located at a specific URL on the server. A *targetNamespace* is defined for the service to qualify the elements of the service with the namespace and to avoid collision and semantic ambiguity with similar elements of other services. The generated WSDL describes what requests and responses could be handled as well as XML formats and data types that are exchanged between the sender and the receiver. All the messages and types are scoped using *targetNamespace* specified in the Web Service class implementation [14].



```
<soap:Envelope namespaces>
  <soap:Body>
    <Add xmlns="Web Service Namespace">
      <a>5</a>
      <b>8</b>
    </Add>
  </soap:Body>
</soap:Envelope>
```

(1) Client calls the proxy's method, Add(5,8)
(2) The proxy serializes these parameters into a SOAP message, and sends it over HTTP to the Web service.
(3) The Web service returns the result of the method call, 13, serialized in a SOAP message.
(4) The proxy class deserializes the response, and returns the integer value 13 to the client application.

```
<soap:Envelope  namespaces>
  <soap:Body>
    <AddResponse xmlns="Web Service Namespace">
      <AddResult>13</AddResult>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 32 – Steps showing the interaction of a Client accessing a Web Service [31]

The proxy, created at the client end using the WSDL, interacts with the Web Service in such a way that it takes into account the *targetNamespace* in the WSDL file [31]. This namespace is sent along with every SOAP message (request/response) to correctly determine what to do with an incoming message, based on namespace value. In addition, the namespace is used to create the *SOAP Action* portion of the message sent to the service. The SOAP action is used by the .NET Framework to route the request to the code that handles the request. The action is created by concatenating the targetNamespace to the name of the method. *Figure 28* shows how web service interacts with the client in a .NET environment.

6.4 What is an Interface version?

Before we start discussing about versioning, it is essential to understand what does an *interface version* mean in terms of Web Services and how is it different from completely *distinct interfaces [28].*

When an interface undergoes a non-backward-compatible change, it is not a version of the earlier service but an entirely new service is created. It no longer represents an instance of the original service, but is a completely new service. An *interface version* is *always backwards-compatible with the preexisting service.* This would mean scenarios like either 1) new operation is added while all prior operations are maintained, or 2) the existing signatures are changed in a manner that is compatible with the original interface. Opportunities for such changes are limited.

Since backwards-compatibility is a requirement for each version, a means to distinguish between earlier and later versions of the same interface is required. For the same, unique namespace for every version in the WSDL document is a recommended practice.

6.5 Naming Convention for Versioning

Having an identifier for each version release of a Web service is essential for simplifying the management of the releases. The approach should facilitate recognition of multiple versions of Web service, including both existing and new interfaces. Different strategies could be implemented like [29]:

- Append version information to the name of the service

- Append version information to the targetNamespace of the service

In WSDL 1.1, version indication is mostly done via the *targetNamespace* attribute of the *definitions* element. As mentioned in Section 5.3, this namespace gives the SOAP messages meaning by relating the messages to a specific bit of code implemented somewhere on the server. This attribute could be used in a number of ways to indicate the version. Version related information can be in the form of 1) version number or 2) date stamps

- *"_vMajor#_Minor#"*. The targetNamespace could be named as *http://web.mit.edu/Service1/v1/1/*. This gives the interface a unique namespace identifier. It gives an obvious indicator of version as well as a place to increment that version in a way that is human understandable.
- *"year/month" or "year/month/date/"*. Another option for the naming of targetNamespace is incorporating a date stamp like *http://web.mit.edu/Service1/2004/05/09/*. The resolution of the date stamp could be used to determine the frequency of versioned releases. This option has an advantage over the earlier one as this versioning scheme is employed by the newer XML specifications. Versioning by date provides an added bonus of figuring out when the version was released.

6.6 Adding New Operations

When new *operations* are to be added to an existing service, it would be expected to combine conceptually related operations together in the same *portType* and hence the same *binding* as per the specifications of WSDL 1.1. However, the specification does not allow extending an existing *portType* using *import* attribute. There is no concept of inheritance of a *portType* by another *portType* (using import) and extending it further by adding further *operations*.

However, WSDL 2.0, which is still in the working phase, do provide this flexibility of inheriting interfaces. An *interface* can optionally extend one or more other interfaces. In such cases the new interface contains the operations of the interfaces it extends, along with any operations it defines.

Backward compatible changes can be incorporated within the same *targetNamespace*. However, to keep a track of the versions of a given service, it is a good practice to associate some uniqueness for each version maybe in the form of a distinct namespaces. The following sub sections describe ways in which versioning could be incorporated using the specifications of WSDL 1.1.

6.6.1 Technique–Maintaining Namespaces of Messages

Multiple versions of a WSDL document are maintained by associating a unique namespace with each version [23]. The uniqueness is achieved by appending version numbers or date stamps as explained in section 5.5. Let us consider a backward compatible change - *addition of new operation to an existing service.* Suppose a new operation (for example *Multiply*) has to be added to the existing service described in Section 5.3. Following are the requirements for the new version:

- The web service and the corresponding WSDL created should indicate a new XML Namespace to maintain version control.
- The operations available in the older version (*Add and Subtract*) should be accessible to existing client – should be backwards compatible.

- The three operations should be available in one *binding* as they are related operations meaning they should appear in the same proxy class generated at the client side.

In order to maintain backward compatibility, it is required to manage the XML namespaces used by the input and output messages while keeping all the operations in one *binding*. The following is achieved by defining an entirely new service with a different targetNamespace and adding explicit attributes to set the namespace used by the request and response messages of each operation. As a result, the generated WSDL will appropriately reflect the correct actions.

Each exposed operation has *SOAPAction* and *Namespace* (Request and Response) attributes which are set using the *SoapDocumentMethodAttribute*. The WSDL generated will also incorporate the changes applied on the class definition. The *SOAPAction* and *Namespace* for already existing operations still remain the same whereas the new operations can have the *SOAPAction* and namespace qualified using the new namespace.

This is one way of incorporating versioning in already deployed services. However, there is a shortcoming with the proposed technique. *One endpoint supports two bindings.* In this strategy, no changes are required at the existing client side. As a result the client proxy would have a class definition of the old binding. However, new version is hosted at the same endpoint on the server side corresponding to the new binding which is backward compatible to the already existing operations. However, this fact is not evident by either the client or the server.

```
[WebServiceAttribute(Name="BasicMathOperations",
Namespace="http://iesl.mit.edu/MathService/2004/06/03")]
public class Mathv2 : System.Web.Services.WebService
{

	[WebMethod]
	[SoapDocumentMethodAttribute("http://iesl.mit.edu/MathService/200
	4/05/09/Add",
	RequestNamespace="http://iesl.mit.edu/MathService/2004/05/09",
	ResponseNamespace="http://iesl.mit.edu/ MathService/2004/05/09")]
	public int Add(int a, int b)
	{
		return a+b;
	}

	[WebMethod]
	[SoapDocumentMethodAttribute("http://iesl.mit.edu/MathService/200
	4/05/09/Subtract",RequestNamespace="http://iesl.mit.edu/MathServi
	ce/2004/05/09",
	ResponseNamespace="http://iesl.mit.edu/ MathService/2004/05/09")]
	public int Subtract(int a, int b)
	{
		return a-b;
	}

	[WebMethod]
	[SoapDocumentMethodAttribute("http://iesl.mit.edu/MathService/200
	4/06/03/Multiply",RequestNamespace="http://iesl.mit.edu/MathServi
	ce/2004/06/03",
	ResponseNamespace="http://iesl.mit.edu/ MathService/2004/06/03")]
	public int Multiply(int a, int b)
	{
		return a*b;
	}
}
```

Figure 33 - .NET Web Service showing the utilization of SoapAction and Namespaces

6.6.2 Technique – Redirection of Requests to earlier versions

This technique would allow maintaining different .asmx endpoints for different versions. This option does not maintain namespace compatibility with existing clients [24]. A way to do this is to create a new instance of web service, add methods in the new XML namespace, then redirect requests of old operations in the new namespace to old operations in the base class. Methods of the older version are available through inheritance.

```
[WebServiceAttribute(Name="BasicMathOperations",
Namespace="http://iesl.mit.edu/MathService/2004/06/12")]
      public class Mathv3 : System.Web.Services.WebService
      {
            [WebMethod]
            public int Add(int a, int b)
            {
                  // Instantiate the orginal class and call the function
                  Mathv1 v1= new Mathv1();
                  return v1.Add(a,b);
            }

            [WebMethod]
            public int Subtract(int a, int b)
            {
                  // Instantiate the orginal class and call the function
                  Mathv1 v1= new Mathv1();
                  return v1.Subtract(a,b);
            }

            [WebMethod]
            public int Multiply(int a, int b)
            {
                  return a*b;
            }
      }
```

Figure 34 - .NET Web Service Sample showing how to derive new instances of service using the older versions

Issues associated with this technique:

- There exists a conflict between message requests for operations common in the new and the old version. Messages qualified by different namespaces are being served by the same old class definition.
- The already existing clients are not compatible with the new version because they use distinct namespaces.
- Both the versions have to be maintained to run the existing and the new clients.
- It would be required that the old clients upgrade to the new service and after a period of time the old service is discontinued.

6.6.3 Technique: Customization - Splitting the WSDL

Customization of WSDL by splitting it into components also enables versioning and reusability of components. The ideal pattern for development of Web Services is to:

- Define the data types in XSD and the remainder of the WSDL elements before coding the underlying concrete implementation.
- Each component is given its own targetNamespace with a common root URI. The common root enables to link the items together and at the same time the further extension is used to separate the various components.

Instead of using the automatically generated WSDL prototype directly, we could modify the generated WSDL documents in an organized and modular way so that the information could be reused and additional functionality to the service could be implemented [23, 24, and 25]. The following table indicates the namespace of the components of the WSDL which were created for the sample Web Service being discussed in the chapter.

Information Type	**Associated URI**
Base Data Types	*http://iesl.mit.edu/MathService/2004/05/09/messages*
Message Data Types + portTypes + Binding	*http://iesl.mit.edu/MathService/2004/05/09/definitions*
Service Implementation	*http://iesl.mit.edu/MathService/2004/05/09/*

Table 2 - WSDL components and their corresponding Namespaces

In the above sample, the common root URI is *'http://iesl.mit.edu/MathService/2004/05/09/'*. Following are the advantages of such a modularization process:

- One endpoint could be used to implement multiple bindings (pointing to distinct portTypes) via collection of ports.
- This technique enables the reusability feature for the construction of a WSDL by using a combination of components.

The following sample shows how multiple WSDL definitions are deployed at one endpoint.

```
<?xml version="1.0" encoding="utf-8"?>

<definitions xmlns:s1="http://iesl.mit.edu/MathService/2004/05/09/definitions"
xmlns:s2="http://iesl.mit.edu/MathService/2004/06/03/definitions"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s0="http://iesl.mit.edu/MathService/2004/06/30"
targetNamespace="http://iesl.mit.edu/MathService/2004/06/30"
xmlns="http://schemas.xmlsoap.org/wsdl/">

 <import namespace="http://iesl.mit.edu/MathService/2004/05/09/definitions"
location="http://localhost/MathService/Mathv1.wsdl" />

 <import namespace="http://iesl.mit.edu/MathService/2004/06/03/definitions"
location="http://localhost/MathService/Mathv2.wsdl" />

 <service name="BasicMathOperations">

   <port name="BasicMathOperationsSoapV1" binding="s1:BasicMathOperationsV1">
    <soap:address location="http://localhost/MathService/Mathv2.asmx" />
   </port>
  <port name="BasicMathOperationsSoapV2" binding="s2:BasicMathOperationsV2">
    <soap:address location="http://localhost/MathService/Mathv2.asmx" />
  </port>
```

Figure 35 - Deployment of multiple WSDL definitions at one endpoint

Version 1 of the service is modularized in the following manner:

- The basic data types are stored in a XSD file with the following namespace- *http://iesl.mit.edu/MathService/2004/05/09/messages*
- The message types, portTypes and binding information is stored in another WSDL file namely, Mathv1.wsdl, in a different namespace -

http://iesl.mit.edu/MathService/2004/05/09/definitions. Also, it imports the schema and the corresponding namespace mentioned in the above bullet.

- Similarly is done for *Version 2*, in which the WSDL containing the abstract definitions imports the schema definition of version 1 for existing operations. This indicates reusability of schema component described for *Version 1* and the new schema for *Version 2* is further extended by adding data types for the newly added operations in a new namespace.
- The main WSDL interface contains the *service* element and distinct *ports* are defined for each of the *bindings* defined in the intermediate WSDL files. The service endpoint of all the *ports* could be the same or different as per the requirements.

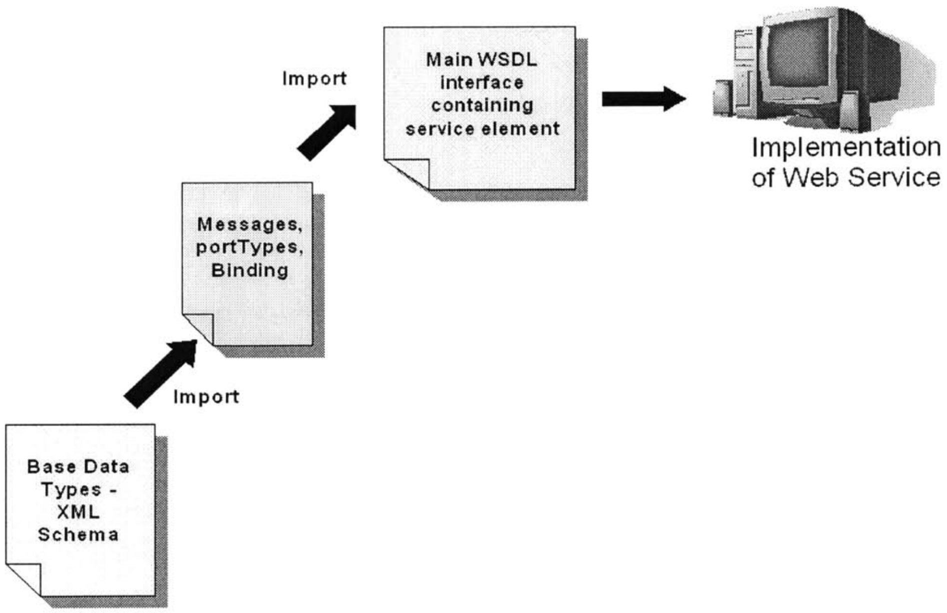

Base Data Types - XML Schema
Import
Messages, portTypes, Binding
Import
Main WSDL interface containing service element
Implementation of Web Service

Figure 36 – Order in which the WSDL components are consumed to deploy a service interface

The advantages of this strategy are:

- Multiple ports could be bound to maintain backward compatibility and version implementation.

- This mechanism has an edge over the above mentioned technique in Section 5.6.1 because it explicitly shows two bindings are implemented by the service which was not evident in the that technique.

6.7 Changing Method Signature

The *signature* of the method comprises of its name, the data types and the sequence of parameters. They are required to be unique for a given class definition [23, 14]. Change of method signature would be required in scenarios where the existing versions of the method are to remain operational and new methods with the same name are to be created to provide the same functionality for another set of parameters. This is achieved by embedding unique message names to each of the operations having the same method name. So it is possible to support *method overloading* in the service description.

```
namespace MathService
{

	[WebServiceAttribute(Name="BasicMathOperations",
Namespace="http://iesl.mit.edu/MathService/2004/06/03")]
	public class Mathv5 : System.Web.Services.WebService
	{
		public Mathv5()
		{
			InitializeComponent();
		}

		[WebMethod(MessageName="Addv1")]

		public int Add(int a, int b)
		{
			return a+b;
		}

		[WebMethod (MessageName="Addv2")]

		public int Add(int a, int b, int c)
		{
			return a+b+c;
		}
	}
}
```

Figure 37 - Web Service Sample showing how to change signature of an operation

The *WebMethod* attribute for each operation has a *MessageName* property which could be leveraged to give unique message names to operations with the same name.

6.8 Updating Data Model

There are scenarios, in which the data types of the messages exchanged within existing operations change with time [23]. This could be in the form of change in the *User Defined Types* or addition of more data types exchanged within an operation. To achieve the same, the following strategies could be leveraged:

- *Open Content Model* as discussed in Chapter 5.
- *Extensibility Model* as discussed in Chapter 5.

6.9 Technique – Insertion of an Intermediate Routing Service

This technique could be used to manage all the changes within a service, whether backwards compatible or non-backwards compatible, giving rise to versions of the service or the creation of new services in cases of non backward compatible changes. But all of them could be managed behind one service endpoint. This is achieved by introducing a routing service at the common service endpoint which would further route the individual requests to their actual implementation codes. This would become clearer with *Figure 34* [28, 29].

When a service method is requested on the server, the request is intercepted by the routing service which acts as a gateway for all the incoming requests to the various versions of that service. The *targetNamespace* is extracted from the *SOAPAction* and the routing service looks for the corresponding implementation code on the server and directs the requests to that location. The mapping between the namespace and the corresponding code implementation is stored either in a database or an XML file.

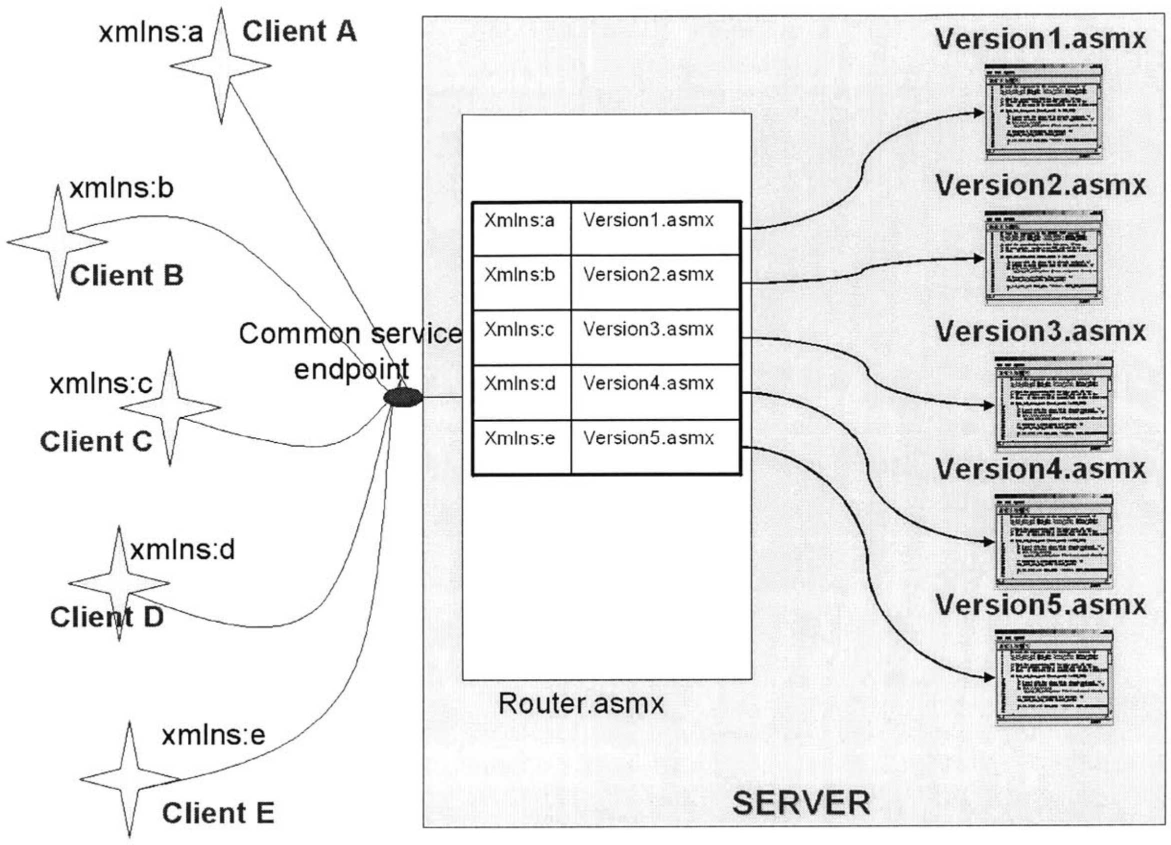


Figure 38 - Maintenance of all the versions using a Routing Service

From the WSDL perspective of such a service, only the interface description of the latest version of the service will be available which could be determined by the extracting the date stamp from the targetNamespace of the latest version. Some logic at the code side is required to be implemented to make such a framework work. Following are the assumptions on which the model stands:

- A strict namespace convention is used for all the versions of the service.
- The namespace contains the version information in the form of date stamp.

Following are the advantages of this model:

- It enables in keeping track with all the changes a service undergoes during its lifetime.
- Interactions with existing clients are maintained and new clients can subscribe to the latest version of the service.

Following are the disadvantages of this model:

- This mechanism of versioning does not scale well as it requires management and deployment of a huge number of versions.
- A lot of overhead required in terms of resources and maintenance of such a model in which one has to deal with all the versions.
- The delay to process a Web Service request is enhanced as one more redirection level is introduced in the model to route the request to the concerned implementation from the common service endpoint.

All these techniques provide some versioning logic which is deployable in a small scale scenario. However, the drawbacks really become conspicuous when tried to scale up. There has to be an inbuilt support in the Web Service definition that will convey the version information.

Chapter 7

Summary

Web Service is a useful technology available today for exposing functionality in the form of services to a heterogeneous network of client components. The baseline standards are simple, easy to implement and adhere to the service oriented architecture. Specifications such as SOAP, WSDL, UDDI are the core foundation of the Web Services Model and addresses the interaction between components in a consistent, ubiquitous and internet centric way. However, the success of early experiments and prototypes has created an illusion of full functionality and an assumption that early success can be extrapolated to long term viability.

These critical foundation elements when used on their own are largely insufficient for real-world e-business solutions. Features like transaction management, security, versioning, trust etc must be added to the Web Service software stack to make the model usable beyond simple messaging and more applicable to enterprise and global class contexts. Specifications for the base standards are available; however, higher level specifications are still being worked upon. Due to the lack of these specifications, the software vendors are coming up with their own proprietary solutions resulting in increased complexity and defeating the purpose of an interoperable model. So it is of prime concern to agree upon the standard high level specifications and API's as well.

Versioning is one such issue which has to be considered for optimizing the maintenance of such services and establishing integrity between the loosely coupled components. The entire architecture would not make sense if there is no reliability between the services. So, once a service is published it has to make sure that it serves the existing clients all the time and does not break without any prior notification.

Two levels of versioning are identified in this programming model- Schema level versioning and Interface level versioning. Versioning at schema level is important because the data being exchanged over messages change with time. Extensibility models, open content models, creation of custom tags and using distinct namespaces are some of the techniques to implement versioning logic to the schema. However, all of them have their own pros and cons and there is no ideal way to do it as yet. Versioning at interface level should be backward compatible because the services should evolve in such a way that they maintain the existing clients at the same time are able to add new operations and modify the existing operations. There are scenarios were multiple versions are required to be maintained, for instance, customization of the service as per the needs of the customer would require maintenance of multiple versions. This thesis, in the absence of specifications, discusses some proprietary solutions to achieve these goals. The proposed techniques are ways of active versioning, meaning the versioning logic has been forcefully inserted in the infrastructure and are not inherently a part of it. However, none of these techniques are scalable to a large extent.

It could be concluded that, before the technology matures in terms of implementation and stability, it is required that the underlying business model, best practices, specifications are created and stabilize over the period of time.

Bibliography

[1] Bassam Tabbara. *Building Manageable Apps: Architectural Overview,* Microsoft Professional Developers Conference, October 26-30th, 2003.

[2] David Orchard, Norman Walsh. *Versioning XML Languages*, 2003. http://www.w3.org/2001/tag/doc/versioning

[3] Brain Tag. *Developing Service Oriented Architectures*, 2003. http://msdn.microsoft.com/architecture/application/default.aspx?pull=/library/en-us/dnvsent/html/FoodMovers3.asp

[4] Robert Chartier. *Application Architecture: An N-Tier Approach –Part 1*

http://www.15seconds.com/issue/011023.htm

[5] Don Box. *"Indigo": Services and the Future of Distributed Applications*, Microsoft Professional Developers Conference, October 26-30th, 2003.

[6] *Web Services Moving Beyond the Hype*, March 2002. http://www.internetnews.com/ent-news/article.php/7_990981

[7] Mark Driver. *Navigate the Web Services Hype Cycle*, September 2002.

http://www.ftponline.com/wss/2002_09/magazine/columns/trends/

[8] Stephen Potts, Mike Kopack. *Disadvantages and Pitfalls of Web Services*, 2003.

http://www.informit.com/articles/article.asp?p=31729

[9] Keith Ballinger. *.NET Web Services –Architecture and Implementation*, 2003

[10] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju. *Web Services-Concepts, Architecture and Applications,* 2004

[11] Eric Newcomer. *Understanding Web Services-XML, WSDL, SOAP, and UDDI*, 2002

[12] Doug Kaye. *Loosely Coupled- The Missing Pieces of Web Services*, 2003

[13] Don Box. *Essential COM*, 1998

[14] Foggon, Maharry, Ullman, Watson. *Programming .NET XML Web Services*, 2003

[15] *XML Schema Versioning*. http://www.xfront.com/Versioning.pdf

[16] David Orchard, *Providing Compatible Schema Evolution*, January 2004.
http://www.pacificspirit.com/Authoring/Compatibility/ProvidingCompatibleSchemaEv
olution.html

[17] Dave Orchard, *Examining wildcards for versioning*, January 2004

http://www.pacificspirit.com/Authoring/Compatibility/ExaminingElementWildcardSibl
ings.html

[18] David Orchard, *Versioning XML Vocabularies*, December 2003.

http://www.xml.com/pub/a/2003/12/03/versioning.html

[19] Henry S. Thompson, *Versioning made easy with W3C XML Schema and Pipelines*,
April 2004. http://www.markuptechnology.com/XMLEu2004/

[20] *Creating Extensible Content Models*

http://www.xfront.com/ExtensibleContentModels.html

[21] David Orchard, *Web Services Pitfalls*, February 2003.

http://webservices.xml.com/pub/a/ws/2002/02/06/webservices.html

[22] Doug Purdy*, Loosely coupling and Serialization Patterns: the Holy grail of service design*, TechEd 2003 (Microsoft Corporation)

[23] Scott Seely, *Versioning Options*, October 2002

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnservice/html/service10152002.asp

[24] Scott Seeley, *Evolving an Interface*, April 2002

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnservice/html/service04032002.asp

[25] Scott Seely, *Splitting up WSDL: The Importance of targetNamespace,* August 2002

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnservice/html/service10152002.asp

[26] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1*, March 2001. http://www.w3.org/TR/wsdl

[27] David Booth, Philippe Le Hegaret, Canyang Kevin Liu. *Web Services Description (WSDL) Version 2.0 Part 0: Primer*, November 2003

http://dev.w3.org/cvsweb/~checkout~/2002/ws/desc/wsdl20/wsdl20-primer.html

[28] Chris Peltz Anjali Anagol-Subbarao. *Design Strategies for Web Services Versioning*, April 2004. http://sys-con.com/story/?storyid=44356&DE=1

[29] Kyle Brown, Michael Ellis. *Best Practices for Web Services Versioning*, January 2004. http://www-106.ibm.com/developerworks/webservices/library/ws-version/

[30] Romin Irani, *Versioning of Web Services*, August 2001. http://www.webservicesarchitect.com/content/articles/irani04print.asp

[31] Scott Mitchel. *An Extensive examination of Web Services – Part 3.* http://aspnet.4guysfromrolla.com/articles/110503-1.aspx

[32] *EAI and Web Services.* http://www.webservicesarchitect.com/content/articles/samtani01.asp

[33] Pat Helland. *Envisioning the Service Oriented Architecture*, Microsoft Professional Developers Conference, October 26-30th, 2003.