Bringing Web Services to Life:
A Prototype E-Mail Web Service

MIT VI-A Program – Summer Assignment 2001
Submitted to fulfill the requirements of 6.929:Undergraduate Project Presentation

Buddhika Kottahachchi
buddhika@mit.edu
MIT VI-A Intern / Lotus K-Station Development Group

VI-A Advisor
Prof. Tomas Lozano-Perez

Manager
Elena Karra

Mentors
Koranteng Ofosu-Amaah
Margaret O'Connell
Thangaraj Veerappan

Abstract

Web Services are an evolving technology that has gained widespread interest in the industry. However, as with all evolving technologies, further improvements must be made to make it suitable for widespread implementation. This project set out to identify such areas of desired improvement and build a prototype that addressed them. Authentication, Privacy, Execution Speed and Ease-of-use were identified as some areas of desired improvement, and a prototype addressing these issues was built. Furthermore, areas for future investigation were also identified.

Contents

1 Introduction

The emergence of the Internet and its incorporation into our lives has caused many changes in our world. The Internet and the entire concept of networking give us a higher level of interconnectivity. People keep creating new means to exploit this interconnectivity. The latest in this process has been the emergence of XML-based Web Services, which are touted as the "next stage of evolution for e-business"[3].

While industry giants like Microsoft, IBM, Sun and even HP have rallied around this technology, it is still an emerging one and therefore has not seen much application level use.

This project was motivated by the Lotus K-Station Development Project's desire to explore the possibility of using Web Services in their next iteration. It was undertaken as an intern project during the summer of 2001 with the Lotus K-Station Development Group. The project focused on identifying and examining key areas of concern with the use of web services and building a prototype that would address some of these concerns.

2 Background

2.1 Overview

Web services as described in [3] are "self contained, modular applications that can be described, published, located, and invoked over a network". Such web services also have the following properties:

- Performs a specific task or a set of tasks
- Interoperates with other web services
- Can be remotely invoked by applications
- No knowledge of the underlying implementation is required to benefit from a web service
- Applications can be dynamically changed by changing the underlying services used

Currently there are two significant manifestations of the vision described in [3]. These are Microsoft's C# based .NET [1] framework and the IBM/Apache/Open Source Community's [2] Java-based framework. Both of these adhere to a strict set of standards, and provide a surprising degree of interoperability.

As explained in [5], both these implementations describe three roles in relation to the Web Services Architecture. They are:

Service Provider - *The entity that decides to expose a software artifact as a web service*.

Must *publish* his web service with a Service Broker.

Service Requester - *An entity that wishes to make use of a pre-published service.*

Must contact a Service Broker and make a *find* query. If the query is successful the Service Broker will return the information required to *bind* with the Service Provider and invoke the required service.

Service Broker - *The intermediary*

Maintains a database of published web services and their public contracts. Makes this information available to Service Requesters when needed.

2.2 Current Web Services Model

In the interests of interoperability and standardization, both the Microsoft and IBM efforts have converged on the same model for web services. This model is shown in Figure 1.

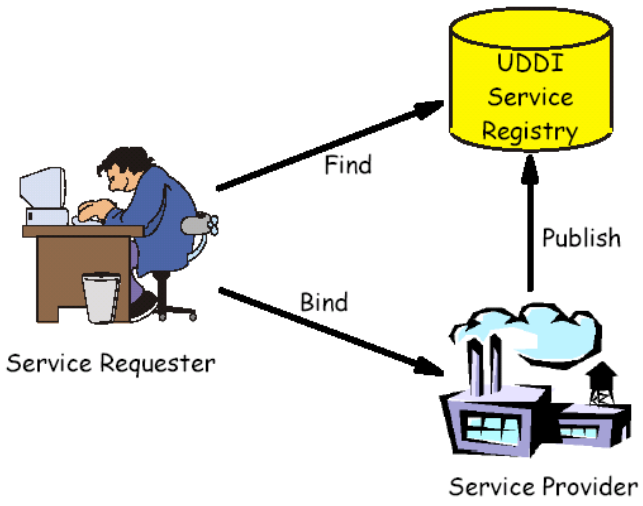

Figure 1: Web Services Model (Source [4])

As can be seen, the role of the Service Broker has been taken by an entity known as a UDDI Service Registry. Also, all the major web services platform implementation efforts have converged on a single set of standards. These standards and their roles in the web services model shown in Figure 1 are described below.

2.2.1 SOAP (Simple Object Access Protocol) [5] – *The language web services speak*

An XML-based protocol that allows simple request-response type messages to be formulated in a standard manner. This allows request/response interactions to take place between different entities. In

Figure 1, SOAP is used for all communication between the Service Requester, Service Provider and Service Broker. This includes service invocation and response. Since it is an XML-based protocol SOAP messages can be sent over any transport protocol. However, because of its ability to get through firewalls and its ease of use, HTTP is the current de-facto standard. Also, since both XML and HTTP are platform independent it follows that SOAP is platform independent

2.2.2 WSDL (Web Services Description Language) [6] – *The language of a web service's public contract*
An XML schema that allows the description of a web service's public contract in a standardized manner. A web service's public contract written up in WSDL is used to publish the web service in an UDDI registry. When the UDDI registry responds positively to a *find* query, it is this contract that is sent to the Service Requester. A web service's public contract describes its methods, argument types, and return values, thereby allowing the Service Requester to invoke it in the desired manner.

2.2.3 UDDI (Universal Description, Discovery and Integration) [7] – *The Web Service directory interface*
This specification defines how a service directory or registry should function. A service registry can be thought of as the "Yellow Pages" of Web Services and one can query it (in the manner specified in UDDI) to find specific services that are required by a Service Requester. All interactions with the UDDI registry take place over SOAP messages. UDDI is still an evolving standard and does not play a significant role in most web service applications today.

A web service, at its most basic form is a fragment of code that performs a task that many others may also perform on a regular basis. This task can range from a simple addition operation to a piece of business logic that a company performs in its daily routine. In this model, when an individual or an organization identifies a code artifact it desires to expose as a web service these steps must be followed:

1. Gain access to a SOAP (Web Services) Server
2. Install the code artifact on the SOAP Server (process varies from server to server)
3. Create a WSDL file to describe the web service and its public contract
4. Publish the web service by publishing the WSDL file to a UDDI registry

A web service made available in the aforementioned manner can be invoked by anyone having access to its WSDL file. Therefore, a service requester who requires access to a particular service will follow these steps:

1. Query a known UDDI Service Registry
2. If a positive result is returned, parse the WSDL file describing the service that was returned
3. Invoke the service using the knowledge gained in step 2.
4. Parse the return values and use them as desired.

2.3 Current Reality

While section 2.2 describes the currently accepted model for Web Services, it does not necessarily depict an accurate picture of current Web Service implementations. As mentioned above, UDDI is still an evolving standard and as such there are no "industry strength" implementations. Therefore, current applications that utilize web services bypass the UDDI Service Registry and sometimes even the Service Broker step. Therefore, current implementations assume that Service Requesters have a priori knowledge of Service Providers. This model is shown in Figure 2.

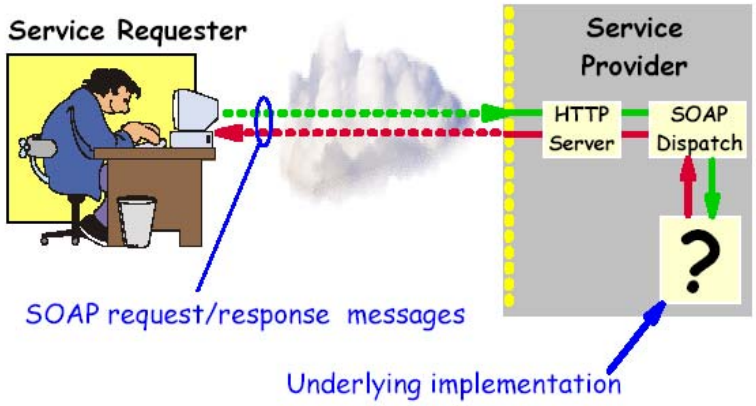

Figure 2: Web Services Invocation (Source [4])

We are interested in developing "industrial strength" web services using a Java-based platform for near-term use in enterprise applications. Therefore, this is the model of interest to us.

3 Problem Description

While web services even without the service registry component have been well received, in general, they are still plagued by some perceived "weaknesses" that make them unattractive for widespread adoption. The primary "weaknesses" identified via this project were the lack of security related to privacy and authentication, mediocre execution speed performance, and difficulty in end-user use.

3.1 Security

All network communication related to web services is currently done over HTTP. HTTP is an open transport protocol. Therefore, this means that when service requests, method invocations, or service responses occur over the Internet, they are visible to anyone who wishes to see them. Since enterprise applications involve transactions that may contain sensitive material, implementers are deterred from using web services in such applications.

Furthermore, some of these transactions that take place over web services require a high level of trust to exist between the parties involved. For example, it was pointed out above that business logic could be exposed as web services. This in turn would lead to business transactions being conducted using web services, and that would require a great deal of trust between the transaction participants. For instance, inventory and payment details can be considered highly sensitive material. At the time of this project, the SOAP Security Extensions [8] had just been released. However, no widely available implementations existed, which meant that it was an unproven solution to the existing problem. The recommended mode of authentication was to use SSL and certificates that leveraged existing proven technologies. This, however, required a high overhead of server maintenance since a record of each authorized user would have to exist on the server. Therefore, this workaround was not a popular one.

3.2 Execution Speed

Enterprise applications traditionally require a high transaction throughput capability. That is, each request must be handled in a timely manner and since requests tend to come almost continuously, backlogs that keep growing are undesirable. In preliminary testing it was found that the response time of our test web services (examples that came with the platforms we tested) could be in the order of a few seconds. This does not seem to scale well, and reveals the need for a workaround that would give us acceptable performance levels.

3.3 Ease of Use

This is another area found wanting. Under the model that existed at the time of the project, to develop web services one had to have an understanding of the Java-based implementation of the SOAP protocol. This meant that end users had to know how to construct SOAP messages that invoked the web service and to parse SOAP responses (using the Java API's provided). Furthermore, if additional layers like SSL were brought in to enhance security, this involved having the end user understand the Java Security Extensions as well. We found this to be an unacceptable burden on the end user. Web services were being portrayed as a disruptive technology that would replace CORBA [9] and Java RMI etc., and as such it would be ideal if end users did not need any new knowledge to benefit from them.

3.4 The Prototype – An E-Mail Web Service

Having identified these three areas of key interest, a suitable web service needed to be prototyped. This prototype was required to address the concerns raised above, and yet be relatively easy to implement. Therefore, it was decided to implement an E-Mail web service. This was mainly due to the fact that the JavaMail API and its reference implementations already existed, which allowed the focus to be on the

three identified issues rather than on lower level E-mail functionality implementation details. E-Mail also had the following other desirable properties:

- Privacy, Trust and Authentication were important in E-mail applications
- Reasonable execution speed was expected
- Potential applications were widespread, and therefore ease of use was important

which made it suitable for the needs of this project.

4 Development Environment

4.1 Hardware

Initially the Hardware used was the developer's own IBM workstation that had a Pentium III 550 Mhz Processor, 256MB of RAM and a 20GB Hard Disk. For preliminary testing this machine served as both the server and the client of the prototype, in addition to serving as the Development machine.

By the time the final version was deployed, access to a server designated to serve up prototype Web Services was gained. This server had 1 GB of RAM and performed much better as a server. Also, having a remote server helped fine tune the performance optimizations that went into the prototype.

4.2 Software

All development coding was done using the IBM Visual Age 3.5.3 IDE (JAVA SDK 1.3). There were several iterations of the server software configuration. Initially, the server ran Apache Tomcat 3.2 and had Apache SOAP 2.2[9] installed as an application on it. A second iteration ran IBM Websphere Application Server 3.5.4 with Apache SOAP 2.2 installed as an application. The final iteration consisted of a pre-release version of IBM Websphere Application Server 4.0[10] – this had built in support for SOAP and the SOAP security extensions. Finally, the JAVAMail 1.2[11] API and its reference implementations were used to realize low level E-Mail functionality.

5 Design & Implementation

The Prototype was designed such that it would be packaged in two separate JAR files for server and client side installation. All relevant code for each side of the Client-Server relationship was contained in the corresponding package. However, these JAR files were also dependent on existing external libraries (also packaged as JARs) that provided support for SOAP, SOAP-SEC, SSL, SMTP, POP3, IMAP4 and XML. The additional libraries are a "one-time install" required on any server that runs Web Services designed using the practices described in this paper. Including all the required packages in one install package can

circumvent the hassle of installing many packages on the client side. However, this could lead to redundant installation when installing more than one Web Service Client on a single machine.

The server side package includes the code required for the underlying implementations of all exposed functionality. Therefore, this is the piece of code that interacts with the mail servers we hit. However, since we plug this into an existing SOAP application server no additional code is required to take advantages of features like SSL since the server handles them.

The client side package is used by end user programmers to interface with the service. The client side package contains a proxy class that handles the task of building and invoking SOAP method calls.

5.1 Functionality Exposed

This prototype exposes six pre-implemented methods designed to make E-Mail related application development easier. These are (detailed JavaDocs in Appendix):

5.1.1 login()

This method is only useful when the service is being invoked with session support (discussed below). Essentially, it initiates a session between the client, the server, and the Mail server. It takes in *username, password, mailhost, hosttype* and *smtphost* as parameters.

5.1.2 getInbox()

Returns the header information for a specified range of messages in a specified user's Inbox. When method is invoked with session support, takes in the integer parameters *start* and *end* that describe the range of messages we are interested in. When session support is disabled *username, password, mailhost,* and *hosttype* are also required.

5.1.3 getMessage()

Returns the contents of a specified message from a user's Inbox. If session support is enabled, only the *message number* is required as a parameter. However, as above, *username, password, mailhost,* and *hosttype* are all required when session support is disabled.

5.1.4 deleteMessage()

Removes a specified message from the user's Inbox. Parameters and the conditions that apply are the same as in getMessage.

5.1.5 sendMessage()

Builds and delivers a message to an SMTP server for routing to the intended destination. Takes in *from*, *to*, *cc*, *bcc*, *date*, *subject*, *message body*, and *smtpHost* as parameters.

5.1.6 logout()

This method is also only applicable when session support is enabled. Its function is to terminate the sessions initiated by invoking the login() method.

These methods and their descriptions are for the underlying implementations on the server side. We decided to make the proxy classes as similar to them as possible and with the exception of all methods requiring an *endpoint* parameter in session disabled mode and an *endpoint* parameter only in the login method for session enabled mode, the method descriptions are essentially the same. However, the client-side proxy class simply creates a SOAP call with the desired parameters and invokes it on the server side – ie. it does not implement any underlying functionality on its own.

An *endpoint*, as referred to above is simply a URL to which SOAP calls can be directed. These are usually implemented in the form of a Java servlet as was the case in this implementation.

Once the core functionality described above was implemented, attention was moved to the three key issues we desired to address.

5.2 Security

Two components of security were identified as being of high priority for this project: privacy and authentication. While SOAP had no explicit support for either, the common approach was to exploit the fact that method invocation and responses were carried over HTTP and use SSL to achieve both of these. However, this approach has several shortcomings as described in Section 3.1. This implementation used the following approach:

The implementation of SOAP Security extensions available on WebSphere Application Server 4.0 was exploited for our purpose. These extensions described how XML Digital Signatures could be incorporated in SOAP messages.

A Public Key Infrastructure was used where each client would sign every message sent to the server (ie. method invocation) and each server would do the same for all messages sent to the client (ie. responses).

This means that if we trust the Public Key that signs the messages we can associate each message with a unique entity. However, this recreates the problem of maintaining costly user databases on the server side – which should preferably be avoided.

Therefore, we propose an alternative scheme where we use a Public Key Infrastructure [12] with a Certifying Authority. For example, the Service Provider can be the Certifying Authority and Clients could be issued certificates signed by the Certifying Authority when they purchase rights to use a particular service. These certificates are standard X.509 [13] certificates with a built in value for length of validity. Therefore, service providers can sell usage rights to clients for distinct durations; this is easily enforceable since the certificate expires automatically. Furthermore, the SOAP server only needs to know information about the Certifying Authority and not about the individual clients or service consumers. All our trust is placed on a Certifying Authority we control, and provided appropriate steps are taken to prevent an adversary gaining control of the Certifying Authority, this model is believed to be satisfactory for our stated requirements concerning authentication.

With regards to the issue of privacy, it was decided to stay with SSL [14] as the mode of implementation. This was preferred since it is already in wide use and supports an extensive cipher suite. However, an important distinction to make is that we only use SSL to establish a secure channel for communication and as such run it with only server authentication enabled.

5.3 Performance

Performance, as described in Section 3.2, is a troubling problem we have to deal with in the existing infrastructure of Web Services (especially in Java based platforms). Reasons for this slowness, stem from various sources ranging from extensive (virtual) memory usage as applications become memory intensive to the disappointing execution speed of existing XML parsers. In such a context little could be done to improve performance. As mentioned above in Section 4.1, moving to better hardware for the server helped fine tune performance; however there was no significant improvement (ie. by more than or equal to an order of magnitude) in performance.

However, it was noticed that E-Mail functionality had a certain property that could be exploited to increase performance noticeably in such applications. The basic idea stems from the fact that E-Mail functionality (especially if you take a WebMail application into consideration) inherently requires the concept of a session. For example, a user logs into his account, manipulates his Inbox and then logs out. Many of the Web Service implementations existing at the time did not address functionality issues that desired this, and an implementation that used that model would have to re-authenticate itself with both the SOAP server and mail server for every manipulation done on a user's Inbox. This introduces considerable overhead and it was observed that reducing this overhead could yield considerable performance gains.

Upon investigation it was found that Apache SOAP 2.2 had a concept of sessions built into it. It was decided to exploit this functionality for our benefit. As mentioned above, our implementation allowed sessions to be created by invoking the login() method and for sessions to be terminated using the logout() function. Sessions were maintained by keeping a single Call (part of the Apache SOAP 2.2 API) object alive in the proxy class for the duration of the session we created.

While this provided a noticeable performance gain in preliminary testing, it was believed that certain functions exposed by this prototype may be required for invocation individually in certain applications. Therefore, the final implementation included support for both session enabled and session disabled method invocation. Furthermore, it is useful to note that this does not provide a universally applicable solution to the performance problem. However, this is a proposal that can be used to improve the situation under circumstances where session management is useful. Examples of such applications are E-Mail, Instant Messaging etc.

5.4 Ease of Use

As stated above in Section 3.3, ease of use is critical for the success of widespread adoption of this technology. Therefore, it was another area that was focused on. The existing model for making this possible at the time of the project was to have, for each service, a WSDL file which any client could process through any of a range of WSDL to Java Proxy Class generators available in the market. However, while we felt this in itself did not provide sufficient ease of use and forced unnecessary complexity onto the end user, it was also observed that many of the WSDL to Java Proxy Class generators were lacking in most commonly used Java object types. Furthermore, they had no means of supporting the security additions made as part of this project nor some of the object types used.

Upon further investigation, it was realized that the concept of WSDL-based proxy generators was valuable at some point in the future, yet they were too primitive for our needs at present. Therefore, it was decided to implement the proxy classes ourselves and include that as part of our deliverable to the client/consumer. Ideally, it was desired that these proxy classes would conform to a set of evolving standards, which could eventually be implemented in the form of a more functional WSDL based proxy generator.

Therefore, all the code required to build and invoke the SOAP calls was included in the proxy class. This step essentially replaced the function of existing WSDL based proxy generators. Furthermore, the proxy classes built included the code required to support the instantiation and maintenance of an SSL connection. In addition to this, code required to use the Public Key Infrastructure and the SOAP Security Extensions were included. This resulted in a complete “bundle” as the final deliverable. By selecting the appropriate functions in the API released for a service, it can be invoked in the manner desired by the end

user. Furthermore, the end user is no longer required to have a detailed understanding of SOAP, SSL, or PKI. All he/she needs to know is how to invoke a simple Java method; a Java method is all that our end-user sees!

To make things even cleaner, it was required that all proxies (and therefore the Web Service itself) returned a Hashtable object as its return value. This Hashtable would have, at least, mappings for "exitStatus" (associated with a Boolean object true or false) and "exitMessage" which would be set to "Success" if method execution was successful and return a suitable error message if it wasn't, all of which is specified in the API.

This makes adoption highly non-disruptive since no demands are made on additional knowledge by the end user. Furthermore, by using this approach it was possible to enforce good practices in implementations of the security components thereby reducing the possibility of vulnerability due to bad implementations. Given all this, it was felt that this was a suitable solution for the problem identified. However, the standardization process needs a few more iterations before we can go back to the model of generating proxies from a WSDL file.

6 Analysis

Once this prototype web service was implemented, a platform to demonstrate its capabilities was desired. Initially, subsets of its functionality were used as parts of larger demos built for internal presentations by a sub-group of the K-Station Development team. Not completely satisfied with that, it was decided to implement a web-based e-mail client that could be used to test the full range of functionality offered by the prototype.

Therefore, a JSP-based "web mail" client was built and deployed on the author's workstation using Apache Tomcat as the servlet engine. This web-based e-mail client made calls to the Web Services server on which the actual service was deployed and constructed HTML code based on the return values. A web based mail client that utilized all the functionality available in the prototype was implemented providing users with a rudimentary interface through which to access their e-mail. Any e-mail account supporting a POP3 or IMAP4 interface could be accessed using this client. However, due to firewall restrictions, initial testing was done using a mail server installed on another intern's machine. All preliminary testing results referred to above were gathered using this client.

Furthermore, this client was used when the entire project was presented at the Lotus Research Lunchtime Luminaries seminar in mid August 2001 – where it received a favorable response. The project was successful in demonstrating several key points. Firstly, it was shown that security issues in Web Services could be effectively addressed using existing frameworks. Secondly, it was shown that significant performance gains could be achieved by observing the properties of the intended application. Finally, it

was shown that Web Services constructed and deployed in the manner done in this project reduces the end-user integration task to a simple Java method call.

7 Future Research and Conclusions

This project has been able to make more concrete the potential of Web Services by creating a working prototype that demonstrates the capabilities of the technology. However, a noticeable piece missing from the big picture is the lack of an automated mechanism to link clients/consumers with desired services. While UDDI is evolving, perhaps a better alternative is the recently released WS-Inspection standard [15], which naturally extends the "current reality" described in Section 2.3. There is definite potential for research in this space. The potential value of breakthroughs in this area is apparent. For example, the company that won the MIT 50K competition in 2000 – Centrata [16] – seems to be involved in this area. Their business plan involves dynamically aggregating CPU cycles based on demand and selling these cycles to those who are willing to pay for them while sharing profits with the individuals who make their free CPU cycles available (idea borrows heavily from the SETI@Home project [17]). Even though they are currently in "stealth mode," the limited information available on their web site is sufficient to indicate that their efforts revolve around some form of XML-based web services.

While working on this project it became apparent that such an application could be built using the technology components we currently have access to and a new structure for serving as the broker. It was felt that the value proposition of Web Services in Enterprise Applications could be significantly improved by constructing a broker structure that could handle load balancing and failure recovery – these are both missing from existing proposals for broker structure. In conjunction with the Centrata model, this could be extended such that institutions could be provided with Enterprise Applications that actually utilize unused CPU cycles within their organizations. This could have significant hardware cost benefits in the long run for these institutions. Therefore, the area of new Broker structures is one of interest and is one in which the author would like to pursue further research efforts.

Furthermore, for Web Services to become "real," another area for research focus will be XML parsing. As stated in [18], none of the existing commercial XML parsers perform sufficiently to enable Web Services to provide performance levels comparable to CORBA. While efforts like Electric XML[19] from The MindElectric and the XML Pull Parser [20] from the SOAP team in the Extreme Computing Lab at Indiana University have shown promising initial results, this is still an area that requires further research in. An interesting approach might be to write a parser that is constructed to simply parse SOAP messages and nothing else. However the merits of this approach need to be further investigated.

Another area of research identified in developing the prototype was in the area of security. This applies primarily to the desire to support usage based billing of Web Services. The model proposed in this project

only supports license-based billing (i.e. certificates which are valid for a specified period of time). However, the likes of Microsoft have stated their desire to sell "software as services" and charge users based on their usage of these services. This would require authenticating individuals, and the possibility of constructing a new structure for client/consumer authentication or building this functionality into the broker structure by extending it.

In conclusion, it must be stated that this project opened the author's eyes to the world of web services. While finding this area both interesting and stimulating, the author hopes to have the opportunity to conduct further research in the area.

References:

[1] Microsoft .NET - http://www.microsoft.com/net/
[2] Apache SOAP - http://xml.apache.org/soap/
[3] Web Services architecture overview - http://www-106.ibm.com/developerworks/webservices/library/w-ovr/
[4] "Dynamic e-business: using Web services to transform business http://www-4.ibm.com/software/solutions/webservices/pdf/wsintro.pdf
[5] Simple Object Access Protocol (SOAP) 1.1 - http://www.w3.org/TR/SOAP/
[6] Web Services Description Language (WSDL) 1.1 - http://www.w3.org/TR/wsdl
[7] Universal Description, Discovery & Integration - http://www.uddi.org
[8] SOAP Security Extensions: Digital Signatures - http://www.w3.org/TR/SOAP-dsig/
[9] Common Object Requester Broker Architecture - http://www.corba.org
[10] WebSphere Application Server - http://www-4.ibm.com/software/webservers/appserv/
[11] JavaMail API - http://java.sun.com/products/javamail/
[12] Public Key Infrastructure - http://www.ietf.org/html.charters/pkix-charter.html
[13] X.509 Certificates - http://java.sun.com/products/jdk/1.2/docs/guide/security/cert3.html
[14] Introduction to SSL - http://developer.netscape.com/docs/manuals/security/sslin/index.htm
[15] Web Services Inspection Language (WS-Inspection) 1.0 http://www-106.ibm.com/developerworks/webservices/library/wswsilspec.html?open&l=892,t=grws,p=wsils
[16] Centrata, Inc. - http://www.centrata.com
[17] SETI@Home : Search for Extraterrestrial Intelligence at Home - http://setiathome.ssl.berkeley.edu/
[18] When less is more: a compact toolkit for parsing and manipulating XML http://www-106.ibm.com/developerworks/xml/library/x-elexml/?dwzone=xml
[19] Electric XML - http://www.themindelectric.com/products/xml/xml.html
[20] XML Pull Parser 2 - http://www.extreme.indiana.edu/soap/xpp/