Web Services: Filling in the Gaps

(Master's Project)

Howard Newman
Advisor: Judith A. Stafford
{hnewman, jas}@eecs.tufts.edu
Tufts University
May, 2003

Abstract

Many professionals hail web services as the next wave in distributed computing and claim that this new distributed technology will profoundly affect the way organizations do business in the coming years. Web services is a service-oriented architecture; its most compelling benefits are that it promotes interoperability and is based on open standards. However, web services must be improved if it is to prove reliable, flexible, and maintainable in today's complex, heterogeneous computing environments. Some core, enabling technologies are lacking important features that can be found elsewhere or are extremely important either in distributed or component-based computing. Several improvement suggestions are offered and each suggestion is discussed at length to reinforce its importance in a service-oriented architecture. The core, enabling technologies of web services are then examined to determine how these suggestions can be leveraged by the web services' architecture. Areas of improvement focus on failure management, event notification, service registration, and interface descriptions. A discussion is also offered on how web services can be combined with other distributed technologies

1 Introduction

Software development has changed dramatically over the years. Developers were once concerned only with writing a program and making sure it behaved and performed well on one machine and under one platform. Programs then became distributed throughout an enterprise and needed to communicate with each other to accomplish particular goals. However, the programs within the enterprise were often developed for the same platform, were developed using the same language, and used platform specific and proprietary protocols for software communication. Currently, software programs are now distributed throughout the world, are installed on many different types of machines, and run under the control of any type of operating system. These software components

need a secure, reliable, and efficient way to communicate. Web Services has emerged as the latest buzzword in technology and its goal is to enable software to talk to other software through open, language-independent standards. Web services can essentially be viewed at a very high level as programmable components that provide a service either over the Internet or an Intranet. These services should be easily found by other services and then invoked to perform particular operations. Some are calling web services everything but a panacea for solving the problem of heterogeneous software communication. On the other hand, the industry cannot seem to agree on a concrete definition of web services, since this is such cutting-edge technology. Many technologists claim that they want to use web services in their businesses but do not exactly know what to use them for, do not know what their true capabilities are, and most important, do not know about the significant features that they lack.

This paper describes web services and what they offer today. It then offers areas where web services can be improved in order to provide more reliable, robust, and flexible distributed solutions for applications. It also discusses reasons for why it would be advantageous for web services to be combined with other distributed technologies. The purpose of the paper is to introduce some significant areas in detail where web services can be improved in order to give the reader a better understanding of where web services stand in the distributed computing world. The paper stresses why these particular areas are essential to improve if web services are to be considered a *bona fide* distributed technology. The paper does not offer complete implementation solutions but suggests at a somewhat high level where improvements can be made and which existing mechanisms should be altered. Therefore, another purpose of this paper is to encourage

further study in this ever-changing field of web services, and to motivate researchers to expand on and even implement some of the mentioned improvements.

Web services are definitely a leap in the right direction. They can bridge platform gaps that could not be previously bridged or at least bridged with considerably more effort. Yet, there exist significant areas where web services' mechanisms must be improved in order to offer reliable, viable, and complete software solutions. If web services are to be truly thought of as a robust distributed technology, they must implement mechanisms that take the shortcomings of the network into account. In distributed computing, the network is the one piece of the puzzle that programs cannot control. If there is a malfunction in the network or if the network comes to a grinding halt, applications cannot repair the network, but can definitely take the necessary steps to ensure that they are not left in an inconsistent state for a prolonged period of time. The clients of web services should also be efficiently notified in the form of an event when one of their interested web services change. Clients should be able to specifically indicate what type of change they are interested in receiving information about, and should be able to process this information quickly.

An efficient and easy-to-use registration mechanism is imperative to have in the world of web services. The registration mechanism should be flexible enough to include a wide array of information, but rigid enough to allow for programs to perform efficient searches on the information. A web service should also clearly specify its interface. An interface that is vague will lead to a web service that is of little use. An interface should be specific so that clients know exactly what to expect when invoking an operation. This is certainly not a new concern. Many think of web services as components, and the same

concerns are true of any components or services that are published anywhere. However, this concern is arguably more important in the web services world since these services could potentially be offered to a much wider audience across the Internet and to mutually untrusting parties.

It is advantageous to combine web services with other current distributed technologies to offer more complete and manageable solutions. The current implementation of web services does not support many features that contribute to robust, distributed software solutions. There are other distributed technologies available on the market today that could compensate for web services' shortcomings. This is not to say that web services cannot implement some of the mechanisms offered by other distributed technologies, but at the present time, it would be better to attempt to bridge web services with other existing distributed technologies.

This paper will first describe web services. It will describe the core enabling technologies of web services and the roles that they play. The paper will then focus on four specific distributed technology areas where web services can be improved: Failure Management, Event Notification, Service Registration, and Interface Descriptions. Lastly, the paper will describe how and why web services should be combined with other current distributed technologies.

2 Background

Businesses realize the need for having distributed applications within their heterogeneous infrastructure and need a reliable, cost-effective, and standard way for these applications to communicate. Consequently, web services are being hailed by many professionals as the next wave of the future in distributed computing with the intent

to profoundly affect the way organizations conduct business in the coming years. While there are various definitions of the term, “web services”, the most general definition is “a programmable component that provides a service over the Internet” [1]. Although this definition is correct, it is so high-level and hides all of the benefits that web services propose to offer. It also ignores the fact that web services are not restricted to use across the Internet; they can be used within company Intranets, which is probably where their use will be most prevalent.

The most important benefit of web services is that they employ open standards. This is essential in a heterogeneous distributed environment where there are applications running on any type of machine and under any type of operating system. A protocol that is based on open standards permits programs written in different languages, and running under different platforms, to effectively communicate. In the past, communication protocols such as CORBA, DCOM, and RMI were used but interoperability is limited when using these technologies. For example, if a DCOM component wanted to talk to a CORBA component, a DCOM/CORBA bridge would have to be created to provide effective communication. Moreover, if the DCOM and CORBA protocols change, programmers would have to modify this bridge in order to reflect the changes [2]. Making these changes is not trivial partly because both DCOM and CORBA use proprietary binary formats for their payloads.

If only DCOM is used for the communication protocol, both the sender and the receiver of messages must be running Windows. On the other hand, web services employ open standard protocols which use marked-up text as their payload, can communicate over the standard web protocol, HTTP, and the only requirement imposed

is that there be a text processor on the participating nodes to decipher the marked-up text. Thus, if a DCOM component needed to communicate with a CORBA component, it would not have to use a DCOM/CORBA bridge. Instead, it could rely on web services text protocols to send data over the wire and then build a Web Services/CORBA bridge, which will be more maintainable than a DCOM/CORBA bridge. Thus, through its use of open standards, Web Services enable disparate software components to effectively communicate.

Another benefit of web services is that they are loosely coupled and encapsulated. [3] If web services and the clients who invoke them are designed properly, making changes in the client should not affect the web service and making changes in the web service should not affect the client. The only way that a change can adversely affect either party is if an interface is changed in a web service and the client application is not notified of this change. In this scenario, a client application could be calling an interface that no longer exists, and will get an unexpected error. In general, both parties make every attempt possible to prevent this situation from occurring. There are often contracts that are signed where both the client and the service agree on the exported interfaces ahead of time. In fact, all a client needs to know about are these exported interfaces; they do not need to know anything about how these interfaces are implemented. This idea is known as encapsulation. Encapsulation is a concept that is central in systems that use object oriented design and in systems where there should be a strict separation between an interface description and an interface implementation. In web services, there are mutually untrusting parties talking to each other that are often separated by many firewalls across the Internet. From strictly a security standpoint, the two parties should

know as little as possible about each other and encapsulation is essential for keeping the "untrusting relationship" intact.

The core enabling technologies of Web Services are XML, Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and Universal Description, Discovery, and Integration (UDDI). It is a little awkward to list XML as a separate technology since WSDL, SOAP, and UDDI are all based on XML, but they are discussed separately in order to gain a more comprehensive understanding of web services' mechanisms. XML is a universal markup language on which all Web Services' technologies are built. WSDL is used to describe the interfaces exposed to clients and the transport protocols clients can use to contact the Web Service. SOAP provides the communications protocol for distributed programs to talk to each other in a networked environment. UDDI provides a registration and lookup mechanism for storing and retrieving Web Services interfaces and offers general information about the web service. Figure 1 illustrates the interaction among these technologies.

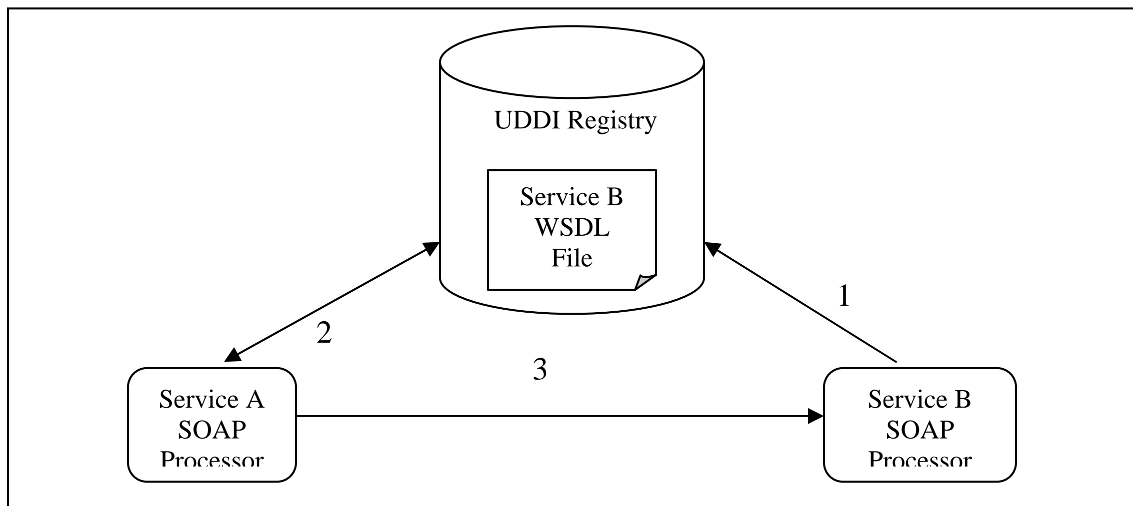

Figure 1 – Interaction among Web Services' components

The author of Web Service B generates a WSDL file and then registers it with a UDDI registry (1). At this point, the UDDI registry can contain a URL that points to the WSDL file that was just registered. Web Service A can then query the registry and obtain the WSDL file (2), and then send a message over the wire to service B (3).

For the purposes of this paper, web services will be defined as programmable components offered over a network that communicate using SOAP, describe themselves using WSDL, and can be found within a UDDI registry.

2.1 XML

As mentioned above, Web Services inherently rely on XML. Ugly in its syntax, but rich in its semantics, XML not only serves to describe particular data that are interpreted by multiple parties, but also governs the rules which tell one how to build XML documents for communicating with Web Services. All of the elements of an XML document and the rules that an XML document must obey to be syntactically correct will not be discussed here, but it is necessary to discuss some basic XML concepts in order to provide some foundation for further reading and research, and to understand how the other core web services' technologies work. Figure 2 is an example of a very plain vanilla XML document:

```
(1) <?xml version = “1.0” encoding = “UTF-8” ?>
(2)
(3) <businessinfo>
(4)         <company> Web Services ‘R” Us </company>
(5)         <city> New York </city>
(6)         <state> New York </state>
(7)         <zip>10003</zip>
(8) <businessinfo>
```

Figure 2 Sample simple XML Document

Line 1 is optional and identifies the document as an XML document. There is currently only one version of XML so only “1.0” can be used for the version attribute. “encoding” is an optional attribute that specifies the method to be used to represent characters electronically. UTF-8 is a character encoding that is used for Latin-alphabet characters that can be stored in one byte. Data is marked up using tags which are names enclosed in angle brackets (< >). [2] In Figure 2, the XML document contains information about a company. The businessInfo tag is used to indicate that a business is going to be described, and this tag is known as the root element of the XML document. Nested within this tag are the company, city, state, and zip tag. White space and indentation is used to express the hierarchical relationship among the tags. This particular whitespace is meaningless to programs that parse the document for correctness.

Accompanying an XML document will be a schema. Schemas precisely specify what elements and attributes are permitted within the document and how they relate to each other. The schema can also specify the data types associated with the elements, so that when data is mapped out of XML to a strictly typed programming language such as Java, the correct data types will be used to store data. The two types of schemas popular today are “XML Schema” and “Document Type Definition” (DTD). DTD’s have some

limitations since they cannot describe data types. [4] In order to make sure that XML documents are correct, parsers are used. There are two types of parsers. A validating parser will examine a schema to make sure that all the elements in the XML file satisfy the rules specified in a schema. A non-validating parser will only ensure that the XML file obeys basic XML syntax rules such as a “there must be a matching “>” for every “<”. In the world of web services, when an XML document is sent across the wire, processors (such as the SOAP processor in Figure 1) validate the XML document by matching elements declared in the XML document with elements declared in the schema and then map the data to a specific programming language data structure. Of course, the processor must satisfy the opposite condition as well; it must convert the data within the programming language data structure to an XML document, which can then be sent over the HTTP wire.

2.2 WSDL

Web services should be described and advertised so that people know how to use them. WSDL provides the structure for describing Web Services. A WSDL document is an XML document that describes the functionality that a Web Service exports, gives the URL of the service, and precisely specifies how clients transfer messages to the service. WSDL essentially represents an XML schema that is used to describe a web service. Potential clients can use the information contained within a WSDL document of several web services to determine which web service will best suit their needs. Both parties that are involved in a conversation must have access to the same WSDL file to understand each other. [4] In other words, both parties must be referring to the same interface specifications. The format of the input and output messages can be different for different

parties. From a clients' perspective, an output message is a call to invoke a specific function; for example, and an input message is a return value from a web service. Nevertheless, the sender needs to know how to format the output message and the receiver needs to know how to interpret the input message

Web services programmers do not need to be intimately familiar with the structure of a WSDL document since there are a wide range of tools on the market that generate WSDL documents, such as [5]. However, it is worthwhile to describe the overall structure of a WSDL document to understand exactly what WSDL specifies for a Web Service and to determine if there are any shortcomings of WSDL. A WSDL file can be categorized into three separate parts: the contents and data types of the messages, the operations performed on behalf of the messages, and the specific protocol bindings that are used for exchanging the messages with the operations over the network. [4] Figure 3 includes the relevant parts of a WSDL document for our discussion. The meaning of the tags will be discussed throughout this section. First, on line 1, the XML version is declared, just like in any ordinary XML file and on line 2, "<wsdl:definitions>" is defined as root element of the WSDL document. The rest of the document then specifies how a client can interact with the web service.

```
1 <?xml version=”1.0”?>
2 <wsdl:definitions name = “BookTitleService”  … >
3
4         <wsdl:message name = “BookTitleService_getBookTitle_Response”>
5                 <wsdl:part name=”response” type=”xsd:string” />
6         </wsdl:message>
7
8         <wsdl:message name = “BookTitleService_getBookTitle_Request”>
9                 <wsdl:part name=”p0” type=”xsd:string” />
10        </wsdl:message>
11
12        <wsdl:portType name=”BookTitleService”>
13                <wsdl:operation name=”getBookTitle” parameterOrder=”p0”>
14                        <wsdl:input name=”getBookTitle”
15                                message=”tns:BookTitleService_getBookTitle_Request” />
16                        <wsdl:output name=”getBookTitle”
17                                message=”tns:BookTitleService_getBookTitle_Response” />
18                </wsdl:operation>
19        </wsdl:portType>
20
21        <wsdl:binding name=”BookTitleServiceSOAPBinding0”
22                type=”tns:BookTitleService”>
23                <soap:binding transport=http://schemas.xmlsoap.org/soap/http
24                                style=”rpc” />
25                <wsdl:operation name= “getBookTitle”>
26
27                        <soap:operation soapAction=”” style=”rpc” />
28
29                        <wsdl:input name=”getBookTitle”>
30                                <soap:body use=”encoded”
31                                        encodingStyle=”http:://schemas.xmlsoap.org/soap/encoding/”
32                                        namespace=”urn:BookTitleService” />
33                        </wsdl:input>
34                        <wsdl:output name=”getBookTitle”>
35                                <soap:body use=”encoded”
36                                        encodingStyle=”http:://schemas.xmlsoap.org/soap/encoding/”
37                                        namespace=”urn:BookTitleService” />
38                        </wsdl:output>
39                </wsdl:operation>
40        </wsdl:binding>
41
42        <wsdl:service name=”BookTitle”>
43                <wsdl:port name=”BookTitleService”
44                        binding=”tns:BookTitleServiceSOAPBinding0”>
45                        <soap:address location=”http://myhost:6060/BookTitleService/” />
46                </wsdl:port>
47        </wsdl:service>
48
49 <wsdl:definitions>
```

Figure 3: Sample WSDL Document [2]

It is necessary for a web service to define the types of its inputs and outputs. WSDL has types that are typically defined using XML Schemas. These XML schemas are often present as separate documents that WSDL includes, so that they can be used by other services. Data types can range from primitive data types such as boolean, integer, and float to more complex types such as structures and arrays. Something important to keep in mind is that the types that are defined within the XML schema do not necessarily have to specifically match the data that is required by the back-end program. [4] This is quite intentional, as a software program on the back-end should not want to expose the exact layout of its data structures to clients for security reasons. As long as there is some mapping mechanism that will take the types defined in the schema and convert them to the back-end program's data structure, any reasonably expressive types should be able to be defined within a WSDL document. In Figure 3, lines 4-10 define messages of type "xsd:string" that will be sent to and received from the web service. Lines 4-6 defines the type of message that will be sent back from the web service and lines 8-10 define the type of message that will be sent to the web service. "<wsdl:part>" is used to specify the name and data type of the message that is exchanged.

The next level of abstraction within WSDL is operations. Operations address the requirements of a Web Service to identify the type of operations being performed as a result of an incoming XML message. All operations must be associated with a previously defined message. If an operation specifies a message both for input and output, then it is a Request/Response type of operation. Other types of operations that WSDL supports are one-way, solicit response, and notification. One-way means that a message is sent and there is no reply. Solicit response is not defined as a part of WSDL

at the present time but it means that one party is requesting a response but does not send a message that adheres to any of the WSDL defined types. Notification is also not yet defined but it specifies multiple receivers for a message and involves some sort of publish/subscribe mechanism. [4]

After the data types and the operation types are defined, they are mapped onto a specific transport protocol, or binding, and are associated with an address that will indicate where they should be sent. WSDL accomplishes this goal by introducing the concepts of "port types", "ports", and "services." Operations and messages are logically grouped together into a port type. In Figure 3, lines 12-19 define a port type. The port type states that the operation "getBookTitle" will take in the message defined on lines 8-10 and will return the message defined on lines 4-6. A port then identifies a transport binding for a given port type and a URL to use for accessing the service. Lines 43-46 define a port, which references a binding, which is declared on lines 21-40. Line 21 states that the name of the binding is "BookTitleServiceSOAPBinding0." Lines 23-24 indicate that "SOAP RPC" is to be used for invoking a service. "rpc" represents a keyword for identifying "SOAP RPC" and "http://schemas.xmlsoap.org/soap/http" is a unique identifier for specifying SOAP over HTTP. "document" can also be used for the style attribute, which means that a message not having an RPC structure would be sent over the wire. Lines 25-39 represent an operation that is invoked using "SOAP RPC." Both the input and the output element specify how the web service environment encodes SOAP requests and responses, respectively. It is important to note that for a given port type, more than one transport protocol or binding as it is called can be used to invoke the operation. This structure gives more flexibility to clients since they do not necessarily

need to get tied down to one transport mechanism. Lastly, the "service" element defined on lines 42-47 encompasses the port element. A service groups together different ports, so that a different set of operations, or port types can be invoked over different transport protocols. The service element can contain several port elements, each of which contains a URL for a unique binding. Since there is only one binding defined in Figure 3, only one port element is included within the service element. Figure 4 illustrates the aggregation relationship among port types, ports, and services.

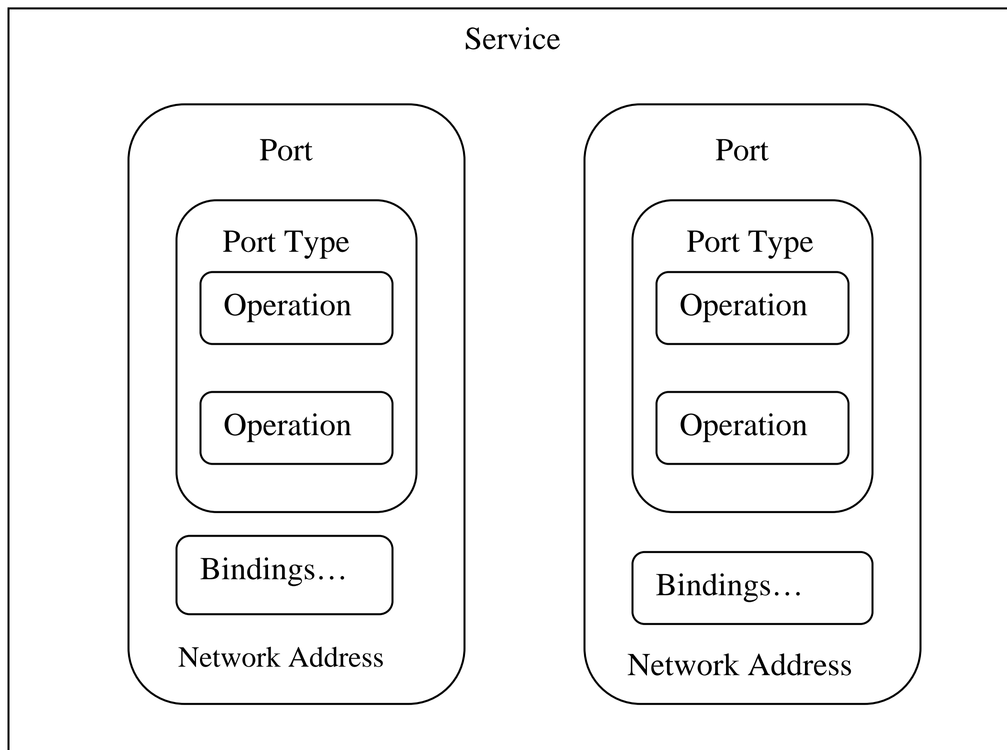

Figure 4 – Association between parts of a WSDL document [4]

As is evident from the diagram, WSDL is pretty flexible since it allows different operations to be grouped together for different purposes.

2.3 SOAP

A protocol must exist for transferring data from one node to another over the Internet. SOAP (Simple Object Access Protocol) allows the sender and receiver of XML documents over the Web to support a common data transfer protocol. SOAP supports transport through virtually any protocol. For example, there are SOAP bindings for HTTP as well as for SMTP.[2] SOAP transports an XML document called a SOAP message across the web. A SOAP message is simply an XML document that adheres to a specific format. SOAP processors are needed on the nodes involved in the conversations to understand the SOAP messages and map them to the web services' implementation language. [4]

A SOAP message has an envelope that describes the content, the intended recipient, and processing requirements of the message. The envelope also specifies the start and end of a message, so that the receiver knows when an entire message has been received. An envelope consists of a Header, which is optional, and a Body element. The Header element provides processing instructions for applications that receive the message, meaning that a header can extend SOAP so that it can incorporate more complex protocols. For example, if one wanted to support authentication, the Header could contain information such as which authentication mechanism to use. The Body element contains application specific data for the intended recipient of the message. [2]

There are two flavors of SOAP calls that can be made: "SOAP RPC" and "SOAP Messaging." "SOAP RPC" is used when the client knows the name of the method that they want to call and the implementation of that method is located in a different address space. The client simply indicates in XML which method they want to call and supplies

any arguments that the method requires. The server may then send back a response to the client but a response is not mandatory. With “SOAP RPC,” the onus is on the client to know *a priori* the name of the method to call and what arguments the method takes. Figure 5 contains an example of a “SOAP RPC” request. [6]

```
<SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="some-URI">
          <symbol>DEF</Symbol>
        </m: GetLastTradePrice>
 </SOAP-ENV:Body>
```

Figure 5 – Sample “SOAP RPC” file [6]

Only the SOAP body is included in this figure. The client is simply calling a function that they know about named “GetLastTradePrice” that is available at a particular URI, which could be an address of a particular service. “GetLastTradePrice” takes a symbol as its argument and the client will await a response after sending this SOAP message. The “SOAP RPC” model is typically used when the interaction between a service and a client can be modeled as a simple synchronous procedure call. [6]

“SOAP Messaging” allows a client to just send XML documents across the wire. Clients do not need to know about how this document will even be used on the server side or what back-end methods will need to be called as a result of the XML document being transferred. Rather, it is up to the server to call the correct method within the application. A server may return a response to the client, but just as in “SOAP RPC,” a return response is optional. Whether one uses “SOAP RPC” or “SOAP Messaging,” a SOAP document still contains a SOAP Envelope, optional Header, and Body. [6] An example of a “SOAP Messaging” call is demonstrated in Figure 6.

```
<SOAP-ENV:Body>
        <bp:GetBookDetails xmlns:bp="http://bookprovider.com">
          <searchCriteria>ISBN</searchCriteria>
          <searchValue>0123454321</searchCriteria>
        </bp: GetBookDetails>
 </SOAP-ENV:Body>
```

Figure 6 – Sample "SOAP Messaging" file [6]

Only the SOAP body element is shown here as well. At first glance, "SOAP Messaging" looks the same as "SOAP RPC." However, in this case, "GetBookDetails" is recognized as a structure that contains information about books, in particular, the International Standard Book Number (ISBN), by the web service. The server processes the message by searching a database of book information, using the ISBN value as search criteria. The server returns detailed information about the book that has the specified ISBN value. Notice that the client does not know the program that is handling this message nor does the client specify a given method to call. The response is not a return value of a method but rather a response generated by the processing application. [6] The SOAP Messaging model is a preferred model when XML-documents need to be passed between multiple nodes and it is expected that each node will change the contents of the document.

2.4 UDDI

After a web service is created, it must be registered with some well-known entity so that clients will be able to find and use the web service. A UDDI registry contains a directory of Web Services descriptions that authors can publish and clients can search. One can think of a UDDI registry as a phone book, but it is more flexible and

contains a wide range of information about a business and the web services it offers. UDDI also contains a generic searching mechanism, which allows one to easily find and examine particular web services.

The information within a UDDI registry can be classified into three distinct categories: white pages, yellow pages, and green pages. The white pages contain general information about a company such as its name, address, contact information, but also include a unique identification string to identify the company. Yellow pages divide companies into various categories based on the products or services they offer. The green pages contain technical information about a company's products, services, and web services. Not surprisingly, the green pages often reference WSDL documents. [2] XML schemas are used to express the data structures comprised within UDDI, which are quite complex.

At a lower level of granularity, the UDDI information model, or UDDI schema as it is often called contains five interrelated data structures that serve to extensively describe a company and its web services. The names of these five data structures are businessEntity, businessService, bindingTemplate, tModel, and publisherAssertion. All of these data structures are described in XML. The businessEntity structure encapsulates general information about a business such as its name, address, and contact information. businessEntity corresponds to the UDDI white and yellow pages. businessEntity references businessService which describes the different types of services offered by the company. The bindingTemplate structure contains technical information about these services, where these services are located, and references a tModel structure, which contains information about how to interact with the web services. A tModel will often

reference WSDL documents but does not have to by any means. The general idea behind a tModel is that it is meant to represent a specification for a web service. The tModel is not specific to a given business or entity, as many businesses may implement web services that satisfy the specification defined within the tModel. One can search for a particular tModel, and see which businesses implement the interfaces defined within the tModel. businessService, bindingTemplate, and tModel correspond to the UDDI green pages. The last data structure, publisherAssertion is used to indicate that a mutual relationship exists between two businesses or businessEntity's. Figure 7 illustrates the relationship between these data structures.

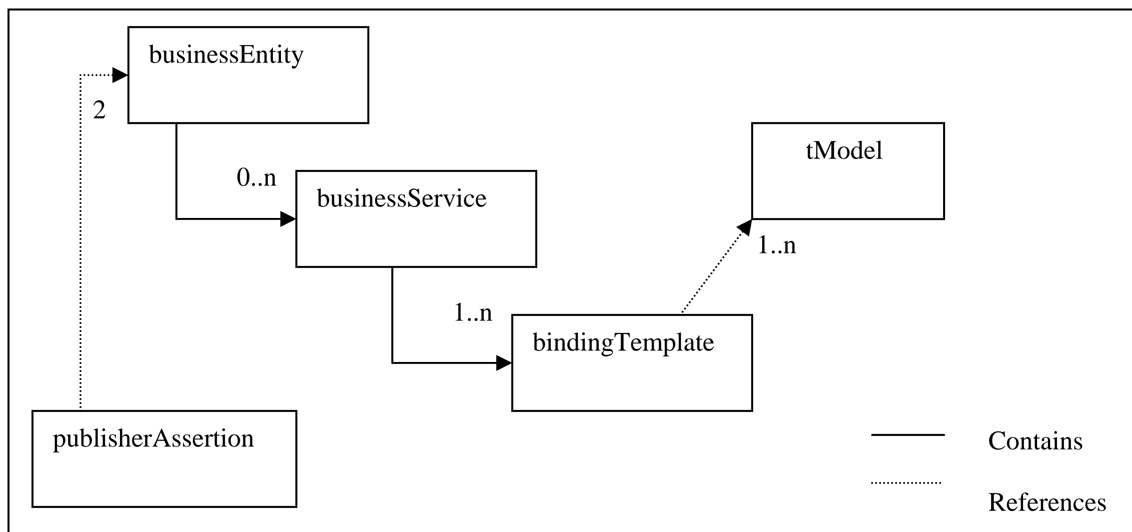

Figure 7 – "UDDI Data Structure Relationships"

UDDI provides APIs for authors to publish web services and for consumers to inquire about web services. The publishing API supports the 'publish' operation, which allows companies to register and update information within a UDDI registry. Important to note is that access to the publishing API's commands is restricted; users are authenticated when they make calls into the API and access to the API is only available

via HTTPS. The inquiry API, in turn, enables consumers to browse the registry for certain types of services. [2]

3 Improvements to be Made in Web Services

3.1 Failure Management

In order for a distributed application to be reliable, there must be some kind of failure management mechanism in place. Failure management mechanisms are necessary because any system, whether it is distributed or not, will never be completely reliable. Software programs can crash, and machines and networks can fail at any time. The failure of one component in a distributed system can bring down the entire system. To make matters worse, and what is even more disturbing in a distributed system is that a client cannot tell which component has failed; for instance, they cannot tell the difference between the targeted component failing and the network being unavailable. There is no common agent that is able to determine what component has failed and inform other components of that failure. There is also no global state that can be examined that will aid someone in determining exactly what error occurred. As stated in [7], "in a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that link."

One cannot design distributed systems so that none of the components can fail. This has proved to be an impossible problem to fix within the industry. However, applications should be designed so that they at least take these failures into account and acknowledge that these failures can happen. Applications must also ensure that they are left in a consistent state in the event of a failure. One way to make sure that applications stay within a consistent state is to put the onus of maintaining state and doing clean up on

the application infrastructure itself [8]. This is the basis for "self-healing" distributed systems. Obviously, if a human had to do the healing every time a failure occurred, then the systems can potentially become unworkable, since the time needed to do a manual repair would be much longer as opposed to the time a system would take to heal itself.

Web services do not have a failure model in place. There is nothing in any of the core enabling technologies, SOAP, WSDL, or UDDI that explicitly takes failure management into account. Web services should take failure management into account since they provide services across the Internet, where failure of certain components is definitely an unfortunate reality. One way to take failure management into account is to make sure that the UDDI registry always contains valid and current information. For example, suppose there is a web service that exists at a particular URL and is registered within the UDDI registry and this web service ceases to work, or is no longer available. There will still exist a stale entry within the UDDI registry for this service. Clients trying to use this service will think the service is up and running when they try to make a connection, but in reality they will not be able to use the service. If these stale web services are not reclaimed by something, they can stack up to a point which can make performing operations on the UDDI registry a very cumbersome task. More importantly, UDDI will give false information to its users as they will see web services registered that are no longer working There needs to be a way within the web services world to detect stale entries and dispose of them.

There are several solutions within the distributed computing community that can be used to dispose of stale entries. One solution is to have something within the UDDI registry constantly pinging web services to see if they are alive. If the web services are

not alive, then they will be removed from the UDDI registry. While this is certainly simple and doable, it is not sufficient. Although a web service may be up and running, there is no guarantee that the producer of the web service will still want it to exist in the UDDI registry, and to be accessible to the world. Also, forcing some mechanism within the UDDI registry to ping every single web service that is registered at particular intervals gives the UDDI registry a very expensive responsibility.

An alternative solution to managing failure management and the disposal of stale entries within the web services world is to use a leasing mechanism. Leasing was used within Jini Network Technology [9] to manage the registration of services. All services that are under the control of Jini are registered with a Lookup service, which can contain references to many services. However, these services are not just registered with a Lookup service forever; instead they are leased. This essentially means that when a service registers itself, it negotiates a lease with the lease grantor, the Lookup service. As long as the service provider continues to renew the lease with the lookup service, a reference to the service will still be present with the lookup service. The service can also cancel its lease with the Lookup service when it wishes to do so. Leasing is a very powerful mechanism that accomplishes several things. First, it guarantees that there will be no stale entries within a registry. If a service provider does not renew its leases, then the service will not be available within the registry; thus, clients will not have access to it. Also, a service may not renew its lease simply because it can't; it may have failed in some way that prohibits it from communicating with any other component. In this case, the service will be removed from the registry. Second, only one party, the lease grantor, knows that a particular service has failed or is unreachable since it can't renew its lease.

The clients that are trying to use a service do not need to know that it failed and do not need to worry about handling any failure scenarios. All of the failure management is done by the Jini Leasing mechanism and Lookup service. Third, failures are guaranteed to be detected, since there is a time limit imposed on how long a service takes to renew its lease. For example, if a service takes out a lease for ten minutes, and fails to renew itself, then it will be removed from the lookup service after ten minutes.

Leases can play a role in web services. UDDI, as it stands today is just a repository of textual information about a web service; it is not content aware. However, failure management should be included within web services, and UDDI should support leasing. This gives the UDDI registry more resource management tasks to do but if UDDI is going to be the place that contains a list of all available and working services, it should at least take the steps necessary to make sure that its registrations are current.

The businessService data structures within UDDI can be leveraged to support a leasing mechanism. As described earlier, the businessService data structure contains descriptions of services that a particular business offers. This structure is comprised of several fields including a unique key to identify the service, a field to reference the parent businessEntity data structure, the name of the service, and a description of the service. It would seem logical to add an additional field into this data structure called something similar to 'duration' that will indicate the lease time of the service. Additional fields such as when the lease was last renewed, and how long before the lease expires, can be added as well. The service provider would need to update these fields by renewing their lease on the service, and they would have to do this through the UDDI publisher API's. There is already a function within the publisher API that can be leveraged called "save_service"

which either stores a new or updates an existing busniessService. A new interface would not even be necessary within the publisher API. While this change to UDDI data structures may seem a little radical at first and a bit naive, these modifications are just taking advantage of the flexibility that UDDI currently offers. UDDI allows one to store a plethora of information about a service, and if one looks closer, there are quite a bit of optional and extensible fields already in place. Adding a few more fields for leasing can just be another optional field that UDDI can support.

However, one must consider exactly what component will be doing the lease management. A component needs to be added that will be able to discard of a service if a lease expires, and implement a counter to keep track of the lease time for a particular service. A short-term solution is to allow a third-party to do the lease management for UDDI. UDDI version 2 allows for third parties to perform validation on data that is entered into the registry. However, this leasing mechanism would be much more complex than just validating data. The leasing mechanism will need to be constantly running, and may potentially need to know about millions of web services, which is certainly no easy task. The number of instances of lease managers that will need to be running will need to be determined as well. If there is a very small number of lease managers running for a large amount of web services, stale entries will not be disposed of as quickly, which will make the concept of lease durations less meaningful. Also, web services are registered with one UDDI registry, but are then duplicated to other UDDI registries across the world. When a lease manager is forced to remove a service from UDDI, it will have to remove this service from all of the UDDI registries in which this service is registered. Admittedly, this is a very huge undertaking that UDDI will have to

take, but this scenario outlines at a high level one way of providing a robust failure management scheme for web services.

3.2 Event Mechanism

Many distributed systems have an event mechanism in place so that clients can be notified when something changes within their interested services. If web services want to be more reliable, self-maintainable, and robust, it would be advantageous for web services to offer some type of bona fide event mechanism. WSDL defines "Notification" as a type of operation, but there is no precise definition of "Notification" defined within the WSDL specification. According to [10], WSDL notification operations represent a client making a single invocation of an operation in a server. A connection between source and target operations represents a one-time event carried over a point-to-point connection, rather than a stable communication pipeline between an event source and one or more event consumers, as is the case in the event notification model. "WSDL will quickly find itself lagging behind the existing web infrastructure unless it is extended to provide first class support for event notification" [10].

"publish/subscribe" is a popular paradigm that is used in many event mechanisms. The general idea behind publish/subscribe is that a client will subscribe to particular events and a server will notify the client in the form of a message when the event occurs. Although many events may occur on the server side, the client will only be notified when the particular events that they have subscribed to have occurred. An event notification service is implemented to keep track of the event subscriptions and notifies the subscribers.

Important considerations must be taken into account when using an event mechanism within a distributed environment and these considerations certainly apply to the web services world. One consideration is that events within a distributed environment may not always get delivered. As was discussed in section 3.1, the network and any of its components can drastically fail at any time. Also, if the recipient of the event fails, a policy must be put in place that will govern how many times the sender should attempt to re-send the event before it discards the event. In addition, in a distributed environment, events may not be sent to the clients in the order they were generated. In a local environment, the order can more or less be guaranteed because a centralized queue is often used to keep the events in chronological order. [8] If a distributed system had a centralized event manager, performance would be severely impacted. Nevertheless, when designing an event model for web services, these important factors must be taken into account.

Before an event infrastructure can be developed for web services, one must consider what types of events a client is interested in receiving. If a client is using a web service, it may want to get notified if that web service gets deregistered from the UDDI registry. Once a client has a WSDL file for a service, it uses the WSDL file to contact the service. However, if the service is removed from the UDDI registry and disconnected from the network, the client will not be able to contact the service. It would be advantageous if a client was notified when a particular service was removed from the UDDI registry, so that they are given some indication that they cannot use the service anymore. This type of event could even work in collaboration with the leasing

mechanism; when an event is removed from the UDDI registry because its lease has expired, a notification can be sent out to all interested clients.

A client may also want to be notified when a web services' WSDL file changes. One can state that if a WSDL file changes, a new web service is really being created since the interfaces, and even the rules for contacting the web service may change. However, this can still prove useful, since one may just be calling the "save_tModel" interface, which is a part of the UDDI publisher API. This will prevent a client from using an outdated WSDL file when accessing a web service. Thus far, two examples that leverage the UDDI API's have just been given for generating events. Yet, there is nothing technologically stopping web services from generating events for any of the publisher API's interfaces. By having an event associated with all of the publisher API's interfaces, clients will know whenever an existing web service has been updated. In the case where a WSDL file is updated, the client will know that they will have to do another lookup within the UDDI registry to retrieve the latest WSDL file.

It is important to note that at the time of this writing, there is an event notification mechanism in the UDDI Version 3 specification [11]. An API is specified that lets clients subscribe for events and allows services to publish events. Clients can receive events either asynchronously or synchronously. Clients can also subscribe to receive particular notifications such as when a new service joins the registry, or when a particular service is removed from the registry. However, these notifications are limited in expressiveness, and are at a higher level of granularity than other types of notifications found in other distributed systems. Also, one does not know how efficient the event notification service is, whether it is designed appropriately to handle events across the

Internet, or what policies are in place to adequately account for the limitations of transmitting events in a distributed environment. Thus, it is imperative to examine what factors must be considered in designing an Internet-based event notification service.

The most important question to answer is what type of design should be implemented so that events are managed and delivered most efficiently. This design must address most, if not all of the limitations of distributed events mentioned earlier. In addition, one must consider the issue of how such an event mechanism can scale to the Internet. Many publisher/subscribe designs have been implemented, but having a publisher/subscribe design implemented across the Internet carries its own ramifications.

SIENA [12], a technology developed by Carzaniga, Rosenblaum, and Wolf is an event notification service that can be used on the Internet. There are several goals of the SIENA project. The overall goal is to use a network topology that implements the publish/subscribe paradigm in a very efficient manner. Another goal is to "maximize expressiveness in the selection mechanism without sacrificing scalability in the delivery mechanism" [12]. In other words, a notification mechanism should have a semantically rich language for specifying what types of notifications a client would like to receive, but at the same time, the processing of these notifications within the event mechanism should not hinder event delivery. SIENA also describes how event notification services are distributed throughout the network in certain topologies to maximize the efficiency of event delivery.

An event service that implements the "publish/subscribe" paradigm usually has just two API's available to clients: publish and subscribe. As Figure 8 illustrates, SIENA supports these interfaces and adds some additional interfaces.

```
publish (notification n)

subscribe(string identity, pattern expression)
unsubscribe(string identity, pattern expression)

advertise(string identity, filter expression)
unadvertise (string identity, filter expression)
```

Figure 8: Interface of SIENA

The parameters of these interfaces will be described first. A notification is a set of typed attributes. Each individual attribute has a type, a name, and a value. An identity string holds a unique value that is associated with either an event generator or an event client. A filter selects event notifications by specifying a set of attributes and constraints on the values of those attributes. A pattern is used to match against one or more notifications based on both their attribute values and the combination they form. [12] For instance, someone might want to only receive an event notification only if a related event happened elsewhere.

"publish" is used by event sources to send an event to an event notification service. "subscribe" is used by clients to express interest in receiving a particular event and takes an identity string and a pattern as its arguments. SIENA adds 3 more API's to its event service: unsubscribe, advertise, and unadvertised. "unsubscribe" is used by clients to indicate that they do not want to be notified of a particular event or events. "advertise" is used by event sources to notify the event notification service about which kind of notifications they will be publishing. The "advertise" API allows the event notification mechanism to best direct the propagation of subscriptions, meaning that subscriptions can be bound to notifications more easily. It is important to note that

“subscribe”, “unsubscribe”, “advertise”, and “unadvertise” accept an identity string as a parameter. The use of this parameter enables subscribers and advertisers to cancel their own subscription, or advertisements, respectively. SIENA also associates a timestamp with each notification to indicate when it was published. This allows the event notification service to detect and account for latency on the matching of patterns, which means that within certain limits, the actual order of notifications can be recognized [12].

As SIENA is a distributed event notification service, it is composed of interconnected servers, each one serving a subset of the clients of the service. The servers communicate in order to cooperatively distribute the selection and delivery tasks across a wide-area network. Three critical design issues must be answered when having a network of servers providing a distributed service. First, the interconnection topology, or the configuration in which the servers must be connected has to be addressed. Second, a routing algorithm, or the information that should be communicated between the servers to allow for correct and efficient delivery of messages has to be considered. Finally, a processing strategy, or where in the network a message should be processed to optimize message traffic must be taken into account. [12]

SIENA was tested on several networking topologies. Two of the networking topologies that SIENA was tested on were a hierarchical client/server architecture and a general peer-to-peer architecture. In a hierarchical architecture, pairs of servers interact in an asymmetric client/server relationship. A server can have any number of incoming connections from other “client” servers, but only one outgoing connection to its own “master” server. The client servers would send notifications, advertisements, and subscriptions to its server. Each server is then responsible for propagating information

that it receives on to its master server. One of the drawbacks to using a hierarchical server architecture is that every server acts as a critical point of failure for the whole network. A failure in one server disconnects all the subnets reachable from its “master” server and all the “client” subnets from each other. SIENA is best suited to use the hierarchical architecture when there are low densities of clients that subscribe and unsubscribe very frequently. [12]

The Peer-to-Peer architecture allows for bi-directional communication between two servers. Unlike the hierarchical architecture, there can be multiple paths between servers, thus making it more robust with respect to failures. SIENA is best suited to use a peer-to-peer architecture when the total cost of communication is dominated by notifications. Peer-to-Peer should also be used in situations where there are a high number of ignored notifications, meaning notifications for which there are no subscribers. [12]

Routing strategies are very important in these network topologies. The main idea behind the routing strategy of SIENA is to send a notification only toward event servers that have clients that are interested in that notification and to also use the shortest path to do so. Two generic principles that are requirements for the routing algorithms are downstream replication and upstream replication. Downstream replication means that a notification should be replicated *down* as close as possible to the parties that are interested in it. Upstream replication means that filters are applied and patterns are assembled as far *up* as possible, meaning near the sources. Classes of routing algorithms that broadcast data can implement these principles. One class, subscription forwarding, has the subscriptions set the routing paths for notification. The subscriptions are

propagated throughout the network so as to form a tree that connects the subscribers to all the servers in the network. When a notification is published, the notification follows the reverse path that was put in place by the subscription. [12]

As in all distributed systems, it is important to determine how well the architecture scales. In particular, the factors that SIENA can control are the definitions of notifications, filters, and patterns, and the complexity of evaluating them. It was found that the complexity of determining whether a given subscription and a notification were related was O (n+m), where n is the number of attribute constraints in the subscription filter and m is the number of attributes in the notification. [12]

SIENA illustrates even further the complexity of putting an event mechanism in place within a distributed environment. Web services could use SIENA as a model for putting an event notification mechanism in place. Web services do not have to include all of the mechanisms that SIENA includes but should at least leverage the research done by the SIENA architects when constructing an event notification mechanism. Each web service would be responsible for coming up with its own notification data structure that would carry the notification information. This notification data structure could be composed of a set of typed attributes, just like the SIENA notification data structure. This data structure of attributes is very flexible since an unlimited number of attributes can be added to the data structure. However, it is a little inflexible since it can only have primitive types as members, whereas web services makes use of some complex data types. Of course, it would be best if there could be some sort of standardized way for representing a notification within the web services world. This standard structure would greatly help in allowing both web services and clients to effectively communicate using a

common language. Anatural place to specify the notification data structure would be in a WSDL file. A new section could be added that would specify the attributes of the notification data structure that a client could use to subscribe to events and a service could use to advertise as well as publish events. Each of these attributes could also have some operations associated with them such as '=', '<', and '>', and the allowable operations would need to be defined as well. An alternative approach to a notification data structure and an expressive language is to have a notification be a value of some named, explicit notification type. This is how Java Distributed Events works. However, this is certainly not feasible on the Internet scale, as it would require some sort of global authority for managing the type space.

Another event related question to consider for web services is exactly how a service will get notified of an event. A handler is often used to specify the means by which an interested party receives notifications. A party can receive notifications either through the use of callback functions or through messages sent over the wire. This is similar to the difference between SOAP-RPC and SOAP Messaging as discussed earlier. In the case of web services, certainly SOAP can be used to transfer events over HTTP or some other protocol. The handler would just have to be defined somewhere. The interested parties, or event consumers implement the handlers. These may or may not be web services, but still need to expose an interface or define rules to indicate how they want to receive events. One way to represent a handler is to have a WSDL file for a client, or in other words, the "subscription listener." This idea transforms a client into a service, but will ensure that events get delivered and handled.

3.3 Service Registration

As mentioned in section 2.3, UDDI is very flexible and essentially allows one to specify almost any of type of information about a web service. The question to answer, though, is whether UDDI is too flexible. As is often the case, increasing flexibility increases the chance of invalid data appearing within a registry, and hinders a clients' ability to perform specific categorical searches. Increased flexibility can also lead to an inconsistency of registration information among different web services.

For example, many of the data structures within UDDI contain optional fields. The contacts data structure, which is contained within the businessEntity data structure, enables a client to contact the company that offers a particular web service. This structure consists of a person's name, which is a required element along with four additional optional elements: phone numbers, email addresses, a description of how the contact information should be used, and a postal address. There is also a "useType" attribute, which is used to describe the type of contact. "Suggested" values for this include "technical questions," "technical contact," "establish account," and "sales account."

There are obviously many problems with this type of loose data structure. None of the optional elements within the businessEntity structure indicate exactly how their data should be formatted. For example, the "address" element contains an "addressLine" element, which represents a postal address but this element does not specify the format of the postal address or even break a postal address up into four additional elements like number, street, town, and zip code. The address simply accepts plain text as its format. One of the ramifications of this format is that there can be invalid addresses, if there is no

validation mechanism in place. Also, addresses will look different within different businessEntity data structures. If one wanted to perform a search for businessEntity data structures based on zip code, they would not be able to do so easily, since a zip code is not defined as its own type within UDDI; it is only part of a plain-text string that should be included with an address. It should be noted that the UDDIv3 specification states that an “address” can be given more meaning by using a tModel in conjunction with the “addressLine” elements. However, it would be better if there was some standardized and organized structure for an address to begin with rather than having a web service producer use a tModel for basic addresses. Another element worth examining in the contact data structure is “description.” The value of this element consists of unstructured plain text, and according to [11], “description” is used to describe “how the contact information should be used,” which is a very vague definition of this element. It would be helpful if UDDI could actually state what it means by this definition, and it is not worth the energy to even guess what this means. Yet, if a description can be classified into a particular category, it would be advantageous for UDDI to define these categories, so that clients can maximize the utilization of the contact structure

While UDDI should be structured better, the current arrangement and format of its data structures is understandable. Information within a UDDI registry will be able to be seen by the general public. Many companies may not want to provide a lot of information about themselves since not all parties can be trusted. In this case, UDDI does not really have much of a choice but to make a lot of its fields optional. Also, UDDI may not want to provide the format for many of its elements because it may want businesses to add whatever type of information they want to these elements. It does not

want businesses to be forced to provide specific information if they do not want to do so. For example, in the case of an “address,” a business may just want to its zip code rather than its street address for security reasons. This allows for businesses to essentially expose and not to expose whatever information they want, even though it stops UDDI from defining a more specific searching mechanism.

Another issue to consider when looking at the UDDI data structure and all of its elements is the types of parties that will be examining information contained within a UDDI registry. One must consider whether programs or humans or both will be evaluating the data. Many pro-UDDI articles say that software programs will be reading data stored within UDDI and making decisions about whether to use a particular service. However, the way some parts of UDDI are structured today, one can make the assumption that humans will be reading and evaluating the data contained within a UDDI registry, rather than software programs. For example, the “description” element within a businessEntity data structure can contain a significant amount of textual information, and it would take considerable effort for a software program to attribute meaning to this text. On the other hand, if this text was in a specific marked-up format, programs would be able to decipher marked-up text and automate the process of selecting particular web services to use, for example. As mentioned above, though, it appears that UDDI does not want to be responsible for defining many specific attributes within particular XML elements that are needed for describing a web service; for now, it is obvious that it wants to make its data structures as flexible as possible.

3.4 <u>Interface Descriptions</u>

As mentioned in section 2.2, a web service description is contained within a WSDL file. The WSDL file contains operations, which clients can invoke to make use of a web services' functionality. These operations usually contain a meaningful name so that clients can deduce with little effort the meaning of the operation. The input and output parameters of an operation are also defined so that clients will know the type of information they will send to, and receive from the operation when it is invoked. One question that immediately comes to mind is whether the description of the operation in WSDL really exposes enough properties to give the client sufficient information about the true capabilities of the operation and the assumptions it makes about the way in which it will be used. To take this question one step further, one must also ask whether the aggregation of these operation definitions gives the client a sufficient idea about the true capabilities of the web service.

Software engineers often develop software that is dependent upon interfaces that have been developed elsewhere. In order to successfully use these interfaces, developers must know about the behavior of these interfaces and any constraints that they must adhere to when invoking a function within an interface. Very often, developers use an API, or an Application Programming Interface, which conveys some interface information about a component. Unfortunately, the terms API, and interface are often used interchangeably, and have different definitions to different people. An API is often a collection of signatures and there are often some comments dispersed throughout the API, which can describe some of the operations in more detail. Yet, the amount of behavioral information that is present in an API is often limited. Some scholars [13]

have noted that there is a major difference between an interface and an API, and say that an interface offers much more than an API. According to [14], "a component interface is a specification of assumptions that a client can make of any component that implements that interface." Another distinction worth noting is that an API is usually written to serve developers who are already using its functions and writing programs against them, whereas an interface is used to convey information that will guide the developer in making a decision about whether to use the interface. An interface should contain all of the assumptions and should be used if a developer wants to get a complete idea about how a component works. The web services' world must decide whether WSDL is representing simply an API for the web service or whether its goal is to represent an interface.

Currently, a WSDL file contains a list of operations and each operation contains a signature, so a WSDL file is nothing more than an API. According to [4], "the requirements for the message data and operations part of WSDL… are that they express enough of the data and semantics of the software program to allow a bridge to be constructed to it over the network using the capability of the transformation and mapping phase at each end." It appears that WSDL is more concerned with making sure a mapping to a Java program, for example, occurs correctly, than offering a meaningful description of its functionality. It seems logical that a WSDL file should really be representing an interface for a web service, so that clients will have a better idea of how to best make use of the web service. Thus, if WSDL truly wants to represent the true behavior of operations, and be more than just an API that exposes operations, additional properties should be specified within a WSDL file for each operation definition.

An interface explains what other developers need to know about a particular entity in order to use it in combination with other entities. It also provides a statement of other visible properties. When one documents an interface, they must strike a balance between disclosing too little information and providing too much information. If not enough information is disclosed, developers will be prevented from successfully interacting with the entity. If too much information is disclosed, the interface can be too complicated to understand and future changes to an interface will be more difficult. Thus, the architect should carefully choose what information is permissible and appropriate for people to assume about the interface and unlikely to change. [15]

There have been several recommendations on how to document an interface. We will not enumerate all of the recommendations here (for a list see [15]). Since these are only recommendations, they all do not have to be used but should be strongly considered. The heart of an interface document is the resources that are provided to its clients. For the purposes of this paper, one can think of these resources as functions or operations. The resources must be defined both syntactically as well as semantically. Defining a resource syntactically involves documenting a resource's signature, which allows programmers to write syntactically correct programs that use the resource. Defining a resource semantically means defining what exactly happens when the resource is used. Semantics can include changes in a component's state brought about by using the resource; this includes things such as side effects that occur when invoking an operation. Side effects can include, for example, how calling the current resource will affect calls to other resources in the future. [15] For example, if someone invokes "pop()" on a stack data structure, the value returned by a subsequent call to "get_size" will be decremented

by 1. Semantics also include documenting human observable results. In the world of web services, this can mean, for example, a different page being displayed in the client's browser after a call is made to the resource. One should also define any resource usage restrictions when defining resource semantics. Resource usage restrictions attempt to supply the circumstances in which the resource should be used. In some cases, data may need to be initialized or a method like "init()" may need to be called before using any of the other resources. Resource usage restrictions can also address how many clients are allowed to use the resource at a given time, or if some clients are only given read-only access to particular resources. [15]

Another part of an interfaces' documentation should include quality attributes. These can include performance and reliability parameters. These parameters will be used when one is deciding on whether to use a particular interface. An interface should also include information concerning what the resource requires. At a very specific level, a resource may require named resources provided by other elements and will need to include the syntax, semantics, and usage restrictions of these resources. At a higher level, a resource will often have system dependencies [15]. For example, there will need to be a Java Virtual Machine installed on a system in order to execute a Java program. Finally, it is recommended that an interface document include a section about the reasons behind the design of an interface. The rationale should explain the motivation behind the design, constraints and compromises, alternative designs that were considered and rejected and why, and any insight the designer has about how to change the interface in the future. [15] By including these rationale and design issues, users of the interface

should be able to get a view inside the “black box,” determine whether the interface meets their needs, and see how they could best use the interface to their advantage.

As mentioned above, it is important to represent both the syntax and semantics of an interfaces’ resources. Describing semantics is very difficult, and understanding semantics can be even harder. As was discussed, there are many types of semantic properties that can be represented. It would be advantageous for the web services’ world to come to an agreement on exactly what kinds of semantic properties should be present within an interfaces’ documentation, or in other words, within a WSDL file. Web services do not have to represent all of the semantic properties that are possible but should at least put a stake in the ground by defining some useful and relatively easy-to-understand properties. Some example properties that could be useful, and are well known are preconditions, and postconditions. A precondition is an assumption that a function makes about the values of variables external to a method. These variables can include the parameters that are passed to the function. In fact, preconditions are often expressed using the variable names of the parameters. For example, after examining a precondition, a client would then make sure that they specify a valid parameter to a function. A postcondition specifies what a given function has accomplished which may not always be evident by the name of the function. Also, the issue of “Quality of Service” parameters are currently being discussed extensively in the web services world and parameters like “average response time,” “average completion time”, or “start-up cost” are viable candidates for semantic properties. Of course, there are many other properties that can be represented, but web services’ must walk before it can run.

In terms of the interface properties of providing information about a particular resource's dependencies and discussing the rationale behind several design issues, web services may not want to provide too much information in these categories. This is where striking a balance between providing too much information and providing too little information comes into play. If a resource is dependent on something that is not available publicly, a web services architect may not want to publish this type of information to the world. The web services world is consumed with mutual untrusting parties and it is often the case that one party wants to express as little about itself as possible. Syntax and semantics are important in describing how to use a web service, and they should be included to a larger extent within web services' today, but a web service should not need to describe all of its dependencies or explain the rationale behind its design.

While it is necessary for a web service to provide more information about its operations, one major challenge is how to represent the aforementioned semantic properties. One must also consider who is going to be processing this information. If this information is only going to be processed by humans, a naïve approach would be to just include comments within the WSDL file to represent these properties. These comments will more than suffice for describing an operation, and will be analogous to how information about a function or method is included within an API. However, according to many web services documents, software should be able to talk to software and decide whether or not to use a particular web service based on commonly defined properties. As was discussed in section 3.3, some of the data about a web service is represented very concretely, and in specific formats but other data is represented in plain

English and no format is specified. Hence, representing operation properties is subject to the same problem. XML is a natural choice for representing this additional information since all of web services' technology is built on top of it, and its use is extremely flexible. Of course, the content within an XML tag must be computer-interpretable and complex schemas would need to be put into place in some cases to represent complex properties. Defining software decipherable properties will add to the scheme that WSDL currently employs.

One of the new buzzwords on the market today is "semantic web" and some of the ideas it promotes can be used in describing Web Services. As its name implies, "semantic web" means giving meaning to web sites or services so that other programs can decipher what a given web service provides, and decide whether they want to use it. As W3C states, the semantic web is the "idea of having data on the web defined and linked in a way that it can be used by machines not just for display purposes, but for automation, integration and reuse of data across various applications" [16].

DAML-S [17], a technology whose goal is to enable the semantic web, is essentially a semantic mark-up for web services. It is concerned with enabling automatic web service discovery, automatic web service integration, automatic web service composition and interoperation, and automatic web service execution monitoring [17]. For the purposes of our discussion, we will only be concerned with how DAML-S can describe web services' interfaces better by using a computer-interpretable description, or in other words a description that does not require human intervention.

DAML-S uses a "Service Profile" to provide a high level description of the service. Within the "Service Profile" are descriptions of the input and output parameters

for a particular function within the web service. However, DAML-S also realizes the need to provide for preconditions and postconditions within its language. In DAML-S terms, preconditions represent “one or more logical conditions that should be satisfied prior to the service being requested” [17]. For example, a precondition can be used to indicate that a valid credit card number must be used before ordering something from an on-line store. DAML-S also goes a step further by having an attribute called “accessCondition” which only allows particular users to access certain parts of the service. DAML-S even takes a giant leap by saying that in the future it will use logical rules for expressing relationships among input parameters. For instance, a rule might say that if a particular input argument is bound in a certain way, more input arguments will not be necessary or will be provided by the service itself. [17] Hence, although it is difficult to modify WSDL to include more properties to describe a service better, it is certainly a doable task and is currently being done by those in the semantic web space.

It is worthwhile to note that at the time of this writing, some specification requests have been submitted to include meta-data information for the Java programming language. Some members within the Java community have submitted a Java Specification request whose aim is to develop a metadata syntax within the Java language which would allow classes, interfaces, fields, and methods to be marked as having particular attributes. JSR 175 [18] states that one of the possibilities is adding a new keyword, “meta” into the Java programming language. It also stresses that there should be a definition of a runtime delivery format for metadata and of runtime API’s so that tools and libraries can access metadata information at deployment time and at runtime.

The JSR does also not rule out compile time checking to ensure the validity of the metadata, as the attributes may take the form of standardized strings [18].

3.5 Combining Web Services with other Distributed Technologies

Web services and other many other distributed technologies in the marketplace today are considered service oriented architectures. Service oriented architectures take existing software components residing on a network and allow them to be published, invoked, and discovered by each other. They allow a software programmer to model programming problems in terms of services offered by components to anyone, anywhere over the network. [19] As was discussed, web services certainly offer benefits to the computing industry today, but this paper has also identified specific areas of web services that should be improved. It should be fairly obvious that all of these improvements require substantial technical effort. At the same time, it is evident that these improvements require a prolonged and complex political effort, as the web services' community is still struggling to standardize improvements to existing as well as future technologies. While the industry waits for these improvements to be implemented, it would be worthy for those in the web services community to invest in an alternative plan. This secondary plan would consist of using web services, and taking advantage of all of its benefits like interoperability, but at the same time, combining it with other distributed technologies to compensate for its shortcomings. One can imagine bridging web services' core technologies with the core technologies of other distributed systems. As a result, all technologies will mutually benefit, as they will combine to offer a more robust and functionally complete infrastructure than any of the technologies could provide alone.

As mentioned in this paper, the main purpose of web services is to promote interoperability. By sending text based messages across the wire, web services can be used by more than one platform. However, we have also mentioned how web services' core technologies are static in nature. They do not take the dynamics of the network into account, so they are not self-administering or self-healing and require human intervention in order to remove or add services from registries. In order to help remedy the static nature of web services and effectively produce better services, web services should be combined with other service-oriented architectures that offer more dynamic capabilities.

We will use Jini [9] as a distributed technology that can be used with web services. Jini takes the fallacies of the network into account and addresses the problems of creating services that are intended to be adaptive, self-healing, self-administered, and distributed. Jini can be combined with web services in several ways. One way is to essentially have the web services be implemented as Jini services on the back-end. When a client is looking for a service to use, they will send SOAP requests to the UDDI registry. Once the desired web service is found, the client could download the WSDL file and use it to invoke particular operations on the web service. Proceeding normally, the SOAP processor on the server will then translate the calls from the client. In the backend and unbeknownst to the client, this service can take the form of a Jini service. The service could even be a part of a federation of services that could be acting in collaboration to perform specific tasks. The SOAP processor could then dispatch calls from a client to all of the services, which will all be found by using the Jini Lookup Service.

Having a web service implemented as a set of Jini services has implications on the overall health of the web service. Since Jini is designed to be used on a network where services can be constantly disappearing and reappearing, the Jini services that comprise the overall web service can be upgraded, replaced, or altered without affecting the outside world's view of the web service. If each of the Jini services were separate web services, or were part of a pipeline where services talk to each other, and Jini was not involved in the architectural picture at all, one would not be able to alter these services without temporarily stopping one of the web services, which should really be unacceptable in distributed computing today.

Another area where it is worthy to explore the combination of Jini and web services is where Jini services and web services need to interact. Currently, there is no way to accomplish such a task in either the Jini or web services world. For the sake of argument, suppose that on a company intranet, there are services that are implemented as web services, have WSDL descriptions, and are registered within a UDDI registry. Suppose that there are other services that are Jini-enabled and have service registrations within one or more lookup services. The web services and the Jini services need to talk to each other so that they can fulfill specific requests. In order to enable this communication, a bridge needs to be constructed between a Jini lookup service, and UDDI/WSDL. Thus, Jini services would need to be converted to web services and vice-versa and these services can be represented as both Jini services and as web services.

Constructing a bridge between UDDI/WSDL and a Jini lookup service is certainly not a trivial task. It is worthy to explore both directions of the bridge: a Jini service finding a web service and a web service finding a Jini service. If a Jini service wants to

communicate with a web service and take advantage of all the benefits that Jini offers, the web service needs to be converted into a Jini service. The information located in UDDI and in the WSDL file of the web service has to be converted into a service registration that resides in a Jini lookup service. The UDDI registry has to be examined to glean information such as the description of the web service, the producer of the web service, and the unique keys that identify the web service. However, as stated in section 3.4, gleaning this information programmatically is no trivial task, as many of the data fields within the UDDI data structures do not require any specific representation format. At minimum, though, the "tModel" must be referenced since this data structure references the WSDL file, which contains the operations that can be performed on the web service, as well as where the web service resides. A parser is also required to look through the WSDL file, examine all of its operations, examine the name of the web service, and create a Java interface.

A Jini proxy then needs to be created as well and registered with the Lookup Service. If the operations defined within a WSDL file uses complex data types for their input or output parameters, Java classes would need to be made as well that represent these data types. These data types would be constructed by examining the XML schemas representing these data types. The technology certainly exists for this today, as the Java language offers a number of classes that are capable of parsing XML documents and generating representative classes. The proxy provides some form of implementation for the operations or methods, but also contains information about how to communicate with the corresponding service. Jini is fortunately said to be independent of on-the-wire protocol, since a proxy and any of the methods implemented within the proxy can use

different protocols to communicate with the web service. It is up to the service to specify which protocols should be used for which methods. WSDL is similar as different operations can be invoked using different protocols. One key difference, though, is that a Jini proxy does not explicitly state what protocols its methods use to communicate with the service, whereas a WSDL file explicitly states the bindings that need to be used for each method in order to contact the service. WSDL gives the client a choice in many cases which communication protocol they would like to use for a particular operation, so the decision rests with the user of the web service as to how to contact the service. Conversely, in Jini, the proxy decides for the client how to communicate with the service, and the client has no influence over what protocol is used. For example, suppose a method within a proxy accepts an array as an input argument. The proxy can decide that if the array is "small," SOAP can be tolerated to communicate with the service. If the array is "large," a different protocol like RMI may be more desirable. Nevertheless, it thus seems logical that by looking at the bindings for individual methods or for a group of methods with WSDL, and with some effort, a Jini proxy can be constructed. However, WSDL does not specify which protocol to use under which circumstances for a given operation, so the proxy will need to be created using just one protocol per operation. With this mapping scheme in place, the service can be used as either a Jini service or as a web service.

Traveling on the "Jini to Web Service" side of the bridge is a more complex undertaking since it is more difficult to generate a UDDI registration and a WSDL file from a Java interface. In fact, it is highly questionable and doubtful as to whether this task can even be accomplished. As was discussed, UDDI can provide an almost

unlimited amount of information about a web service. A WSDL file contains more information about a web services' operations than a Java interface contains. Examining a Java interface will really not give any information that can be placed into a UDDI registry. Also, the types that are expressed in a Java interface as part of return values or input parameters can be very complex and it can take considerable effort to represent these within a WSDL file. The inner working of these types are often not available, so generating an XML schema for some types will not be possible. As discussed previously, WSDL has the ability to have different groups of operations be invoked over different transport protocols. By looking at a Java interface, one cannot determine the type of transport protocol that could be used, as the intention of a Java interface is to abstract away the underlying protocols that are used to invoke the service. It could even be the case that a Jini proxy used SOAP for all of its methods to communicate with the service, but one would not be able to tell that this was the case by looking at the Java interface. If the source code was available for the proxy, and it usually isn't, one could examine this code to see what protocols are used for which methods, but this really defeats the purpose of Jini, since clients should not care which protocols are used. This paradigmatic difference represents a roadblock and hinders our ability to map from WSDL and UDDI to Jini.

It should also be considered whether Jini services should even be exposed as web services. Jini services are meant to be run behind a firewall where access to a service is severely limited. If one was successful is making a web service from a Jini service, many of the transport protocols that are used in web services can penetrate firewalls. Another important thought to keep in mind is that a web service may encompass many Jini

services; Jini services are designed to represent more fine-grained functionality than a web service, and are often at a lower level of abstraction. [20] If each of these Jini services was exposed, functionality that was not meant to be exported would be. The service can also be registered within a public UDDI registry where everyone will be able to see the service and gain access to it. The current inability to map from a Jini service to a web service may be a blessing in disguise.

4 Summary

This paper has offered a comprehensive technical discussion about the nature of web services and how it can be improved. Web services has been been portrayed in a positive light as it promotes interoperability, loose coupling, and is language independent and encapsulated. However, as in many other innovative and breakthrough technologies in the current, turbulent software market, there are areas within web services that need improvement. It is imperative to have some type of failure management scheme in place in distributed technologies and web services is not an exception to this rule. One way of addressing failure management, as this paper mentioned, is to have a management model in place that will dispose of stale entries within the UDDI registry. An event mechanism is also highly recommended within distributed technologies so that consumers of a service get notified of changes within the service or within a service registration mechanism. Web services should attempt to leverage some of the work that was done in the SIENA project in order to build an efficient, robust, and flexible event notification service.

A service registry should be organized so that it supports efficient searching and ensures some level of consistency for data representation. UDDI is very flexible,

imposes no requirements on how some of its data should be organized, and thus leads to an inconsistent data representation, making it almost impossible to perform efficient queries without substantial human intervention. The importance of well documented interfaces cannot be overlooked in any technology that is meant to be accessible to other entities, and web services is no exception, as it can potentially be used by a wide range of applications across the Internet. Well documented interfaces should effectively convey to the user enough information about the web service, so that there is no doubt in the user's mind what the web service is going to do when an operation is invoked. Unfortunately, web services fall short in this category and need to provide more information in their descriptions. Web services can also leverage other distributed technologies to make up for its shortcomings such as self-healing. They can also be bridged with other service-oriented infrastructures so that different types of services can communicate with each other. In this paper, Jini Networking Technology was used as an example of a service-oriented architecture that can be combined with web services.

5 Conclusion

Although many of the core technologies that comprise web services need to be improved, web services is definitely a promising technology and has a future. It will be used to bridge platform gaps that have never been easily bridged before, and there are a wide variety of players in the market that are offering various tools to assist programmers in the development of web services. The improvements mentioned in this paper will not be considered "show-stoppers" or render web services unusable in many environments right now, but they will in some, and will certainly be factors in the future as all distributed technologies mature. In fact, all of the areas of improvements mentioned in

this paper are applicable to many other distributed technologies and not just web services. However, since interoperability within heterogeneous infrastructures is such an important factor in information technology today, these improvements carry a substantial amount of weight in the field of web services and should be addressed in the very near future.

The question that remains is the capacity in which web services should be used. When one selects a technology, there is always a trade-off involved. One must balance what the technology does and does not do with the needs of the business. Understanding what a technology does not do is often much more difficult than understanding what it does do. Yet, understanding what a technology does not do is essential to know before an investment is made, and the goal of this paper was to establish such an understanding. It really cannot be said in black and white where web services should and should not be used but a few factors should be taken into consideration. One must decide what types of functionality are important to include in their distributed application. One must also decide on the type of environment where this technology should be used. If one wants to design a distributed application that is used over the Internet and has all of the bells and whistles that a distributed application is supposed to have, web services is not the best technology to use right now. If one wants to use web services within a company Intranet and just have some bells and whistles, the drawbacks mentioned in this paper should be taken under close consideration, but web services can still be used. Essentially, the developer has been given the tools. Some tools offer too many features for someone and developers have to climb a steep learning curve, whereas others do not offer enough features, and have significant holes in several key areas. Like tools, web services give the developer an option of how and where to use them to achieve results.

Appendix: Acronym List

API – Application Programming Interface

COM – Component Object Model
CORBA – Common Object Request Broker Architecture

DAML-S – DARPA Agent Markup Language – Services
DARPA – Defense Advanced Research Project Agency
DCOM – Distributed Component Object Model
DTD – Document Type Definition

HTTP – Hypertext Transfer Protocol

JSR – Java Specification Request

RMI – Remote Method Invocation
RPC – Remote Procedure Calls

SIENA – Scalable Internet Event Notification Architectures
SOA – Service-Oriented Architecture
SOAP – Simple Object Access Protocol

UDDI – Universal Description, Discovery and Integration
URI – Uniform Resource Identifier
URL – Uniform Resource Locator

W3C – World Wide Web Consortium
WSDL – Web Services Description Language

XML – Extensible Markup Language

Bibliography

[1] Canosa, J. "Introduction to Web Services", February, 2002, http://www.commsdesign.com/design_corner/OEG20020125S0103

[2] Deitel, H.M., Deitel, P.J., Gadzik, J.P., Lomeli, K., Santry, S.E., Zhang, S. *Java Web Services for Experienced Programmers*. Upper Saddle River, NJ: Prentice Hall, 2003 pp. 24-210.

[3] Bloomberg, J., Schmelzer, R. "The Pros and Cons of Web Services.", Zapthink Research, May, 2002.

[4] Newcomer, E., *Understanding Web Services XML, WSDL, SOAP, and UDDI*. Boston: Addison-Wesley, 2002, pp. 31-174.

[5] WASP Developer Advanced 3.0.1- version 3.1.1, Systinet.

[6] Ort, E. "Web Services and the Sun ONE Developer Platform," November, 2002, http://sunonedev.sun.com/building/tech_articles/webservice.html#uddidesc

[7] Waldo, J., Wyannt, G., Wollrath, A., Kendall, S. "A Note on Distributed Computing," Sun Microsystems Laboratories, November, 1994.

[8] Edwards, W. K., *Core Jini*. Upper Saddle River: Prentice Hall, 1999. pp. 64-404.

[9] Arnold, K., O'Sullivan, B., Scheifler, R.W., Waldo, J., Wollrath, A., *The Jini Specification*. Reading, MA: Addison-Wesley, 1999.

[10] Curbera, F., Mukhi, N., Weerawarana, S. "*On the Emergence of a Web Services Component Model,"* Component Systems Group, IBM T.J. Watson Research Center, June, 2001.

[11] Bellwood, T., Clement, L., Ehnebuske, D., Hately, A., Hondo, M., Husband, Y.L., Januszewski, K., Lee, S., McKee, B., Munter, J., von Riegen, C. "UDDI Version 3.0", July, 2002, http://uddi.org/pubs/uddi-v3.00-published-20020719.htm

[12] Carzaniga, A., Rosenblaum, D.S., Wolf, A.L., "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems,* Vol. 19, No. 3, pp. 332-383, August 2001.

[13] Wallnau, K.C., Hissam, K.C., Seacord, R.C. *Building Systems from Commercial Components.* Boston: Addison-Wesley, 2002, pp.55-56.

[14] Parnas, D. "Information Distribution Aspects of Design Methodology," in *Proceeding of 1971 IFIP Congress*, Amsterdam: North Holland Publishing, 1971, pp. 339-344.

[15] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., *Documenting Software Architectures*. Boston: Addison-Wesley, 2002, pp.223-258.

[16] "Semantic Web", http://www.w3.org/2001/sw/.

[17] Anupriya, A., Burstein, M., Hobbs, J.R., Lassila, O., Martin, D.L., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T., Sycara, K. Zeng, H. "DAML-S, Semantic Markup for Web Services"

[18] Bloch, J. "JSR 175 A Metadata Facility for the Java Programming Language," http://www.jcp.org/en/jsr/detail?id=175

[19] Dearing, R. "Service-oriented Architecture Using Jini," http://searchwebservices.techtarget.com/originalContent/0,289142,sid26_gci871817,00.html

[20] Waldo, J., "Integrating Java Web Services and Jini Network Technology," Sun Microsytems, December, 2002.