Web Services: SOAP, UDDI, and Semantic Web

Justin R. Erenkrantz

ICS 221
University of California, Irvine
Irvine, CA 92697-3425
jerenkra@ics.uci.edu

Abstract

As the World Wide Web has grown, it has been a challenge to allow meaningful understanding of content on the web. To face this challenge several technologies and initiatives have been introduced under the umbrella of web services - among these are the SOAP and UDDI initiatives. This paper provides an overview of these initiatives. This paper will also discuss the challenges that may prohibit the widespread usage of these initiatives. This paper will also discuss the proposed semantic web and discuss possible challenges to its widespread adoption.

1. Introduction

The growth of the World Wide Web has created a virtual forum that allows rapid exchange of information between parties. However, there is no common manner for transferring application-specific data. The key protocols of the current web infrastructure are HTTP[13] and HTML[25]. HTTP concerns itself with how data should be transported between a server and client. HTML defines the predominate data format that is used to render text on the current infrastructure.

However, these technologies are not designed to enable meaningful communications between peers. HTTP utilizes the traditional server-client network architecture. While it may be cheap to introduce a new web server into a network, HTTP is not designed for two servers to autonomously transfer application-specific data. A HTTP server will only respond to requests from clients for the data it is responsible for.

The data format of HTML is useful for rendering a website, but it is geared towards presentation of elements on a user agent. Additionally, over the years, the number of sites containing invalid HTML has compromised the integrity of the specification. While HTML has proven that it is easy to learn, most current web browsers will leniently parse web pages. Errors in the syntactical nature of a website are usually corrected by the browser without the user's intervention.

This lack of precision makes it difficult to rely on HTML for meaningful representation of data. The data may be ill-formed which may lead to imprecise understanding of the original intent of the content. When conducting communication between peers, a shared agreement must be reached on what the data is and what it should mean. There should be no room for misunderstanding between peers.

The cloud of technology that should enable this level of peer-to-peer interaction is called Web Services. We will examine SOAP, a new initiative that defines a much stricter data format that allows integrity and allows for proper syntactic validation of the message. We will also introduce UDDI, which is a service for discovering available web services using a public directory. Finally, we will look at the Semantic Web. In addition to defining the syntax of these interactions using Web Services technologies, the Semantic Web may also be helpful to define the semantic meanings of these interactions.

2. Web Services

The current W3C Working Draft Glossary on Web Services defines Web Services as "a software system identified by a URI, whose public interfaces and bindings are defined and described using XML" [8].

The current incantation of the World Wide Web is built around passive informal interactions. Currently, the web is centered around content. It is not always possible to interact with the sources of content. Instead of interacting with Google through a web page, Google could expose their Search API and allow programmatic access to their search engine. In fact, Google has exposed their Search engines in such a manner[1].

With traditional web pages, there is no metadata that describes how to interact with a website. Competing sites may offer similar functionality using a variety of mechanisms. This presents a challenge to meaningful business-to-business integrations. It may lock in a partnership because it is too cost-prohibitive to create a new relationship with another partner even though there is a high level of dissatisfaction.

For a business partner to integrate with another business using web-based technologies, a custom bridge must be built. If one of the parties redesigns their website, the bridge may have to be rebuilt. The bridge may not be able to rely on translation from the old to the new format because the old website is removed. If the business relationship is severed and a new partner is acquired, a brand new bridge must be built because there is no shared interface with the previous partner. This makes it difficult to create and use interchangeable relationships on the World Wide Web.

Therefore, the goal of web services is to enable active well-defined interactions. It should be possible to create connectors that can withstand change to the layout intended for users. The core components should be exposed in a meaningful manner. Layering components and connectors should be supported by any web service.

There are a number of specifications that are crucial to the Web Services goal. One of the key components is SOAP, a mechanism for transferring content. Another key component is UDDI, a mechanism for discovering web services. Together, these two technologies and several others attempt to create Web Services.

3. SOAP

Simple Object Access Protocol (SOAP) Version 1.2 is defined by the W3C as "a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment" [18]. SOAP is meant to promote shared understanding of data in a way that machines can easily and correctly parse them. To achieve this goal of extensibility, SOAP uses XML as the principal data format.

While SOAP is meant to be protocol-agnostic, the specification defines protocol bindings frameworks to describe how SOAP messages are transported on the wire. Currently, most SOAP interactions travel over HTTP, but hypothetical SMTP interactions are also described in the W3C SOAP primer [23].

SOAP originates comes from the prior XML-RPC specification [26]. When SOAP is used with the HTTP protocol binding, it is functionally equivalent to XML-RPC. One of the main limitations of XML-RPC is that it has a limited type system. For example, parameter values are not ordered or labeled. This can result in ambiguity in determining the appropriate mapping of parameter values. SOAP addresses this ambiguity by leveraging XML Schema to expand the data structures that can be represented by publishing a scheme for the SOAP message[12].

SOAP consists of several components and actors that work together. A SOAP envelope consists of the data to be transmitted. Each actor is represented by a server node that has a role in processing the message that defines its behavior and responsibilities. In addition, SOAP also has an error structure that allows for graceful handling of faults. Each of these will be further discussed in detail in the following sections.

3.1. SOAP Envelope

A SOAP message consists of two portions: a SOAP header, and a SOAP body. The header serves as the metadata for the message, while the body defines the data in the message. The actual content of these sections are generally left to the application to define. However, as will be discussed later, the SOAP specification has defined what actions SOAP nodes may perform on components contained within the envelope.

SOAP Header

The SOAP Header portion of the envelope consists of multiple XML entities called header blocks. These header blocks serve to define the metadata for this transaction. Each header block may also be targeted at specific SOAP nodes by identifying the role of the node that should process it. Therefore, we can view the SOAP header as containing hop-to-hop information as well as describing the SOAP body in a end-to-end fashion.

Since the message may be transformed by intermediaries before arriving at its final destination, a *mustUnderstand* XML attribute may be included for all header blocks. The presence of this attribute indicates that if a SOAP node is targeted via the role attribute and does not recognize or understand the header, it must generate a SOAP fault.

SOAP Body

The SOAP body consists of the actual XML-formatted end-to-end data. The SOAP body should only be processed by the SOAP receiver. The syntax and semantics of this body is left undefined by the SOAP specification. The underlying application may generate a SOAP fault if the body is malformed or inconsistent.

3.2. SOAP Nodes and Roles

There are several types of participants in a SOAP transaction. Each participant has its own duties and roles as defined by the specification. Since SOAP can be modeled on the request/response network paradigm, there is a participant responsible for the origination of the message and another participant that is responsible for the responding to that message. Due to the typical protocol binding with HTTP, the SOAP specification allows for an intermediary participant that is responsible for relaying messages and possibly altering the content.

As mentioned above, each SOAP header block may include a role attribute that defines which nodes may indicates which role should process it. However, these roles do not include any routing information. The actual routing of a message between intermediaries is not defined by the SOAP specifications.

SOAP None Role

The first standard role is the *none* role. It is defined by the XML namespace *http://www.w3.org/2002/06/soap-envelope/role/none.* It is invalid for a SOAP node to participate in this role. Tagging a header with this role may be useful for including information that may not be manipulated by any intermediaries.

SOAP Next Role

The second standard role is the *next* role. It is defined by the XML namespace *http://www.w3.org/2002/06/soap-envelope/role/next.* All SOAP intermediaries must act in this role. The final recipient of the message should also act in the *next* role.

Once the relevant SOAP header is parsed by a node, it does not have to be passed to subsequent nodes. In this manner, header values tagged with this role are useful for hop-to-hop information.

SOAP Ultimate Receiver Role

The final standard role is the *ultimateReceiver* role. It is defined by the XML namespace *http://www.w3.org/2002/06/soap-envelope/role/ultimateReceiver.* Only the final recipient of the SOAP message may act as the *ultimateReceiver.* This is useful for including information that only concerns the final destination. Intermediaries may not modify header blocks that are tagged with the ultimateReceiver role. This allows end-to-end data to be preserved.

SOAP Sender

The SOAP sender is the originator of the SOAP message. It is responsible for creating the message and defining the initial construction for the message. The SOAP sender may deliver the message to either a SOAP intermediary or

the SOAP receiver. It should be realized that the SOAP sender may not be aware if it is directly sending the message to the SOAP receiver or to a SOAP intermediary. In a correctly implemented SOAP architecture, this distinction should not matter.

Since the SOAP sender is responsible for the creation of the SOAP message, this node does not act in any defined roles. However, it may tag certain SOAP header blocks with the correct roles.

SOAP Intermediary

This party receives a message from either a SOAP sender or another SOAP intermediary. It must act in the *next* role. There may be an unspecified number of SOAP intermediaries before a message reaches the final SOAP receiver.

A SOAP intermediary may be active or passive. An active intermediary will alter the content of the message to be fit the semantic definitions of subsequent nodes. A passive intermediary will not otherwise change the content of the message, but will route the message accordingly.

SOAP Receiver

This is the final destination of the SOAP message. This node is responsible for interpreting the message. If the message calls for a response, the SOAP receiver should generate a reply. The SOAP receiver acts in both the *next* and *ultimateReceiver* roles.

It is at this stage that the semantics of the message are finalized. Until the SOAP receiver is reached, there is no firm definition of what the end-result of a SOAP message will be. Application-specific errors usually will only be generated by the SOAP receiver.

3.3. SOAP Fault

A SOAP fault is generated when an error occurs during the processing of the SOAP message. A fault may be generated by a SOAP intermediary or by a SOAP recipient. A SOAP faults is separate from binding-related errors. A binding error is reported using the error mechanisms of the underlying transport protocol. When a SOAP fault occurs, no additional data may be returned. Therefore, it is not possible to return partial data and a SOAP fault in the same message.

A SOAP fault must contain a code element which describes the type of error that occurred. Furthermore, it must also contain a reason element that should provide further explanation as to why the fault was generated. Optionally, the SOAP fault may indicate the node and role where the fault originated. This is to help identify where the error occurred if the routing is not explicit. The SOAP fault may also contain a detail element which further describes the reason the fault occurred.

3.4. SOAP Transmission

SOAP is primarily meant only to represent data in a structured format. SOAP does not explicitly tie itself to a particular method of interaction. As described in [19], SOAP can be used by a variety of different asynchronous and synchronous interaction models called message exchange patterns (MEPs).

The core SOAP specification describes a protocol binding with HTTP [17]. Therefore, most uses of SOAP primarily use HTTP as a transport mechanism. The SOAP HTTP protocol binding restricts itself to two MEPs: SOAP request-response, and SOAP response to a HTTP request.

When using HTTP for the underlying protocol, a SOAP message is typically POSTed to a SOAP-aware URL. This POST body will contain the SOAP envelope with the appropriate content-type set. The HTTP server receiving this POST will then either act as a SOAP recipient or as a SOAP intermediary. In the case of the SOAP intermediary, the POST body will be forwarded to the next hop. If an error occurs at any point during the processing, the errors should be returned with an appropriate HTTP error code and, if available, a SOAP fault description.

3.5. Problems with SOAP

In theory, SOAP creates a very fine line about what it can and can not do. However, in application, the predominate use of SOAP has corrupted the integrity of its architecture. Most of these problems can be traced to architectural mismatches with the predominate protocol binding - HTTP.

Layering of resources and representations

HTTP's primary architectural style is Representation State Transfer style (REST)[14]. REST creates a distinction that resources and the representation of those resources. One representation can be translated into another representation by applying the correct content filter. These filters can then be layered on top of each other until the desired representation is achieved.

With the presence of active intermediaries, SOAP has a similar resource layering concept. Active SOAP intermediaries can translate the data into syntactically or semantically different SOAP messages as desired. SOAP intermediaries can be chained together to produce a meaningful representation. In this way, SOAP has kept the resource/representation distinction of REST.

However, SOAP is meant to be extensible so that it can be resistant to changes in the underlying representations. Yet, SOAP does not provide a strong versioning and extensibility system. If a system changes in a manner that breaks the old system, a custom bridge must still be built or the original system must be modified to work with the new system.

Idempotent operations

One of the fundamental concepts of HTTP is idempotent operations. Certain HTTP methods (such as GET) are classified as idempotent. If a GET is performed multiple times on the same resource, the results should be identical. Other methods (such as POST) are non-idempotent. If a POST is performed multiple times on the same resource, the side effects are undefined by the specification.

In practice, most SOAP interactions are performed via the POST HTTP method. Therefore, it is not possible to know whether an message will be idempotent without

acquiring semantic knowledge of what the SOAP receiver will do with the message. However, in HTTP, by only looking only at the method name, it is possible to identify whether the resulting HTTP operation will be idempotent. The protocol itself defines whether the operation is idempotent without any relationship to the resource.

This inability to rely upon idempotency presents a significant obstacle to intermediaries that wish to intelligently cache SOAP messages sent over HTTP. SOAP should allow for a mechanism to identifying messages as idempotent. Section 4.1 in [17] attempts to address this by mentioning that HTTP bindings with SOAP should be used in a manner that is friendly to the current architecture of the World-Wide Web.

Table 1 provides an example of this mismatch and the proposed alternative that promotes web-friendly behavior.Instead of using POST for idempotent requests, the request should be made using a HTTP GET. However, the request is no longer in SOAP and does not get the benefits of the structured data. Therefore, sites should measure the ability to represent the request in SOAP versus using a idempotent HTTP request.

```
Canonical SOAP example (Example 12a in
[23]):

POST /Reservations HTTP/1.1
Host: travelcompany.example.org
Content-Type: application/soap+xml...
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope...>
  <env:Body>
    <m:retrieveItinerary...>
      <m:reference>
        FT35ZBQ
      </m:reference>
    </m:retrieveItinerary>
  </env:Body>
</env:Envelope>

Web-friendly alternative (Example 12b in
[23]):

GET /Reservations/itiner-
ary?record=FT35ZBQ  HTTP/1.1
Host: travelcompany.example.org
Accept: application/soap+xml
```

Table 1: Soap Mismatch Example

Use of SOAP Envelope

The use of a separate SOAP envelope collides with the notion of separation of HTTP metadata and data. The SOAP protocol binding contains the entire SOAP envelope in the body of the HTTP request. Therefore, in order to properly parse the request, a SOAP intermediary must examine the entire body of the HTTP request. A HTTP proxy can operate only be examining the metadata of the request.

A better solution would allow alteration of how a SOAP message is delivered based on the underlying protocol. Protocols that already provide for a metadata/data separation should have those mechanisms utilized by SOAP bindings. A HTTP protocol binding including SOAP headers in the HTTP headers while including the SOAP body in the HTTP body would be more efficient. This change would allow the SOAP intermediary to route the message based only on the HTTP headers.

Two-level naming system

Additionally, the SOAP HTTP binding also suffers from a two-level naming system. In Table 1, the message is POSTed to the /Reservations resource. Inside of the SOAP envelope, it indicates that the *retrieveItinerary* method should be executed. This makes it difficult to determine what the actual function is without understanding the body.

In this example, any intermediary that attempts to route a message would have to understand the entire /Reservations namespace. The granularity of the naming system is insufficient to allow an intermediary to only intercept retrieveItinerary requests without looking at the body of the SOAP request. However, doing so, may be in violation of the SOAP specification.

As indicated in Table 1, a potential solution is to utilize the SOAP response message pattern described in Section 6.3 of [17]. A request would be a HTTP request with no SOAP components, while the response would be a SOAP response embedded inside of a HTTP request. However, this leads to an asynchronous method of operation. Rather than replying with a HTTP response, a SOAP message would be in the response.

This solution allows regular HTTP proxies to cache the request using its normal mechanisms. However, this solution may cause problems for SOAP intermediaries. A SOAP intermediary would have to have two methods of interactions - a SOAP proxy and a HTTP proxy. This may lead to significant overhead for implementors of a SOAP-aware proxy.

If an active intermediary intends to rewrite the normal HTTP request, it must rewrite the request using the HTTP syntax.

Efficiency of XML

SOAP's use of XML allows for packaging of data in a well-defined format. However, XML is not meant to be a high-performance transport mechanism[11]. Rather XML serves the purpose of being both moderately human-parseable and precisely computer-parseable. XML does not define the semantics of the message - that is left to the application to define.

Properly-formed XML may be easily verified by the human eye without knowing the semantic meaning of the message. Knowledge of the generic syntactical structure of XML is all that is required for a computer to validate an XML document. SOAP is trading off the extensibility of XML for a potential loss in performance.

Additionally, due to the hierarchical nature of XML, it may be inefficient to parse a message if there is an interest only in a segment of that message. In order to properly validate a segment of a message, the entire message may have to be validated. A SOAP message might include two SOAP

header sections. These types of abnormalities should be detected as early as possible, but if a parser were to stop after seeing the first SOAP header section, it would not detect the error.

Furthermore, XML is inefficient for binary transport. As a partial solution, XML does allow for CDATA elements. These elements are not meant to be parseable by an XML parser. However, there are certain character sequences that are still invalid within a CDATA element. Furthermore, even when the length of the binary stream is known ahead of time, depending upon the implementation, the XML parser may have to parse character-by-character. If the language could take advantage of known binary lengths, this might allow for reduction of inefficiencies.

Without modifying the XML specification, transporting binary content within SOAP can be addressed by adding a level of indirection. Instead of including a binary representation of a picture in a SOAP message, one would include a URL for this picture. If the recipient is interested in the picture, the recipient would fetch the picture from the URL. However, this increases the number of required round-trips to fetch all components of the message.

In XML, all parseable content must be properly escaped. For example, certain characters (such as '<' or '&') are not valid. Therefore, any content which would otherwise contain these characters must be escaped. This may introduce additional processing to verify that all content is syntactically proper before transmission.

mustUnderstand header

In a SOAP message, there is a *mustUnderstand* attribute that may be attached to an element. If a SOAP intermediary or recipient does not understand the element which contains this attribute, it must generate a SOAP fault. However, it does not mean that there is shared semantic meaning as to what the value means.

For example, a SOAP sender could intend for a value to mean one thing while an intermediary or the final recipient understands that value to mean something else. While SOAP does not present to rectify semantic conflicts, it still is susceptible to semantic mismatches even when syntactical equivalence is achieved.

3.6. SOAP Example

Table 2 provides an example of a SOAP request for the price of a catalog item, and a cacheable response with the associated price[19, Examples 24 and 25]

Request:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://
www.w3.org/2001/09/soap-envelope">
  <env:Body>
    <c:CatalogPriceRequest
xmlns:c="http://example.org/2001/06/
catalog">
      <c:PartNumber>ABC-1234</c:PartNum-
ber>
    </c:CatalogPriceRequest>
  </env:Body>
</env:Envelope>
```

Response:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://
www.w3.org/2001/09/soap-envelope">
  <env:Header>
    <ca:CacheControl xmlns:ca="http://
example.org/2001/06/cache">
      <ca:CacheKey>ABC-1234</
ca:CacheKey>
      <ca:Expires>2001-03-09T08:00:00Z</
ca:Expires>
    </ca:CacheControl>
  </env:Header>
  <env:Body>
    <c:CatalogPriceResponse
xmlns:c="http://example.org/2001/06/
catalog">
      <c:PartNumber>ABC-1234</c:PartNum-
ber>
      <c:PartPrice c:cur-
rency="USD">120.37</c:PartPrice>
    </c:CatalogPriceResponse>
  </env:Body>

</env:Envelope>
```

Table 2: SOAP Example

4. UDDI

The Universal Description Discovery and Integration (UDDI) system defined as "a set of services supporting the description and discovery of (1) businesses, organizations, and other Web services providers, (2) the Web services they make available, and (3) the technical interfaces which may be used to access those services."[5]

The Universal Description Discovery and Integration (UDDI) system is built as a mechanism on top of SOAP. Its primary goal is to allow multi-organizational collaboration of web services. Without a good naming and discovery system, it can be hard to find the services one is looking for. Therefore, by promoting a Web Services repository, awareness can be increased of currently available web services.

Each organization can publish information about its web services to an UDDI server or node. This information should be enough to allow access of the web service. Other organizations can then either subscribe to changes in the repository, or search the repository based on keywords in the web service description. Since UDDI is meant to be universal in its reach, it also has support for internationalization.

UDDI support clouds of nodes forming together to create a registry. While UDDI can work with only a single server, it is designed to allow for replication and synchronization between multiple UDDI nodes. This allows for distribution and fault-tolerance within the UDDI framework. There has been substantial work on creating a UDDI Business Registry that acts as a public UDDI repository[2].

4.1. Key entities

UDDI defines four key objects that compose the data

stored within an UDDI registry. They are *businessEntity*, *businessService*, *bindingTemplate*, and *tModel*. As discussed below, these objects are arranged in a hierarchical relationship. Furthermore, each object can only be owned by a single publisher at a time.

businessEntity

The *businessEntity* object describes the participants in the registry[5]. It contains the basic information for contacting the entity represented in this object. It also contains information for categorizing the entity using conventional business taxonomies. An example of the data contained may be a physical mailing address, electronic mailing address, or website. The *businessEntity* also serves as the parent to the rest of the UDDI data objects.

The ***businessEntity*** object does not necessarily need to represent an entire corporation. If the registry is private to a corporation, it may make sense to scope the *businessEntity* objects to departments. However, when publishing to the public UDDI Business Registry, the *businessEntity* should represent an entire corporation.

businessService

The *businessService* object describes the class of web services that a *businessEntity* offers[5]. A *businessService* may contain many different *bindingTemplate* objects. Each *businessService* is the child of only one *businessEntity.*

The *businessService* provides a logical grouping for related web service offerings that an organization provides. For example, all of the ordering-related web service components can be grouped under one *businessService* object, while the shipping-related web-service components can be grouped separately.

bindingTemplate

A ***bindingTemplate*** object refers to an individual web service[5]. A ***bindingTemplate*** is a child of one *businessService*. Each *bindingTemplate* is associated with one *tModel*.

This object contains the actual information required to access the web service. However, it does not define the implementation semantics for this web service. A *bindingTemplate* only provides the URL at which the particular instance of this web service can be reached.

tModel

A *tModel* object specifies the contract that a particular web service will operate under [5]. This technical model may describe the web service using WSDL[9] or XSD [12]. A *tModel* may be referenced by many *bindingTemplate* objects that share a common interface, but live a different locations. It should be noted that the *tModel* does not contain the actual binding information, but a link to the authoritative binding information for that web service.

4.2. Actions

The following actions can be performed on a UDDI registry: inquiry, publication, security, custody transfer, subscription, replication. These interfaces are implemented using SOAP with its HTTP protocol binding.

Inquiry

Due to the large nature of the registry, it should be possible to perform queries on the registry. Therefore, the inquiry API is used to locate and obtain detail on entries in the registry[5]. Users may browse the registry using this interface. They may also drill-down on entries to examine details or children of the entities. Additionally, users may perform queries on the registry.

The inquiry API provides mechanisms for searches using partial information concerning a registry entity. The system can then perform wildcard searches on this data and return any results that match the qualifiers. The UDDI inquiry API also provides a mechanism to control sorting of the returned results.

Publication

The publication API is used to create or modify information in a UDDI registry[5]. Depending upon the security policy of the UDDI registry node, any publisher may be able to create a new *businessEntity* object. Once the ***businessEntity*** is created, its children nodes can be added by authorized publishers. Modifications to the registry entries are handled through this interface.

Security

The security API provides for the ability for users to prove their identity to the UDDI registry and for the UDDI registry to maintain awareness of that user[5]. Upon successful credential authentication, the system will generate an opaque token that can be provided to the other UDDI APIs. The system allows for expiration of the tokens by the system. The user may also request that the token be discarded by the registry to effectively log out of the system.

Most of the other UDDI APIs require the presence of this authentication token to modify the registry. If the individual represented by the token is authorized for performing this operation, then the registry will allow the transaction to commence. Otherwise, a security violation will be reported.

Custody Transfer

As described above, most objects in the registry may only be owned by one publisher. Therefore, a Custody Transfer API exists to allow migration of ownership between publishers[5]. Once a publisher has transferred custody of an object, it can no longer issue modifications to the UDDI registry for that object.

Subscription

UDDI allows for monitoring of resources in the registry [5]. A user can request to be notified whenever an addition, modification, or deletion occurs. The subscription can also have qualifiers to indicate which entries should be monitored.

These subscriptions may occur at any level of the entity hierarchy. Therefore, a request to be notified about all changes for a particular *businessService* can be issued. If any *bindingTemplates* or *tModels* underneath this *businessService* are altered, an event would be generated.

A subscription can last for a finite duration. When creating the subscription, a user may indicate when they wish to

begin receiving notifications. In addition, the user may also issuing a stop date for notifications. As a convenience, the interface allows for renewal of prior subscriptions.

UDDI notifications may occur either synchronously or asynchronously. Under the synchronous model, the user saves subscriptions on the server. Upon the demand of the client, the notifications that they have not yet received are returned.

Asynchronous events can also be handled via the UDDI infrastructure. When subscribing to an event using the asynchronous interface, the user provides a subscription listener routine. This callback routine may either be an email address or a HTTP URL with a UDDI-defined SOAP service running on it. Rather than issuing events in real-time, a UDDI node may choose to queue events up and periodically issue the asynchronous notifications.

Replication

One of the interfaces that is not primarily meant for user interaction, but for server interaction is the replication API. The UDDI specification allows for servers to cooperate and spread the load amongst many physical servers[5]. UDDI allows for multiple-master replication - where each server allows for modification of its data. Then, the changes are replicated across the UDDI registry nodes in a controlled fashion.

In the UDDI replication scheme, each server should maintain a complete copy of the registry. Whenever a change occurs, a notification of change is emitted to the peer servers. When a node receives this notification, it can then pull the changed data from the node that sent the change notification.

4.3. Challenges to UDDI

There are several challenges and issues with UDDI that may affect its acceptance and chances for long-term survival. The first is that UDDI has a non-uniform security model. The second is that UDDI may be limited by its ability to only have single ownership of an entity. Finally, the subscription model may place an undue burden upon the clients of the UDDI system.

Non-uniform security model

One key concern about UDDI is that it does not guarantee a common security policy across all registries [5]. Each node is free to setup its own security policy irrespective of the other nodes' security policies. If the servers are under decentralized control, this may present a challenge to maintaining the integrity of the UDDI registry. UDDI attempts to resolve this problem by introducing root and affiliated registries[5].

A root server can delegate portions of the domain space to other servers. These delegates are then authoritative for that domain. Each server is then responsible for maintaining internal consistency of its data. However, the data stored on that server is replicated to all other servers in the registry. Furthermore, most of the discussion in the UDDI specification deals with ensuring that no duplicate keys exist across registries rather than true delegation and separation.

UDDI also has an interface for retrieving the security policy of the current node. When publishing the details of an entity, the publisher is recommended to be aware of what the policy is on all other nodes. A UDDI node in a registry could exist that allows anyone to update the contents of the registry. Therefore, this site could be used to alter or overwrite the correct registry data.

However, checking each individual server may be an infeasible approach as the number of nodes scales. A publisher would have to ensure the security policy of each node before publishing. Therefore, the best policy may be to artificially limit the number of nodes to only those that are operated by legitimate and trustworthy businesses with appropriate security policies.

However, even potentially unsafe entities might be able to participate in a replication scheme that had true delegation. A model similar to DNS[24] might be a better system for a naming system. In DNS, caching and authoritativeness are handled in a well-defined and scalable manner. The DNS replication model does not require complete duplication of all information on each nodes, but uses a caching algorithm to gain similar performance effects without sacrificing authoritativeness.

Single ownership

Another concern about UDDI is that entities can only be owned by a single publisher at any point in time. It is not possible to have two businesses collaborate on the ownership of an object. One goal of Web Services is to promote relationships between businesses. A potential collaboration would be the development of a web service. Yet, UDDI does not allow for this to occur.

A possible solution to this problem is to select one of the businesses to be responsible for the shared object. All changes to the resource must be made by an authorized representative of the responsible party. This impairs the ability of the other party to modify the resource. If both parties are equally responsible, this delegation may not be acceptable.

However, another option is to create a virtual entity that representations the collaboration. Both parties can then have appropriate representation. However, the business entities as viewed in UDDI would no longer reflect the reality. The business entity would not exist outside of the UDDI registry - it would only exist to allow both parties to share responsibility for their joint venture.

Furthermore, it is questionable whether such an approach would successfully scale. Would a company that interacts with a lot of other companies have to create individual businessEntity objects for each collaboration? If so, then the worthiness of this approach should be placed in doubt.

Duration and history of subscriptions

Another concern about UDDI is the fact that its subscriptions have a limited duration. As a matter of policy, a UDDI node may place an upper bound on the duration of subscriptions. Therefore, a subscription must be constantly renewed to be in effect.

This requires the client being able to manage the state of

its subscriptions internally. It must be able to determine when the subscriptions are going to expire and renew them before expiration occurs. This may place a burden on the client to verify its subscription state periodically.

A potential rationale for this decision is that it requires validation that the notification is still desired. If the party is no longer interested, it may let the subscription expire without renewal. This may lower the activity on the server to process events that are not actively required anymore. In open systems, this might be a concern if it is expensive to generate the events.

A parallel concern that is not addressed by the UDDI specification is whether subscriptions are only valid on the single node, or on the entire registry. If the subscription is made to an individual node in a UDDI registry cluster and that node is taken out of service, are the subscriptions still available? Or, must a UDDI client expect that its servers may not always be available and able to recreate the subscriptions on new nodes? It may also be possible for some nodes to have a disproportionate amount of subscriptions that it must handle. A better scheme may be to balance the subscriptions across the entire registry.

In the UDDI specification, synchronous notification requests will return all unseen events. Each time a synchronous event is fetched from the server, it is marked as seen. On subsequent requests, those events will not be sent to the user. However, this places a burden on the server to maintain which synchronous events the client has not yet requested.

4.4. UDDI Example

Table 3 gives an example[5, Appendix G.1] of a UDDI transaction that searches for all businesses that start with the name ABC and returns two results - one for ABC Consulting and another for ABC Contractors.

5. Semantic Web

In addition to the umbrella of Web Services, there is also work on creating a Semantic Web. The W3C working group defines the Semantic web as way to "bring to the Web the idea of having data defined and linked in a way that it can be used for more effective discovery, automation, integration, and reuse across various applications."[22]

Similarly to Web Services, the Semantic Web is looking to enhance the interaction of sites. If Web Services could be viewed as the syntactic agreement of how web sites can interact, then the Semantic Web is an agreement on what is being transferred. Therefore, in addition to agreeing on a common format of how data should be transferred, there is also a contract as to the meaning of the transferred data.

By creating shared agreement of what the data actually represents, this would allow true understanding of content on a website. Any possibility of ambiguity or confusion would be removed because the meaning of any representation is predetermined. Therefore, even with a Web Services-enabled server, it may not always be possible to reach agreement on what the returned data means. Only a Semantic Web-enabled server would allow for agreement on what

Request:

```
<?xml version="1.0" encoding ="UTF-8" ?>
<find_business xmlns="urn:uddi-
org:api_v3" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
  <findQualifiers>
    <findQualifier>
      uddi:uddi.org:findQuali-
fier:approximateMatch
    </findQualifier>
  </findQualifiers>
  <name>ABC%</name>
</find_business>
```

Response:

```
<?xml version="1.0" encoding ="UTF-8" ?>
<businessList xmlns="urn:uddi-
org:api_v3" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
  <businessInfos>
    <businessInfo businessKey="1234-56">
      <name>ABC Consulting</name>
    </businessInfo>
    <businessInfo businessKey="1234-67">
      <name>ABC Contractors</name>
    </businessInfo>
  </businessInfos>

</businessList>
```

Table 3: UDDI Example

the content transferred actually means.

5.1. Overview of Semantic Web Technology

Like Web Services, there is no single specification that defines the Semantic Web. Also like Web Services, most of the relevant work on the Semantic Web is being backed by the W3C. The initial Semantic Web roadmap was issued by W3C founder Tim Berners-Lee[6]. The W3C is also producing a variety of recommendations for technologies that can enable a Semantic Web.

The majority of current activity is in the area of web ontologies. An ontology can be defined as "description of the concepts and relationships that can exist for an agent or a community of agents"[16]. Automatic discovery of ontologies in a computer setting has been a long time focus of artificial intelligence researchers. So far, they have achieved a substantial body of knowledge[10]. Furthermore, ontologies has much deeper origins within mathematical logic.

So far, the W3C has produced a recommendation for a web ontology built around the Resource Description Framework (RDF)[21]. RDF can be represented in a variety of ways, but the preferred delivery mechanism is XML[4]. RDF serves as a language to describe metadata that describes a resource available on the web[20]. Work has been done to create a core vocabulary for RDF that all websites can share[7].

For example, information describing the ownership of a resource can be expressed in RDF. In a Semantic Web-enabled world, an analysis can be done on all documents to determine what other pages a particular author has created. Additional information about the author may also be

retrieved.

A long-term goal of the Semantic Web is to describe the actual content rather than resource metadata properties. Content should move away from being described using free-form natural languages. Instead, content should be written using the structured nature of RDF.

For example, an organizational chart could be entirely described using RDF. In order to present the data, a RDF user agent could display the chart using a graphical medium. Additionally, RDF inference engines could be used to analyze the relationships between employees. Queries could be placed to determine the precise nature of relationships between sets of employees.

5.2. Challenges of the Semantic Web

There are two major classifications that challenge the viability of a Semantic Web. The first one is that it is not a small technical feat to create such ontologies. However, the more serious challenge arises from a social perspective. Therefore, it may be possible to address the technical challenges, but it may not be possible to address the social challenges inherent in creating a Semantic Web.

Ambiguity in Natural Languages

The technical challenges facing the Semantic Web are related to the ambiguity inherent in any natural language. If a machine is to understand the content of a website, it must be able to correct place the words in its proper context. However, this is a significant challenge that has so far resisted efforts by research and industry.

The approach taken by RDF is to restrict the language to a structured format. However, this is not a natural way to express content. RDF requires introducing formalism to content where there may not be a precise categorization. Categorization of elements may be a subjective process where individuals could differ on how an item should be classified even in the same ontology. Relying upon humans to perform the classification may be introduce errors.

RDF also requires content creators to express properties in a specific format rather than allowing for automated discovery of the properties. Currently, content on the World-Wide Web is expressed in a free-form manner using HTML. HTML does not attempt to define a formal language from which inferences can be drawn. Instead, HTML is meant for presentation of content.

Additionally, the web infrastructure allows for publication of content in many different natural languages. It may not be possible to achieve equivalent semantic representations between languages. Some concepts may not transfer correctly from one culture to another. While this may not be a serious problem, it is a possible obstacle.

Multiple Ontologies

Another consideration is who defines the ontology that is used. If the ontology is not consistent between sites, then true semantic communication can not be achieved. There must be a shared form of communication. Currently, RDF only describes the structure of the ontology, but it does not define the ontology. This flexibility allows for the creation of custom ontology schemas. However, that comes at the price since not everyone will agree on what should be contained within a RDF schema.

Without a shared ontology used by all sites, the situation may not be any better than it currently is. Each site would be free to define the ontology using their particular model. While the syntax of the ontology would be well-defined, unless the same ontology is used, they again may mean different things.

Tim Berners-Lee has suggested that there be a creation of RDF translators that can handle this job[6]. RDF descriptions should be evolvable and able to be mechanically translated from one format to another. However, translations of this sort have not met with success. In the area of software architectures, it has been concluded that it is often hard to conduct translations between architecture description languages[15]. While these languages are in the same problem domain, it is not always possible to achieve the bi-directional translation as mandated by the Semantic Web.

Unintentional Non-Participation

A larger concern for the Semantic Web is how it deals with non-participation. For various reasons, sites may not decide to publish their data in the relevant formats. If the Semantic Web can not encompass all of the available content, then there may be a significant omission of knowledge.

One reason for non-participation is that pre-existing sites may not update their content to follow the recommendations required to creating a Semantic Web. A major advantage to the current web infrastructure is its low barrier to entry. It does not require a lot of technical training to publish a web site. A Semantic Web-enabled site may require a higher level of administrative expectations that prohibits hobbyists from publishing on the web. This would create an exclusionary web infrastructure that goes against a key principle of the current infrastructure.

There are two possible solutions to this: creating freely available tools or allowing natural language harvesting of data. If the tools to create a Semantic Web-enabled site are free and of sufficient quality, then the barrier of entry may be lowered. However, this assumes that the creator of the content has the necessary incentive to update their current content. If the perceived advantages do not outweigh the time to update the content, the changes will not occur even if the tools are free.

Another possible solution is to allow semantic capture of data. A spider would crawl the website and infer the semantic content from the original content. However, current efforts to capture semantic knowledge from natural language is not thorough enough to serve the needs a Semantic Web.

Intentional Non-Participation

Even if the site has the technical expertise to update their site and there is a perceived benefit, it still may not be in everyone's best commercial interest to switch to a Semantic Web. It may make sense to withhold some of the content or limit the participation to parties that are not really peers.

Currently, both Amazon[3] and Google[1] provide

SOAP interfaces to their backends. In the license agreement for both interfaces, usage of the system can not be done by commercial parties. As a way to enforce this, both systems place an artificial limitation on how often a user may issue Web Services request. This limits the amount of information that can be retrieved using this interface.

However, there is a difference between allowing data to be accessed versus presenting a formal semantic representation. If Amazon made semantic information about their books publicly available, then it would be possible to for a competitor to leverage this work. A competitor may then write an RDF translator to convert Amazon's data to their own system. Therefore, it may make business sense for companies to withhold critical information from a Semantic Web to maintain an advantage over their competitors.

6. Conclusion

The technologies presented here are a first step to enabling communications between peers in a web-like infrastructure. However, there are many obstacles to making these types of interactions a reality. Currently, the technologies are still only focused on solving accidental syntax problems rather than essential semantic problems. The specifications have allowed for a tradeoff between flexibility and standards. These specifications are so flexible that true semantic relationships have yet to emerge.

Two web services that implement the same functionality may still have different interfaces. SOAP merely allows for the separation between display of content from the content itself. Current technologies like HTML do not create this level of separation. Therefore, just removing the element of display from content is a significant win. However, SOAP by itself does not allow for inherent migration from content providers.

UDDI is a necessary technology that is required to make the adoption of Web Services widespread. Without a good naming system, Web Services would become unmanageable. Yet, UDDI suffers from scalability issues that would essentially cripple it if it gained widespread acceptance that it seeks. Issues of security, replication, and subscription would overwhelm a widespread UDDI infrastructure.

The Semantic Web tries to make semantic mismatch a slightly easier problem to handle. Yet, there is still no consensus on how to define true semantic meaning. For example, there is no RDF format that everyone agrees can fully describe a document. Each individual can come up with their own RDF schema. The RDF specification allows for a RDF schema to be published, but it does not specify one true schema. The Semantic Web technologies may make the problem of addressing semantics easier, but it only allows the infrastructure to be built to support this rather addressing it directly.

The question remains whether components on the web can truly become replaceable components. Are the interfaces going to become well-defined to the point where search engines can be plugged in using the same SOAP interface? Can a book vendor be replaced at will without loss of information? The current guess leads to the assumption that this will not occur. Technologies like SOAP may make it a little easier to switch from one provider to another, but changing providers will still incur a heavy cost. The outlook for a silver bullet to achieve semantic matching is quite dim.

References.

[1] *Google Web APIs*. <http://www.google.com/apis/>, Google, HTML, 2002.

[2] *UDDI Business Registry.* <http://www.uddi.org/register.html>, UDDI, HTML, 2002.

[3] *Amazon Web Services.* <http://www.amazon.com/webservices>, Amazon, HTML, 2002.

[4] Beckett, D. *RDF/XML Syntax Specification.* <http://www.w3.org/TR/rdf-syntax-grammar/>, World Wide Web Consortium (W3C), HTML, 2002.

[5] Bellwood, T., et al. *UDDI Version 3.0*. <http://uddi.org/pubs/uddi-v3.00-published-20020719.pdf>, World Wide Web Consortium (W3C), PDF, 2002.

[6] Berners-Lee, T. *Semantic Web Roadmap.* <http://www.w3.org/DesignIssues/Semantic>, World Wide Web Consortium (W3C), HTML, 1998.

[7] Brickley, D. and Guha, R.V. *RDF Vocabulary Description Language 1.0: RDF Schema.* <http://www.w3.org/TR/rdf-schema/>, World Wide Web Consortium (W3C), HTML, 2002.

[8] Brown, A. and Haas, H. *Web Services Glossary.* <http://www.w3.org/TR/ws-gloss/>, World Wide Web Consortium (W3C), HTML, 2002.

[9] Christensen, E., et al. *Web Services Description Language 1.1*. <http://www.w3.org/TR/wsdl>, World Wide Web Consortium (W3C), HTML, 2001.

[10] Clark, P. *KBS/Ontology Projects Worldwide.* <http://www.cs.utexas.edu/users/mfkb/related.html>, HTML, 2002.

[11] Dodds, L. *Intuition and Binary XML*. <http://www.xml.com/pub/a/2001/04/18/binaryXML.html>, XML.com, HTML, 2001.

[12] Fallside, D. *XML Schema: Primer.* <http://www.w3.org/TR/xmlschema-0/>, World Wide Web Consortium (W3C), HTML, 2001.

[13] Fielding, R., et al. *Hypertext Transfer Protocol -- HTTP/1.1.* Internet Engineering Task Force, Request for Comments Report 2616, June, 1999.

[14] Fielding, R. and Taylor, R.N. Principled Design of the Modern Web Architecture. In *Proceedings of the International Conference on Software Engineering (ICSE 2000).* p. 407-416, Limerick, Ireland, June, 2000.

[15] Garlan, D., Monroe, R., and Wile, D. ACME: An Architecture Description Interchange Language. In *Proceedings of the CASCON '97.* IBM Center for Advanced Studies. Toronto, Ontario, Canada, November, 1997.

[16] Gruber, T. *What is an Ontology?* <http://ksl-web.stanford.edu/kst/what-is-an-ontology.html>, Stanford Knowledge Systems Lab, HTML, 2001.

[17] Gudgin, M., et al. *Simple Object Access Protocol (SOAP) 1.2: Adjuncts.* <http://www.w3.org/TR/SOAP/soap12-part2/>, World Wide Web Consortium (W3C), HTML, 2002.

[18] Gudgin, M., et al. *Simple Object Access Protocol (SOAP) 1.2*. <http://www.w3.org/TR/SOAP/soap12-part1/>, World Wide Web Consortium (W3C), HTML, 2002.

[19] Ibbotson, J. *Simple Object Access Protocol (SOAP) 1.2: Usage Scenarios.* <http://www.w3.org/TR/xmlp-scenarios/>, World Wide Web Consortium (W3C), HTML, 2002.

[20] Klyne, G. and Carroll, J. *Resource Description Framework: Concepts and Abstract Syntax.* <http://www.w3.org/TR/rdf-concepts/>, World Wide Web Consortium (W3C), HTML, 2002.

[21] Lassila, O. and Swick, R.R. *Resource Description Framework Model and Syntax Specification.* <http://www.w3.org/TR/REC-rdf-syntax/>, World Wide Web Consortium (W3C), HTML, 1999.

[22] Miller, E. *Semantic Web Activity Statement.* <http://www.w3.org/2001/sw/Activity>, World Wide Web Consortium (W3C), HTML, 2002.

[23] Mitra, N. *Simple Object Access Protocol (SOAP) 1.2: Primer.* <http://www.w3.org/TR/SOAP/soap12-part0/>, World Wide Web Consortium (W3C), HTML, 2002.

[24] Mockapetris, P. *Domain Names - Implementation and Specification.* Internet Engineering Task Force, Request for Comments Report 1035, November, 1987.

[25] Pemberton, S., et al. *XHTML 1.0*. <http://www.w3.org/TR/html/>, World Wide Web Consortium (W3C), HTML, 2002.

[26] Winer, D. *XML-RPC Specification.* <http://www.xmlrpc.com/spec>, Userland, HTML, 1999.