5-2009

A Framework For Transitioning Enterprise Web Services From XML-RPC to REST

Sean Kennedy
Athlone Institute of Technology, skennedy@ait.ie

Owen Molloy
National University of Ireland, owen.molloy@nuigalway.ie

51. A FRAMEWORK FOR TRANSITIONING ENTERPRISE WEB SERVICES FROM XML-RPC TO REST

Sean Kennedy
Athlone Institute of Technology, Athlone, Co. Westmeath, Ireland
skennedy@ait.ie

Owen Molloy
Dr. Owen Molloy, National University of Ireland, Galway, Ireland.
owen.molloy@nuigalway.ie

Abstract

Web Services are defined by the W3C as "a software system designed to support interoperable machine to machine interaction over a network". There are however, several alternatives as to how Web Services can be implemented: WS-* and Plain Old XML (POX) are popular approaches that markup their RPC (Remote Procedure Call) based payloads with eXtensible Markup Language (XML). Both approaches can use HyperText Transfer Protocol (HTTP) for transferring their messages. Representational State Transfer (REST) is an alternative approach that is gaining in popularity. This research-in-progress paper presents the issues of XML-RPC based Web Services (XML verbosity and message opacity) and why a RESTful approach solves these issues. We present results which show the improved performance. We present a framework that outlines a translation from XML-RPC to RESTful format for "*read*" style request messages. This framework is ideally suited to enable enterprises to gradually transition from XML-RPC to RESTful Web Services.

Keywords

SOA, Web Services, WS-*, REST, POX, XML.

1. Introduction

Service Oriented Architecture (SOA) is an architectural approach to building application systems whereby the focus is on loosely coupled sets of services which can be invoked over a network. In an SOA, software services are exposed over a network via well defined service contracts/interfaces. This abstracts the consumer of that service from knowing or caring how that service is implemented i.e. it allows applications to share data and invoke each other even if their operating systems and/or programming languages differ. This gives the consumer the flexibility to pick and choose the required service independent of the implementation (Graham et al. 2005). Web services is one approach to implementing an SOA.

There are several alternatives to implementing Web services – WS-*, POX and REST. Both the WS-* and POX encapsulate XML-RPC style Web service invocations in XML and only use HTTP POST when invoking Web services over HTTP (discussed further in section 2.1). Both approaches POST to a gateway URI where the XML must be parsed to figure out the Web service name and parameters.

WS-* wraps the invocation in SOAP (an XML messaging protocol). SOAP has an envelope which contains (optional) headers and a mandatory body section. SOAP achieves message extensibility i.e. Quality of Service (QoS) via its headers. WS-* binds to any transport, with HTTP the default.

POX is very similar to a SOAP message except that it contains no SOAP elements. Thus a POX message is smaller is size to a SOAP message but POX has no QoS support whereas SOAP has. POX typically runs over HTTP also.

Another alternative is REST. REST is an architectural style which defines constraints to induce certain system properties e.g. scalability. Two of these constraints are: unique URI's and a Uniform Interface (discussed further in section 2.3) While REST does not mandate a specific protocol, HTTP is an instantiation of REST principles. RESTful approaches use the core HTTP verbs GET, PUT, POST and DELETE to retrieve, update, add and delete resources respectively.

In this paper, we investigate normal Web service invocation i.e. with regard to SOAP we are referring to SOAP messages *without* any QoS headers. This paper is based on HTTP being the data transfer protocol.

Our hypothesis is, that when invoking XML-based *read*-type Web Services there is no need to encapsulate the message in verbose XML constructs while at the same time abusing HTTP POST.

Encapsulating the message in XML (be it POX or SOAP) increases the message footprint. Given that an HTTP GET message has no entity (or message) body, we believe that a framework to translate read-style SOAP/POX messages from HTTP POST with an XML payload into HTTP GET will reduce the message size.

Little states in (Little 2008) that "*Web services uses HTTP for a really good reason, and it's so that you can tunnel through firewalls*". This is referred to "tunnelling" and is not how HTTP was intended to be used. Using POST to carry all Web service invocations means that Web intermediaries e.g. caches and proxies are unable to inspect the message (Costello 2002). Converting POST to a RESTful GET will make the message visible to Web intermediaries. This would enable caches with their inherent efficiency.

Conditional-GET is attractive because if the resource has not changed on the server since the clients' last request (for the same resource), there is no message body sent back to the client.

Thus, our framework will have the following contribution (when compared to SOAP/POX implementations):

- Reduced request message size (due to the lack of XML content)
- Reduced number of messages in the network and faster response times (due to native HTTP caching)
- Reduced number of response messages (due to Conditional GET)

The server must accomplish the following:

a) Implement caching i.e. mark the relevant responses as cacheable i.e. stock quotes would not be cacheable whereas TV guides may be
b) Implement Conditional GET i.e. determine if a message body should or should not be sent back i.e. has the resource changed
c) Host a mapping tool which allows the user to state which of the XML-RPC methods are to be transformed to HTTP GET

The remainder of this paper is organised as follows: Section 2 gives an overview of the technologies, Section 3 is related work. Section 4 describes the framework. Section 5 presents the test environment and discusses the results. Section 6 is the Discussion section. Lastly, Section 7 outlines Future Work.

2. Web Services

As stated in the abstract, Web Services are defined by the W3C as "a software system designed to support interoperable machine to machine interaction over a network". Both SOAP and POX use XML-RPC encapsulated in XML as their paradigm for Web service execution.

2.1. SOAP

Interaction with WS-* Web services is via SOAP messages. SOAP is simple, flexible and extensible. As it is XML based, SOAP is programming-language, platform and hardware independent. SOAP 1.2 came into being in June 2003 (Graham et al. 2005). SOAP 1.2 introduced the *WebMethod* property which allows you to change the HTTP method used. However, as Tilkov states in (Tilkov 2004), "the Web Services (WS-*) approach tunnels everything through POST...that's not true for SOAP 1.2 in theory, although in practice everybody still does it". As well as that, the URI is still common to all services at that endpoint in SOAP 1.2. This means that even though the HTTP verb may now be correct, how can an intermediary determine what resource is being accessed when the resource is still embedded in an XML envelope? It is important to note that because WS-* is "protocol agnostic" it cannot exploit the advanced features of HTTP e.g. caching (Tilkov 2004).

2.1.1. SOAP Messaging Model

A SOAP message is an XML document which consists of a SOAP envelope which contains optional headers and a mandatory body section. SOAP achieves extensibility via the optional header section. SOAP headers enables WS-* to achieve QoS e.g. security, reliability and transactionality. The message body surrounds the application specific content that represents the central purpose of the message.

2.1.2. Document style SOAP

When structuring a SOAP message, Document-style is now best practice. Document style SOAP uses XML Schema data types and the messages are actual XML instance documents.

2.2. Plain Old XML (POX)

This approach is similar to WS-* in that it is based on XML. It is a more lightweight approach as it relies on fewer protocols and messages have no SOAP content i.e. POX has no SOAP envelope, SOAP headers or SOAP body content. This means that POX messages are smaller. However, POX does not support QoS.

2.3. REST

REST is an architectural style for building distributed hypermedia systems (systems linked by hyperlinks). In REST, one accesses "resources". Resources are abstract concepts e.g. you may ask for a "web page" resource/concept and what is returned is a concrete manifestation of that concept called a representation e.g. a HTML page. The representation places the client into a *state*. If the client traverses a link from that web page to another web page, the client is *transfer*red into another state. Hence the term Representational State Transfer. There are several cornerstones to REST:

2.3.1. Uniquely Addressable Resources

A Uniform Resource Indicator (URI) allows one to describe the location of some resource anywhere in the world from anywhere in the world. One of the key features of REST is that every resource must have a unique URI. These URIs are logical and not physical.

2.3.2. Uniform Interface

The Uniform Interface refers to what is the same for all resources that you interact with. It is implemented via the following four sub-constraints (Fielding 2000):

- Resources must be able to identify themselves (URI).
- Manipulation of resources via a small set of methods e.g. HTTP GET, PUT, POST and DELETE.
- Self-descriptive messages (because a resource can have multiple representation types e.g. XML and HTML, the message must state which type it is so that clients can interpret it).
- Hypermedia as the engine of application state (resource representations contain hyperlinks to enable client transitions between application state).

In a RESTful architecture, everything is modeled in terms of resources which are accessed using URIs, and the four HTTP operations of GET, POST, PUT and DELETE.

2.3.3. Protocol Design

REST mandates that the protocol used must support the properties of layering, statelessness and caching.

In the context of the Web, HTTP has all these characteristics. While REST does not mandate the use of HTTP, it is the de facto protocol in REST.

3. Related Work

There have been efforts to improve the efficiency of XML data transfer in various scenarios. The main contribution of our effort is that rather than using XML compression technologies as in (Johnsrud Lars et al. 2008) we are applying RESTful techniques to reduce both the message size and also to reduce the number of messages sent. The environment for (Johnsrud Lars et al. 2008) is a mobile network where the research has a military focus. This meant that the SOAP based Web service request could not be compressed as the intermediaries, for security reasons, wanted to inspect the headers. They compressed the reply message only. In our framework, rather than tunnelling using POST we use HTTP properly i.e. GET where appropriate. Consequently, the target (origin) server will not receive every request, only the requests that are not served by intermediate caches. As well as that, the server will not send

back a full reply in all instances (only when the resource has changed). Thus the overall number of messages on the network is reduced.

As far back as 2003 people have realised the caching issue with XML Web Services: "*XML Web services present new challenges ..., as well as their lack of involvement in the caching process*" (Microsoft Research 2003). This research implemented a client-side SOAP cache to mimic continued access to Web services from mobile devices during disconnections. This cache was implemented as a data store. Their findings are interesting: "*The diverse nature of Web services poses a major problem in identifying the semantics of the operations exposed by the Web service*". Also, "*the applications ran just fine while disconnected as long as... the cache manager could identify similar requests*".

Our proposal to use REST principles (HTTP GET and unique URIs) solves these issues. In fact, Microsoft state that "*Web-browser caches, map URIs to HTML pages and need worry about only one operation – namely, the HTTP GET operation*". The Uniform Interface constraint of REST has major advantages over the specific interface approach of RPC-XML (SOAP/POX) in the areas of visibility into system interactions. (Vinoski, S (a) 2008). So rather than attempting to implement another SOAP caching mechanism, we are proposing to leverage the Web's proven caching architecture.

In (Briggs 2006), Briggs outlines a framework similar in appearance to ours (a RESTful back end with a SOAP front end). However, we differ in two important aspects. The first is that POX is not considered at all, only SOAP. The second is that the approach is the complete opposite to our approach. Briggs suggests wrapping RESTful services with SOAP to enable access to those who require it. Thus, new clients would be SOAP based and existing SOAP clients left untouched. Our approach is to transition all clients to RESTful format i.e. remove SOAP/POX. Exiting clients will use the mapping tool until convenient to migrate. New clients would be coded RESTfully.

4. Framework

Figure 1 outlines the framework architecture. The framework consists of a mapping from XML-RPC to REST. Both SOAP and POX surround an RPC-style method-call with XML metadata. As stated previously, with regard to SOAP, we will target the messages which have no QoS elements i.e. no SOAP headers. Both SOAP and POX tunnel using HTTP POST. This makes the message opaque to intermediaries.

4.1. Mapping Server

The client executes an application on the mapping server passing it is XML-RPC document. The application checks to see if the XML-RPC request is a read-only request i.e. does the XML-RPC service name exist in the mapping database table. If the service does not exist in the mapping table then the Web service invocation proceeds as normal i.e. an XML document is POSTed to a gateway URI. If it does exist in the mapping table i.e. the Web service is a read-only request, then the XML document intended for a gateway URI i.e. a common endpoint, is translated into a RESTful Conditional GET message with a unique URI. The new message will have no XML content, just a HTTP verb, URI and certain required HTTP headers. For example, the Conditional GET will require the "If-modified-since" header (Vinoski, S (b) 2008). There is a close relationship between the data (method and parameters) encoded in XML and the resource structure in the RESTful URI.

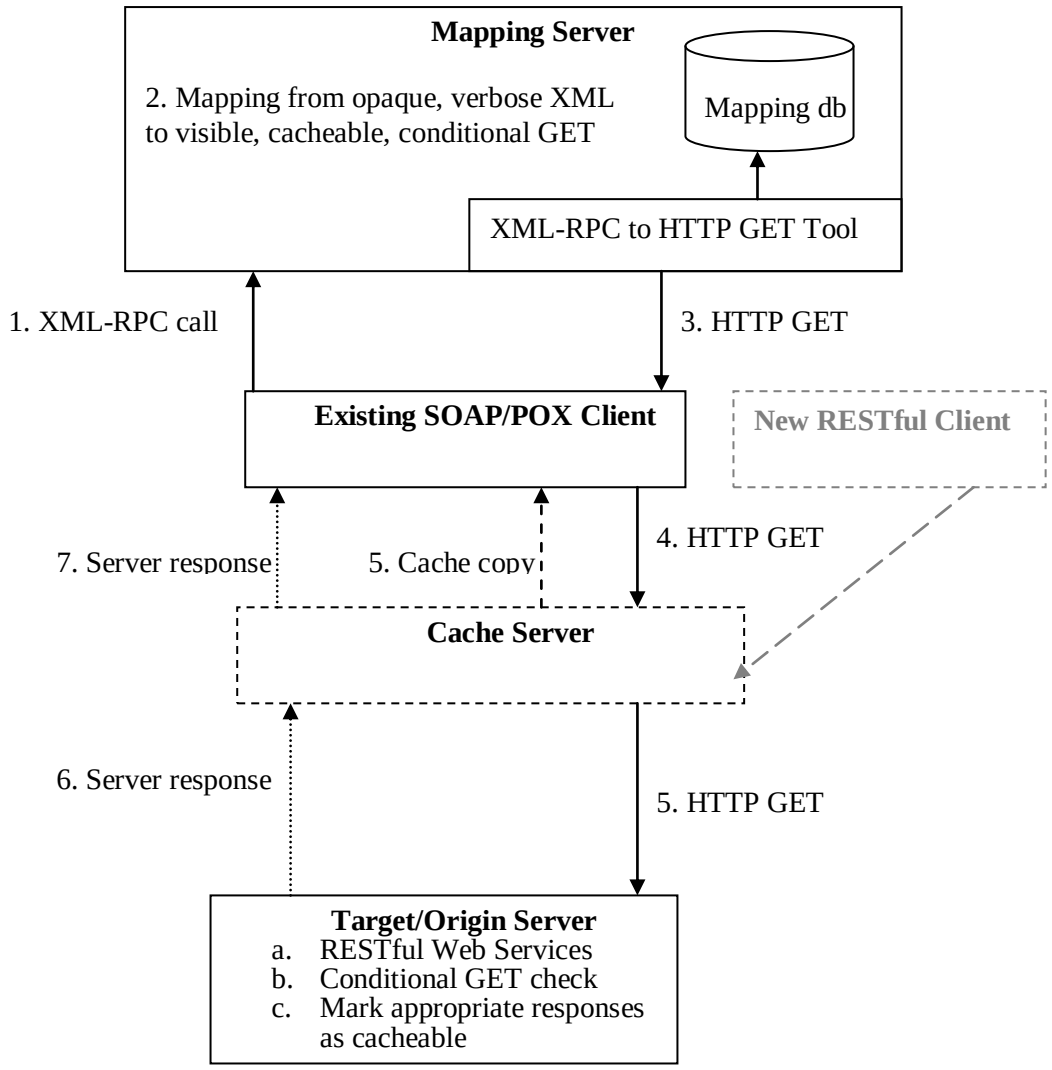

Figure 1 Framework Architecture

4.2. Cache server

A cache server will be inserted between the client and the target server. This will allow performance comparisons to be carried out between SOAP/POX and our framework where multiple equivalent requests are made. Note that it is the target/origin server that marks its responses as cacheable and for how long. However, intermediaries such as caches can inspect the new message to see if they have a fresh up-to-date copy and if so, return it without the need to forward on the request to the target server.

4.3. Conditional GET

As small scale systems may not have caching intermediaries, cacheability may not be of interest within the bounds of the enterprise. There are two approaches that we will outline here (Vinoski, S (b) 2008).

Date and Time: When a client issues a GET to a resource, the server inserts the date and time of the most recent change to the resource in the "*Last-modified*" header. The next time that client wants to GET that same resource, it sends along the value from the "*Last-modified*" header it received last time in the new requests "*If-modified-since*" header. The server uses this header to see if the resource has changed since the date and time specified by the client.

If the resource has not changed then the server returns a HTTP status code of 304 which means "not modified" *along with an empty message body* signifying that the client can continue to use the previous representation it received.

Entity Tags: The entity tag approach uses a resource hash to detect changes rather than date and time because the date and time mechanism leaves open a one second window where changes cannot be detected. The server returns the hash value as a string in the Etag header. Clients send this has value back to the server for subsequent requests in the "*If-none-match*" header. The server re-hashes the resource, compares it to the "*If-none-match*" header and if they are the same return status 304 along with an empty message body as before. One has to ensure that Etags computing costs are not prohibitive i.e. retrieving the whole resource representation (possibly involving several database queries) just to find out that it has not changed may not be very efficient. It might be just as quick to send back the whole representation in the first place.

As the date and time approach is more efficient, it is the approach we are going to take.

4.4. Target Server

The server side consists of RESTful implementations of the relevant Web services. To cater for new RESTful clients, other HTTP verbs must be supported, namely PUT, POST and DELETE. GET is required to support both new RESTful clients and existing SOAP/POX based clients (which will be transformed). A Java servlet will be written to listen on a particular root. The servlet will then parse the remainder of the URI and invoke the relevant RESTful Web service. As the Web service is now RESTful we are no longer restricted to XML as the response type but can return any Internet Assigned Number Authority (IANA) Multipurpose Internet Mail Extensions (MIME) types. New clients will be written RESTfully.

5. Testing

5.1. Test Environment

A test environment has been developed on an Apple MacBook Pro laptop with a 2.4GHz Intel Core 2 Duo processor and 4GB of memory. The IDE is NetBeans v6.5 which has integrated JAX-RS (Java API for Restful Web Services). Apache Tomcat v6.0.18 is the servlet container. As Tomcat does not come with a Web service engine, we downloaded and installed Apache Axis2 v1.4.1. A network analyser monitor was inserted between the client and server so as to monitor the messages going between them. The Web services were kept very simple as we were more interested in the message sizes and how long it took for the relevant processing logic to be called than in the processing logic itself. Each Web service was a pseudo-retrieval of a bank balance for customer id 12345678. Each Web service, for consistency, returned a single XML element (a balance of 123.45, without any database queries).

A client Java application was set up which called three methods: REST(), POX() and SOAP() respectively. These methods called a RESTful, POX-based and SOAP-based Web service respectively. In order to call the RESTful Web service we downloaded and installed Apache httpclient v4.0.

5.2. Message Sizes

Figures 2 details the RESTful, POX and SOAP based Web service request and reply message sizes. As regards the request message, we can see that the POX and (especially) SOAP messages are significantly larger than the RESTful message. Their character counts are 190, 392 and 515 respectively (counting spaces).

With regard to the response messages: the RESTful, POX and SOAP responses are 249, 167 and a bloated 406 characters respectively. The difference between REST and POX sizes is due to the Conditional GET and Caching headers (totalling 88 characters).

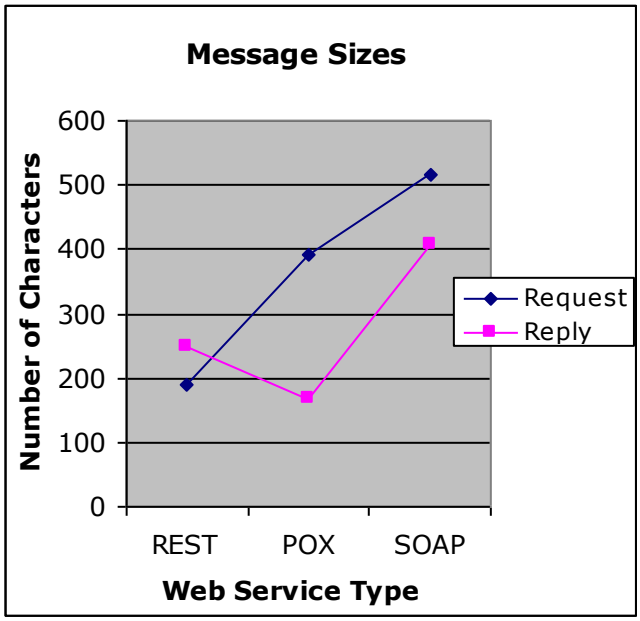

Figure 2 Message Sizes

5.3. Timings

Timings were taken before and after 10 calls to each Web service to determine how long each one took. These timings were typical of previous tests. Figure 3 details our findings. As can be seen, REST and POX are similar while SOAP is twice as slow as REST/POX.

6. Discussion

Web services are heavily based on XML. However XML is verbose and the more XML protocols you use the more bloated the message becomes. This is why SOAP messages are larger than POX messages. In fact, Bray stated in (Bray 2008) that "*REST does what [the SOAP stack] was trying to do in a much more viable, elegant, cheap, affordable way except that we've got no tooling around it yet*".

Some organisations are moving away from SOAP and using POX stating poor SOAP performance and no need for QoS as reasons for this (Allied Irish Bank 2007). Couple this with Heinemeir Hansson's (Heinemeir Hansson, D. 2007) quote "*a renaissance of HTTP appreciation is building under the banner of REST*" and we believe that providing enterprises with a mapping framework to enable them to ease the transition from existing SOAP/POX implementations to RESTful HTTP is valuable.

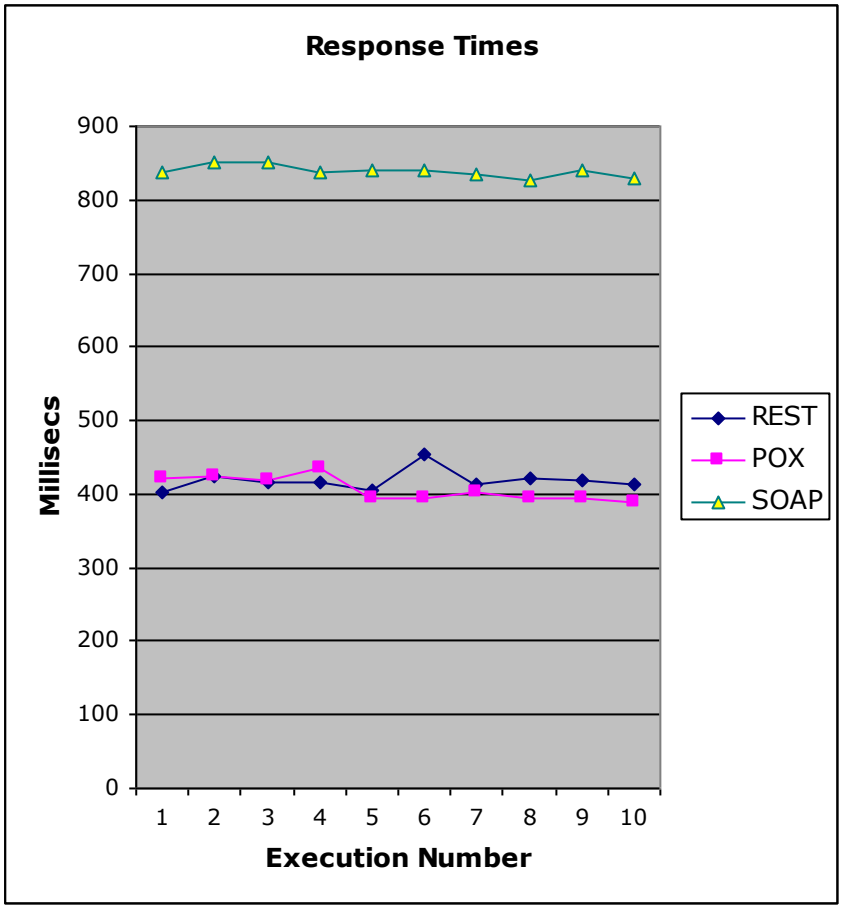

Figure 3 Response Times

The advantages of our framework are as follows:

1. The request message footprint is significantly reduced. This is an important consideration in environments where bandwidth is low and communication latency/costs are high e.g. mobile Web services (Johnsrud Lars et al. 2008). We believe that the increased message response size (when comparing REST to POX), which is due to the presence of the caching and conditional GET headers, is worth the overhead.
2. The new message is a transparent HTTP GET. This is important as SOAP/POX implementations currently have to find alternative ways to cache their messages (Microsoft Research 2003). As Vinoski states in (Vinoski, S (c) 2008) "the uniform-interface constraint (of REST) helps enable *visibility* into client-server interactions, making it easier for developers to apply critical distributed systems concepts such as proxying, caching, intermediation and monitoring".
3. Organisations considering RESTful Web Services may already have SOAP/POX implementations in place. While the server would migrate from SOAP/POX to REST immediately, a wholesale replacement of the clients can be avoided by adopting the framework. Servers can migrate from SOAP/POX to REST without breaking any clients i.e. no re-compilation of the clients is required as the interface is still the same from the clients perspective. These clients can migrate when convenient. New clients can talk directly to the RESTful Web Services immediately.

7. Future Work

The framework needs to be fully implemented with more realistic Web services as in (Johnsrud Lars et al. 2008) where a Phonebook Web service interacts with a backend datastore. Once implemented, the framework would support *Lo-REST* (Pautasso et al. 2008) (i.e. HTTP GET and POST only). We intend the architecture to support *Hi-REST* (Pautasso et al. 2008) (HTTP GET, PUT, POST and DELETE). This would involve further mappings such that all of the XML-RPC interface-specific Web services would then be transformed to RESTful HTTP (not just the read-only Web services).

References

Allied Irish Bank, private conversation IT architects, 2007.

Bray, T., SUN and co-inventor of XML, in an interview at O'Reilly Open Source Convention, 2008

Briggs, J. *Playing Together Nicely: Getting REST and SOAP to Share Each Other's Toys,* 2006, http://www.onjava.com/pub/a/onjava/2006/02/15/jython-soap-interface-to-rest.html.

Costello, Roger L. REST (Representational State Transfer), 2002 http://www.xfront.com/files/tutorials.html.

Fielding, R. *Architectural Styles and the Design of Network-Based Software Architectures*, doctoral dissertation, Dept. of Computer Scirnce, Univ. of Calif, Irvine, USA, 2000.

Graham S. et al, *Building Web Services with Java*, Sams Publishing. Indianapolis, USA 2nd edition. 2005

Heinemeir Hansson, D., Foreword in *RESTful Web Services by Richardson L and Ruby S*, O' Reilly, 2007

Johnsrud Lars et al. *Efficient Web Services in Mobile Networks*, European Conference on Web Services, Dublin, 2008.

Little, M., Red Hat Director of Standards and Technical Development Manager, interview at Qcon, London, 2008 http://www.infoq.com/interviews/mark-little-qcon08.

Microsoft Research. *Caching XML Web Services for Mobility*, Queue magazine, May 2003.

Pautasso, C. & Zimmermann O. & Leymann F. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision, *Proceedings of International WWW Conference*, Beijing, China, 2008.

Tilkov, S., innoQ consultant and member of JAX-RS (JSR-311) expert group, Software Engineering Radio interview, 2008 http://www.se-radio.net/podcast/2008-05/episode-98-stefan-tilkov-rest.

Vinoski, S (a). *Serendipitous Reuse*, IEEE Internet Computing magazine, Jan-Feb, pp 84-87, 2008.

Vinoski, S (b). *RESTful Web Services Development Checklist*, IEEE Internet Computing magazine, Nov-Dec, pp 94-96, 2008

Vinoski, S (c). *Demystifying RESTful Data Coupling*, IEEE Internet Computing magazine, Mar-Apr, pp 87-90, 2008.