A RESTful implementation of the WS-Agreement specification

Roland Kübert, Gregory Katsaros, Tinghe Wang
Höchstleistungsrechenzentrum Stuttgart
Nobelstraße 19
70569 Stuttgart
{kuebert,katsaros,twang}@hlrs.de

ABSTRACT

Representational State Transfer (REST) is an architectural style for distributed systems. RESTful web services have been gaining popularity in the last years. The Java API for RESTful Web Services (JAX-RS) has been specified as Java Specification Request 311 and is therefore an official part of Java; with the Jersey framework, a robust reference implementation of the specification exists. We examine in how far RESTful web services can fulfill tasks that have been defined as WS-* specifications. In particular, we investigate how a RESTful design and implementation of the WS-Agreement specification can be realized, presenting a light-weight approach to the creation and management of service level agreements.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: Standards; C.5.m [**Computer System Implementation**]: Miscellaneous; C.2.4 [**Computer-communication networks**]: Distributed Systems

General Terms

Design

1. INTRODUCTION

The terms service level and service level agreement (SLA) have been initially defined and widely used in the telecommunication sector. The need for providing to the customer a certain level of quality of service (QoS) along with adoption of more flexible business policies drove the research and development to invest a lot of effort to this very topic [1][2][3][4]. From the emergence of service oriented architecture (SOA) until cloud computing lately, SLAs have been an integral part of contemporary research in those fields as well [5][6][7]. SLAs can be seen as bipartite electronic contracts, which define service levels between a service provider and a customer.Service levels define acceptable QoS thresholds agreed upon by both parties. As a proposed recommendation by the Open Grid Forum (OGF), the WS-Agreement Specification defines a language to specify an SLA and defines a protocol to create SLAs based on so-called templates, which are blueprints for the final agreements. It is fundamental to automate the contracting process as well as the whole agreement life cycle, including creation, provisioning, monitoring and assessment of agreed terms. Currently there are implementations of the WS-Agreement specification with the traditional SOA approach, which have been mostly applied and tried in European research projects [8] [9] [10].

In contrast to the classical SOA approach realized with SOAP web services, RESTful web services are an architectural practice which utilizes web standards such as HTTP, URI, XML. In the last years, the RESTful style is gaining popularity in web development. In this paper we present the implementation of the WS-Agreement specification with a RESTful approach, which can be divided into four steps: identifying resources, defining URLs, deciding on resource representation and assigning HTTP methods.

The remainder of this work is organized as follows: in section two we are presenting the two major architectural styles for developing web services, in section three we are analyzing the Web Services Agreement Specification (WS-Ag), in section four we present detail of the implementation of the aforementioned specification using the REST architectural style and in the last section we conclude and summarize our findings.

2. WEB SERVICES AND REPRESENTATIONAL STATE TRANSFER

For years the best practice for realizing web services was through SOAP [11], a standard recommended by W3C [12], OASIS [13] and supported by enterprises such as IBM and Microsoft. SOAP interfaces are usually defined by a Web Service Description Language (WSDL) [14] document which describes methods and expected inputs and outputs, while Extensible Markup Language (XML) [15] is used to describe the schemas for those inputs and outputs. This very set of languages and standards, which can be often termed as WS-*, is being used extensively for the implementation of complex web service frameworks by researchers as well as by enterprises. In 2000 Roy Fielding in his doctoral dissertation [16] defined Representational State Transfer (a.k.a.REST), an "architectural style for distributed hypermedia systems". From that point on REST became a strong competitor to WS-*.

WS-REST 2011, Mar 28-28 2011, Hyderabad, India

The fundamental principle of the REST concept is that everything is a resource identified by a Uniform Resource Identifier (URI). It consumes HTTP as an application level protocol utilizing four of HTTP's request methods - GET, POST, PUT and DELETE - for the realization of the CRUD operations (create, read, update, delete). Interactions with resources are stateless in the sense that each request to the server must contain all the necessary information in order for the latter to understand or generate the state of the resource. The equivalent to WSDL for RESTful services is WADL, a machine-readable description of HTTP-based applications [17]; WADL, in contrast to WSDL, which is W3C a recommendation, was a member submission by Sun to the W3C in 2009 but "W3C has no plans to take up work based on this Submission" [18].

SOAP utilizes HTTP as a transport protocol by sending messages through the POST request method. SOAP web services are, similar to RESTful web services, normally stateless as well, but can be rendered stateful as proposed in the WS-Resource Framework (WSRF) set of specifications; thus, the server can maintain state across multiple transactions [19]. Finally, the security for the WS-* family of specification is defined by the WS-Security framework, while for the RESTful services, secure communication is realized through HTTPS (HTTP+SSL/TLS). As also presented in the analytical comparison in [20], both techniques are rather similar, although having their own strong and weak points. Overall, the lack in alternative options that REST provides, results in a simplified structure that offers better flexibility and control. When it comes to enterprise applications though, the reliability and better defined security features of WS-* make SOAP a probably more appropriate solution.

3. THE WEB SERVICES AGREEMENT SPECIFICATION

The Web Service Agreement Specification (WS-Agreement) is a proposed recommendation which has been worked out by the Grid Resource Allocation Agreement Protocol (GRAAP) working group of the Open Grid Forum (OGF). The OGF is a community "committed to driving the rapid evolution and adoption of applied distributed computing" [21]. It has published more than 150 documents, for example the well-known Open Grid Service Architecture (OGSA) specification, the Open Cloud Computing Interface (OCCI) specification, grid extensions to the FTP protocol (GridFTP) and many more [22].

WS-Agreement describes an XML schema for specifying service level agreements, for example between a service provider and a consumer, as well as a simple protocol that describes how parties can communicate in order to build an agreement. WS-Agreement builds on other web service specifications, namely the following:

- Web Services Addressing (WS-Addressing) 1.0 Core
- Web Services ResourceProperties (WSRF RP) 1.2
- Web Services Resource Lifetime 1.2 (WSRF RLF)
- Web Services Base Faults 1.2 (WSRF BF)

Elements of these specifications are included into WS-Agreement, for example the Endpoint Reference from WS-Addressing which describes how resources can be addressed or fault types from BaseFaults which define error messages that can be thrown by operations. Other parts of these specifications are used to extend the interface that WS-Agreement specifies by itself. This is done by including port types described by WSRF RLF and WSRF RP, which can be thought of as inheriting from these specifications as super classes.

WSRF RLF provides operation for specifying the life time of a resource while WSRF RP describes standardized operations to get, set or query so-called "resource properties"; they can be thought of as getters and setters in Java, which are instance methods that retrieve or store values from object instances.

3.1 WS-Agreement port types

Port types can be thought of as specifications of an interface of a service, more precisely "A port type is a named set of abstract operations and the abstract messages involved" [23]. In WSDL 2.0, port types have consequently been renamed *i*nterfaces [24].

WS-Agreement itself specifies the following port types:

- AgreementFactory
- PendingAgreementFactory
- Agreement
- AgreementAcceptance
- AgreementState

These port types offer the possibility to implement different scenarios, for example asymmetric, symmetric, direct or deferred, and combinations of these. We focus on the implementation of a case that is often used, the "simple client-server" scenario [25]:

> The AgreementFactory can be invoked by initiators to create a responder-side Agreement and to perform monitoring and management using only WS client mechanisms. Both port types would be implemented by the responder.

The only relevant port type in this scenario are the AgreementFactory port type, which offers an operation to create agreements, and the Agreement port type itself. It is, however, desirable, to use the AgreementState port type as well. The AgreementState port type's purpose is "to define a resource document type for monitoring the state of the agreement"; it should be "composed into a domain-specific Agreement port type", that means it is not foreseen to be used on its own.

3.2 Creating an agreement

The AgreementFactory is invoked through one operation, createAgreement. The input is an AgreementOffer and, possibly extension elements. We focus on the generic part and ignore the extensions; as we deal with the client-server scenario, we can safely ignore the input *i*nitiatorAgreementEPR, which is "a contact point (...) where the invoked party can send messages pertaining to this Agreement", as the client does not implement any server functionality.

The response of a call to *c*reateAgreement is an Endpoint Reference, that is the address of a resource that was created,

indicating the acceptance of an agreement [26]. If the offer was rejected, a fault is to be thrown [25, p. 39].

But how does the invoker know what to send to the responder? This problem is solved by the AgreementFactory, respectively the invoker, providing templates. Templates describe how an offer can be created, without giving a guarantee on its acceptance; they offer a mechanism to allow the responder to define constraints on possible fields which can be easily validated. Templates are, however, not exposed through "normal" operations by the service - like, for example, the "createAgreement" operation - but are resource properties. Resource properties can be accesses through generic operations specified in WSRF, for example *GetResourceProperty* to retrieve a single resource property, *GetMultipleResourceProperties* to retrieve a number of resource properties etc. [19, pp. 21–30].

3.3 Using an agreement

If an agreement has been reached, that means an agreement resource has been created; otherwise, a fault would have been thrown. An agreement exists in a certain state after its creation, the specification dictates that this state must be either "Observed" or "Complete".

Section 3.1 described the different port types offered by WS-Agreement, among them the AgreementState port type. This is to be used together with the Agreement port type in order for the agreement to have states.

An agreement, after creation, can be in three states: pending, observed, or rejected. Pending means the agreement has neither been accepted or rejected while observed means it has been accepted and rejected means it has been rejected. Agreements can either run until completion or be terminated. Completion means that an offer had been accepted initially but all activities regarding the agreement have been finished; termination means that and offer has been terminated and that no obligations exist any longer. Penalties may be imposed when an agreement is in the terminating state.

An agreement can be

- terminated or
- queried.

Queries against an agreement can be made using the *GetResourceProperty* operation defined in WSRF RP; this allows a caller to obtain the different parts of the agreement, that is the name, id, context and terms.

4. FROM WS-AGREEMENT TO RESTFUL AGREEMENT

It is not possible to directly "translate" the WS-Agreement specification to a RESTful service. This is due to the nature and differences of WS-* and RESTful services (see section 2). One can, however, implement a service that is similar to WS-Agreement using RESTful principles, thereby providing a RESTful way to create and manage agreements. This section describes the steps performed in order to implement a RESTful Agreement service [27, pp. 69–79]. We have already identified the requirements our service needs to fulfill, therefore we begin with identifying the resources our service needs. We then decide on which resource representations to offer, which URLs we use to access our resources, how we validate messages that are passed to us from the caller and, finally, how we represent state.

Figure 2: The four steps in developing a RESTful web service.

4.1 Identifying resources

From the requirements discussed in the scope of the client-server scenario that we want to support, we can identify agreements and templates. Both templates and agreements appear in two forms: single resources and lists of resources. This leads us to the following resources:

- Template
- List of templates
- Agreement
- List of agreements
- Agreement state

4.2 Deciding on resource representations

We want to provide a straightforward translation from the WS-Agreement specification to RESTful implementation, therefore we of course have to offer XML as a representation of our resources. For simplicity, we offer a textual representation of the resources as well where it makes sense.

The XML representation comes naturally from WS-Agreement, which specifies an XML schema for templates, agreements and states. WS-Agreement, however, does not specify a representation of a list of templates or agreements. A list of templates is defined like follows:

```
<templates>
<wsag:Template
  xmlns:wsag="..."
  ...>
  <wsag:Name></wsag:Name>
</wsag:Template>
<wsag:Template
  xmlns:wsag="..."
  ...>
</wsag:Template>
</templates>
```

Likewise, a list of agreements is defined as follows:

```
<agreements>
<wsag:AgreementType
  xmlns:wsag="..."
  ...>
  <wsag:Name></wsag:Name>
</wsag:AgreementType>
<wsag:AgreementType
  xmlns:wsag="..."
  ...>
</wsag:AgreementType>
</agreements>
```

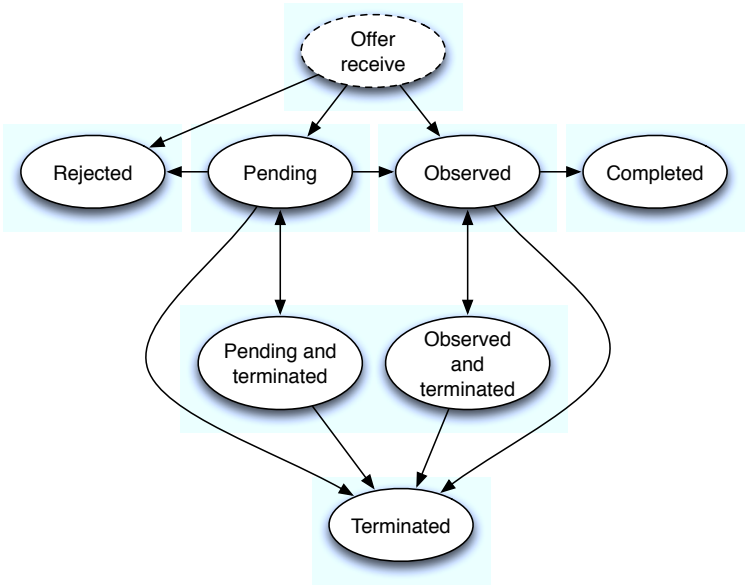

Figure 1: State diagram of agreement states.

The advantage of using WS-Agreement's XML schema is that the RESTful service can easily communicate with "legacy" services using WS-Agreement. Representing templates, agreements etc. as well in other forms, for example JSON [29], can provide for a better usability, for example using JavaScript.

4.3 Defining URLs

The URIs of a RESTful web service define its API, which in our case is a public API. APIs should be logical, hierarchical and as permanent as possible [27]. We therefore try to develop an API that is not likely to change much [28].

We assume that our web service will be deployed at `http://localhost:8080/rest-agreement/`. The host name will change for each service provider, but the path from the host name on should be consistent with the one we propose[1].

As proposed in [27], we define keywords for identifiers that do not change as parts of the URI and we enclose dynamic keywords in curly braces. The URIs for out service therefore are:

/rest-agreement/templates With the GET method, this URI returns a list of all templates. The POST method allows the creation of a new template. The PUT and DELETE methods are not supported for a list of templates.

/rest-agreement/templates/search/{query} With the GET method, this URI returns a list of templates that match the given query. The POST, PUT and DELETE methods are not supported.

/rest-agreement/template/{id} With the GET method, this URI returns a representation of a template with a unique identifier specified in the URI; with the PUT method, it updates a template; and, with the DELETE method, it deletes a template. [2]

rest-agreement/agreements The POST method allows the creation of a new agreement. This method on this URI is thus equivalent to WS-Agreements "createAgreement" call; an agreement is only created if the service provider, that is the service, agrees. The GET method returns a list of all agreements, the PUT and DELETE methods are not supported.

rest-agreement/agreement/{id} The GET method invoked on this URI returns a specific agreement. As agreements are binding once agreed upon, they cannot be updated, hence there is no support for the PUT method. Likewise, there is no support for the POST method. Agreements can be deleted with the DELETE method, although the service provider has to keep into account that he may be obliged to store agreements even after they have expired.

rest-agreement/agreement/{id}/state The GET method invoked on this URI returns the state of the agreement with the current id. The POST method allows the creation of a resource holding state for an agreement with the given id, the PUT method allows the update of a resource and the DELETE method removes a resource. Except for the GET method, only the service provider should be allowed to invoke the methods.

4.4 Validating templates, offers and agreements

Using an XML Schema Document (XSD), the WS-Agreement specification defines the syntax and data format to describe how to represent SLAs, templates etc. Thus, every SLA instance (SLA Template, SLA offer and SLA) must be compliant with the WS-Agreement XSD in its life cycle. To

[1]Note that it is possible to deploy the service to the root of the host as well, but this is normally not intended as many web services can be running on the same host.

[2]Clients should of course not be allowed to call the PUT and DELETE methods. We refrain from further description of the implementation of security.

ensure the correctness of an SLA instance, the validation includes two parts:

1. Every SLA instance is in XML format.
2. Every SLA template or SLA is an instance of the WS-Agreement XSD.

In a RESTful implementation, the first validation can benefit directly from the HTTP protocol by setting the HTTP media type with XML (application/xml, text/xml, application/*+xml). Based on the HTTP response code, one can know if the document is valid XML. If not, it surely is not valid according to the WS-Agreement schema and does not even need to be processed further but an error code "406 not acceptable" can be returned to the sender.

The second validation can be trivially realized by Java types generated directly from the WS-Agreement schema. XMLBeans, for example, compiles a schema into Java objects which can be accessed in a JavaBeans style way, using getters and setters [30]. XMLBeans provides different possibilities for creating instances, for example through deserialization of a text string. This allows for a straightforward validation without using custom code. Domain-specific parts of course need to be handled in a special way, but this can be realized, for example, through an XML schema for the domain-specific parts, leveraging the advantages of XMLBeans again.

4.5 Handling state

A point that is often stressed when talking about REST is that "application state is to be kept on the client". This seems unusual if one comes from a WS-* world - after all, why shouldn't resources have a state themselves? Actually, the formulation of keeping application state on the client is kind of misleading. RESTful service can of course manipulate the state of resources through creation, retrieval, update and deletion; every result of an operation, however, must be a resource representation [27, p. 12].

There are at least two options for handling state in a RESTful service [31, p. 7]:

- Encode state in URIs or
- store state in a persistent storage (for example a database) and encode a reference to the state in the URIs.

The question how to represent state does not pose itself for templates: templates either exist or they do not exist; the service provider has to store templates, of course, as the client needs to query them from the service provider. We decided to simple use a file system-based storage as it can be implemented easily.

Agreements come into existence if mutual agreement between provider and client is reached. Should we encode the agreement state then in the URI and let the client specify it during calls? In our case, this does not make sense: firstly, the service provider needs to store the resource anyway in order to access it, for legal reasons, etc. If the agreement would only exist on the client side, a client could easily manipulate it and force the provider to fulfill terms he did not agree too. Agreements therefore need to be saved on the provider side in a persistent way; we decided to use a database-backed system for this. There will most probably be a high fluctuation in the creation and deletion of agreements so it makes sense to not use a file system-based solution for this.

The actual "agreement state" proposed by WS-Agreement is a property of a port type that can be implemented by an agreement; as it is not specified in the agreement document, we could either store it as a domain-specific extension or use a separate resource to represent it. As has been described in section 4.3, we have chosen to use a separate resource for the agreement state. This makes for a cleaner implementation as we don't have to specify, inject and extract domain-specific parts into the agreement documents. It is as well quite close to the original WS-Agreement specification, with the difference that the agreement state is not part of an agreement resource but a resource in its own right.

5. CONCLUSIONS

This work has presented how a web service specification, WS-Agreement, can be implemented in a RESTful way. For this implementation, we have focused on one specific scenario that is presented by WS-Agreement, namely a simple client-server scenario. In this scenario, the client does not provide any interface to the service provider, but in the literal sense of the word acts purely as a client accessing the interface offered by the service provider. This is only one out of a number of scenarios that can be addressed by WS-Agreement; it is, however, one that has been addressed and implemented quite often using WS-* tools, which is, without doubt, due to its inherent simplicity. Apart from that, this scenario uses all relevant parts of the WS-Agreement specification: it specifies an endpoint where a client can create agreements, it uses template and agreement documents and has the ability to track the state of an agreement.

Even though there are conceptually quite many differences between the approaches taken by the WS-* web service and REST, we have implemented a solution that supports the basic client-server scenario. As REST is solely an architectural approach and not a specification, we have only used the WS-Agreement specification itself for defining the resources and thereby the messages we use. In this sense, a RESTful implementation is much more straightforward as the REST world does not yet suffer from the many specifications that exist in WS-*[3]. REST itself of course makes use of other specifications as well (HTTP, XML, HTML, XML Schema, URL, URI, etc.).

The RESTful implementation presented in this work has three great advantages in comparison to the WS-* implementation: bookmarkability, multiple representations and well-defined links [32]. Bookmarkability refers to the fact that every URI points to a unique entity and that every entity is referenced by a URI. Multiple representations means the fact that resources can be provided in different views: an application might use XML to communicate with the server while HTML can be used provide human users with the possibility to quickly investigate resources. The well-defined links show the connection between operations and resources.

In conclusion, with little programming complexity - one does not need to have deep experience in REST or use lots of code to develop a service like the one we presented - one can quickly implement a powerful service that mimics the behavior of the WS-Agreement specification but provides, for the first time, the opportunity to make use of service

[3]OASIS alone provides more than a dozen specifications, see `http://docs.oasis-open.org/`

level agreements in a RESTful way.

6. ACKNOWLEDGMENTS

This work has been supported by the OPTIMIS project (http://www.optimis-project.eu/) and has been partly funded by the European Commission's IST activity of the 7th Framework Program under contract number 257115. This work expresses the opinions of the authors and not necessarily those of the European Commission. The European Commission is not liable for any use that may be made of the information contained in this work.

7. REFERENCES

[1] J. Kosinski, P. Nawrocki, D. Radziszowski, K. Zielinski, S. Zielinski, G. Przybylski, and P. Wnek. Sla monitoring and management framework for telecommunication services. In *Networking and Services, 2008. ICNS 2008. Fourth International Conference on*, pages 170 –175, 2008.

[2] C. Courcoubetis and V. Siris. Managing and pricing service level agreements for differentiated services. In *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, 1999.

[3] P. Bhoj, S. Singhal, and S. Chutani. Sla management in federated environments. In *Integrated Network Management, 1999. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on*, 1999.

[4] L. Lewis and P. Ray. Service level management definition, architecture, and research challenges. In *Global Telecommunications Conference, 1999. GLOBECOM '99*, 1999.

[5] M. Alhamad, T. Dillon, and E. Chang. Conceptual sla framework for cloud computing. In *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, pages 606 –610, 2010.

[6] M. Alhamad, T. Dillon, and E. Chang. Sla-based trust model for cloud computing. In *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, pages 321 –324, 2010.

[7] S. de Chaves, C. Westphall, and F. Lamin. Sla perspective in security management for cloud computing. In *Networking and Services (ICNS), 2010 Sixth International Conference on*, pages 212 –217, 2010.

[8] BREIN Consortium. BREIN project homepage. `http://www.eu-brein.com/`.

[9] BEinGRID Consortium. BEinGRID project homepage. `http://www.beingrid.eu/`.

[10] IRMOS Consortium. IRMOS project homepage. `http://www.irmosproject.eu/`.

[11] SOAP. Project Website: `http://www.w3.org/TR/soap/`.

[12] W3C. Project Website: `http://www.w3.org`.

[13] OASIS. Project Website: `http://www.oasis-open.org`.

[14] WSDL. Project Website: `http://www.w3.org/TR/wsdl/`.

[15] XML. Project Website: `http://www.w3.org/TR/XML/`.

[16] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. AAI9980887.

[17] M. J. Hadley Web Application Description Language (WADL) Sun Microsystems, Inc. Mountain View, CA, USA, 2006

[18] Y. Lafon Team Comment on the "Web Application Description Language" Submission `http://www.w3.org/Submission/2009/03/Comment`

[19] M. Banks Web Services Resource Framework (WSRF) Primer v1.2. OASIS Web Services Resource Framework (WSRF) Technical Commitee

[20] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 805–814, New York, NY, USA, 2008. ACM.

[21] Open Grid Forum. Open Grid Forum home page. `http://www.ogf.org/`.

[22] Open Grid Forum. OGF documents. `http://www.ogf.org/gf/docs/?final\&all`.

[23] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Service Definition Language (WSDL). Technical report, World Wide Web Consortium (W3C), March 2001.

[24] J. J. Moreau, R. Chinnici, A. Ryman, and S. Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. Candidate recommendation, W3C, March 2006.

[25] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). Technical report, Global Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) WG, September 2005.

[26] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing - Core. Technical report, W3C, May 2006.

[27] J. Sandoval. *RESTful Java Web Services*. Packt Publishing, Birmingham, UK, 2009.

[28] T. B. Lee. Cool URIs don't change. 1998.

[29] D. Crockford RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON) IETF RFC, `http://tools.ietf.org/html/rfc4627`, 2006.

[30] The Apache Software Foundation. Apache xmlbeans project website. `http://xmlbeans.apache.org/index.html`.

[31] S. Allamaraju and M. Amudsen. *RESTful Web Services Cookbook*. O'Reilly, Sebastopol, 2010.

[32] S. Weerawarana. WS-* vs. REST: Mashing up the truth from facts, myths and lies. Presented at QCon San Francisco, 2007.