RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision

Cesare Pautasso
Faculty of Informatics
University of Lugano
via Buffi 13
6900 Lugano, Switzerland
cesare.pautasso@unisi.ch

Olaf Zimmermann
IBM Zurich Research Lab
Saeumerstrasse 4
8803 Rueschlikon, Switzerland
olz@zurich.ibm.com

Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
70569 Stuttgart, Germany
frank.leymann@iaas.uni-stuttgart.de

ABSTRACT

Recent technology trends in the Web Services (WS) domain indicate that a solution eliminating the presumed complexity of the WS-* standards may be in sight: advocates of REpresentational State Transfer (REST) have come to believe that their ideas explaining why the World Wide Web works are just as applicable to solve enterprise application integration problems and to simplify the plumbing required to build service-oriented architectures. In this paper we objectify the WS-* vs. REST debate by giving a quantitative technical comparison based on architectural principles and decisions. We show that the two approaches differ in the number of architectural decisions that must be made and in the number of available alternatives. This discrepancy between freedom-from-choice and freedom-of-choice explains the complexity difference perceived. However, we also show that there are significant differences in the consequences of certain decisions in terms of resulting development and maintenance costs. Our comparison helps technical decision makers to assess the two integration styles and technologies more objectively and select the one that best fits their needs: REST is well suited for basic, ad hoc integration scenarios, WS-* is more flexible and addresses advanced quality of service requirements commonly occurring in enterprise computing.

Categories and Subject Descriptors

A.1 [**General Literature**]: Introductory and Survey; C.2.2 [**Computer Communication Networks**]: Network Protocols; D.2.11 [**Software Engineering**]: Software Architectures; D.2.12 [**Software Engineering**]: Interoperability; H.1 [**Information Systems**]: Models and Principles

General Terms

Design, Standardization

Keywords

Architectural Decision Modeling, HTTP, REST, Resource Oriented Architecture, Service Oriented Architecture, SOAP, Technology Comparison, Web Services, WSDL, WS-* vs. REST

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

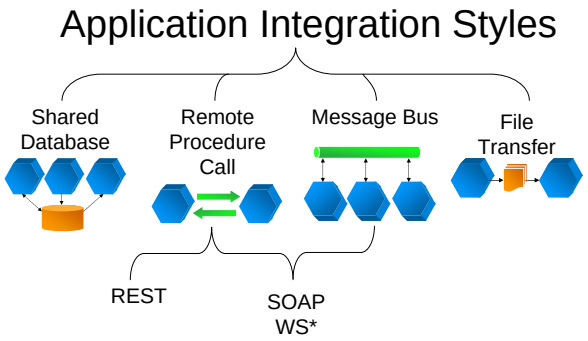

Figure 1: Putting the WS-* vs. REST decision in the context of application integration styles

1. INTRODUCTION

Many different styles can be used to integrate enterprise applications (Figure 1). The choice between relying on a *shared database*, using batched *file transfer*, calling *remote procedures*, or exchanging *asynchronous messages* over a message bus is a major architectural decision, which influences the requirements for the underlying middleware platform and the properties of the integrated system [20]. The "Big"[1] Web services technology stack [1, 45, 47] (SOAP, WSDL, WS-Addressing, WS-ReliableMessaging, WS-Security, etc.) delivers interoperability for both the Remote Procedure Call (RPC) and messaging integration styles [41]. More recently, an alternative solution has been brought forward to implement remote procedure calls across the Web: so-called RESTful Web services [12] are gaining increased attention not only because of their usage in the Application Programming Interface (API) of many Web 2.0 services [32], but also because of the *presumed* simplicity of publishing and consuming a RESTful Web service [40].

Key architectural decisions in distributed system design, such as the choice of integration style and technology, should be based on technical arguments and a fair comparison of concrete capabilities delivered by each alternative. Instead, the WS-* vs. REST debate has unfortunately degenerated into biased and religious arguments which create only confusion and expectations that cannot be fulfilled [19].

[1]We follow the naming convention introduced in [33].

In this paper, we take a quantitative approach to compare the two integration styles technologies based on architectural principles and decisions. The comparison is based on our industry project and teaching experience with both approaches. This way, the relative complexity (or simplicity) of WS-* and RESTful Web services can be measured in terms of 1) the number of decisions that have to be made, 2) the number of alternatives (options) that are available, and 3) the relative cost as indicated by development effort required and technical risk involved. As we will show, decisions may affect one another so that, e.g., deciding to use a RESTful approach at a high level will constrain the options for the choice of asynchronous messaging middleware at a lower level. Also, in some cases, the lack of options concerning some decisions (e.g., transactional guarantees, or the reliability of the message exchange) in REST may only apparently indicate a reduced complexity because the only choice left is to implement a custom solution, which incurs higher development and maintenance effort and risk.

This paper is organized as follows. First in Sections 2 and 3 we give background information on WS-* and RESTful Web services. In Section 4, we introduce our decision-centric comparison methodology. We then apply it by listing and comparing several important principles (Section 5), conceptual (Section 6) and technology (Section 7) decisions involved in adopting WS-* and REST. In Section 8, we present related work and in Section 9 we draw our conclusions.

2. SOAP AND THE WS-* STACK

Providing seamless interoperability between heterogeneous middleware technology stacks and fostering the loose coupling of *service consumer* (requestor, client) and *service provider* are the major design goals of Service-Oriented Architecture (SOA) *concepts* and Web services *technologies*.[2]

2.1 Concepts and Technology

On the conceptual level, a service is a software component provided through a network-accessible endpoint [16]. Service consumer and provider use messages to exchange invocation request and response information in the form of self-containing documents that make very few assumptions about the technological capabilities of the receiver. In particular, there is no notion of a remote object reference that would require an object broker to manage a distributed memory address space [43].

On the technology level, *SOAP* is an XML language defining a message architecture and message formats, hence providing a rudimentary processing protocol. The SOAP document defines a top-level XML element called `envelope`, which contains a `header` and a `body`. The SOAP header is an extensible container for message-layer infrastructure information that can be used for routing purposes (e.g., addressing) and Quality of Service (QoS) configuration (e.g., transactions, security, reliability). The body contains the payload of the message. XML Schema is used to describe the structure of the SOAP message, so that *SOAP engines* at the two endpoints can marshall and unmarshall the message content and route it to the appropriate implementation.

[2]We distinguish between concepts and technologies as follows: there are many ways to implement SOA. For example, Message-Oriented Middleware (MOM) and, if programming language independence is not a requirement, message-driven Enterprise JavaBeans can be viewed as SOA implementation technologies. In this paper, however, we concentrate on XML-based Web services as defined by W3C and OASIS [27] and restrict the discussion to the scope of the WS-I Basic Profile 1.1 [47]. For more information we refer the reader to [1, 45, 50].

Web Services Description Language (WSDL) is an XML language for defining interfaces syntactically. A WSDL `port type` contains multiple abstract `operations`, which are associated with some incoming and outgoing `messages`. The WSDL `binding` links the set of abstract operations with concrete transport protocols and serialization formats. At the time of writing, the only standardized binding uses SOAP over HTTP [21]. Vendors have defined several additional ones to support messaging protocols as well as proprietary legacy system interfaces. Service endpoints are addressed either on the transport level, i.e., with Universal Resource Identifiers (URIs) for SOAP/HTTP-bound endpoints, or on the messaging level via WS-Addressing [44].

By default, there is no notion of state. The interaction with stateful Web services is covered by the WS-Resource Framework [28], which handles the management of stateful resources behind a Web service interface. The WS-* technology stack covers also many other QoS features required to ensure the interoperability of advanced middleware systems [45]. Given the modularity and composability of the approach, this has led to a fairly large set of WS-* specifications.

2.2 Strengths

In spite of their perceived complexity, the SOAP message format and the WSDL interface definition language have gained widespread adoption as the gateway technologies capable of delivering interoperability between heterogeneous middleware systems.

One advantage of WS-* is *protocol transparency and independence*. Using SOAP, the same message in the same format can be transported across a variety of middleware systems, which may rely on HTTP, but also on many other transports. The transport protocol may change along the way. Being declared as SOAP headers, QoS aspects such as encryption and reliable transfer can be made independent from the transports used along the path ("end-to-end QoS").

Using WSDL to describe a *service interface* helps to abstract from the underlying communication protocol and serialization details as well as from the service implementation platform (operating system and programming language). WSDL contracts provide a machine-processable description of the syntax and structure of the corresponding request and response messages and define a flexible evolution path for the service. As business and technology requirements change, the same abstract service interface can be bound to different transport protocols and messaging endpoints. In particular, WSDL can model service interfaces for systems based on synchronous *and* asynchronous interaction patterns. This kind of flexibility becomes fundamental when building gateways for pre-existing legacy systems, which may not always use Web-friendly protocols.

Furthermore, mature SOAP engines and WSDL tools [2, 11] effectively hide the perceived complexity from the application programmer and integrator. According to our personal project experience [54], it is not required to study the specifications to be able to develop interoperable services, assuming that the selected runtimes and tools adhere to the WS-I Basic Profile [47]. Test clients can be generated from the WSDL contracts automatically.

2.3 Weaknesses

Paradoxically, the power delivered by current WS-* tooling that make it so easy to turn existing software components into Web services can also be misused [35]. Thus, it is important to avoid *leakage* across abstraction levels. Interoperability problems can occur when, for instance, native data types and language constructs of the service implementation are present in its interface. This weakness

can be mitigated by stating and enforcing certain design and coding guidelines such as *contract-first* development.

Given the expressivity of the WS-* stack of standards, early implementations have been plagued by interoperability problems. Apart from misinterpretations later clarified by WS-I, these can be partly blamed on the impedance mismatch between XML and existing (object-oriented) programming languages. For instance, the translation between the XML on the wire and the corresponding memory data structures has been problematic and is the main source of performance inefficiencies [15]. Taking Java as an example, several attempts were necessary (Apache SOAP to JAX-RPC 1.0 and 1.1 to JAX-WS and JAX-B) to produce a stable Web service marshalling layer [25]. Furthermore, XML Schema is a very rich language, making it difficult to identify the right construct to express a data model in a way that is fully supported by all SOAP/WSDL implementations. This problem can be avoided with profiling, identifying a subset of XML Schema types such as sequences which is "good enough" for most integration scenarios and known to be interoperable.

3. REST

REpresentational State Transfer (REST) was originally introduced as an architectural style for building large-scale distributed hypermedia systems. This architectural style is a rather abstract entity, whose principles have been used to explain the excellent scalability of the HTTP 1.0 protocol and have also constrained the design of its following version, HTTP 1.1. Thus, the term REST very often is used in conjunction with HTTP.

In this section we outline the main characteristics of REST focusing on the current interpretation used to define "RESTful" Web services. For more information we refer the reader to [12, 14, 33].

3.1 Technology Principles

The REST architectural style is based on four principles:

Resource identification through URI. A RESTful Web service exposes a set of resources which identify the targets of the interaction with its clients. Resources are identified by URIs [5], which provide a global addressing space for resource and service discovery.

Uniform interface. Resources are manipulated using a fixed set of four create, read, update, delete operations: `PUT`, `GET`, `POST`, and `DELETE`. `PUT` creates a new resource, which can be then deleted using `DELETE`. `GET` retrieves the current state of a resource in some representation. `POST` transfers a new state onto a resource.

Self-descriptive messages. Resources are decoupled from their representation so that their content can be accessed in a variety of formats (e.g., HTML, XML, plain text, PDF, JPEG, etc.). Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control.

Stateful interactions through hyperlinks. Every interaction with a resource is stateless, i.e., request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, e.g., URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction.

3.2 Strengths

RESTful Web services are perceived to be simple because REST leverages existing well-known W3C/IETF standards (HTTP, XML, URI, MIME) and the necessary infrastructure has already become pervasive. HTTP clients and servers are available for all major programming languages and operating system/hardware platforms, and the default HTTP port 80 is commonly left open by default in most firewall configurations.

Such lightweight infrastructure, where services can be built with minimal tooling, is inexpensive to acquire and thus has a very low barrier for adoption. The effort required to build a client to a RESTful service is very small as developers can begin testing such services from an ordinary Web browser, without having to develop custom client-side software. Deploying a RESTful Web service is very similar to building a dynamic Web site.

Furthermore, thanks to URIs and hyperlinks, REST has shown that it is possible to discover Web resources without an approach based on compulsory registration to a (centralized) repository.

On the operational side, it is known how to scale a stateless RESTful Web service to serve a very large number of clients, thanks to the support for caching, clustering and load balancing built into REST. Due to the possibility of choosing lightweight message formats, e.g., the JavaScript Object Notation (JSON [10]) or, in the extreme, even plain text for very simple data types, REST also gives more leeway to optimize the performance of a Web service.

3.3 Weaknesses

There is some confusion regarding the commonly accepted best practices for building RESTful Web services. *Hi-REST* recommendations have been established informally, but are not always fully followed by so-called *Lo-REST* implementations: Hi-REST, advocates using all of the 4 verbs (`GET`, `POST`, `PUT`, `DELETE`)[3]; recommends the use of (so-called) "nice" URIs; and suggests the use of Plain Old XML (POX) for formatting the content of messages[4]. Lo-REST, on the other hand, focuses on the minimum common denominator. Thus, only 2 verbs (`GET` for idempotent requests, and `POST` for everything else) are used. This is due to the fact that proxies and firewalls may not always allow HTTP connections that use any other verb. Also, `POST` and `GET` are the only two verbs that can be used in the `method` attribute of an XHTML form[5]. These restrictions have led to a series of workarounds, where the "real" verb is sent using either a special HTTP header (`X-HTTP-Method-Override`) or – like with Ruby on Rails – a hidden form field (`_method`). As with most non-standard workarounds, these may not be understood by all Web servers, and require additional development and testing effort. Regarding the message payload format, Lo-REST simply enforces the use of MIME-Types, but does not restrict the data to be in a particular format.

Another limitation makes it impossible to strictly follow the `GET` vs. `POST` rule. For *idempotent* requests having large amounts of *input* data (more than 4 KB in most current implementations) it is not possible to encode such data in the resource URI, as the server will reject such "malformed" URIs[6] – or in the worst case it will crash, exposing the service to buffer overflow attacks. The size of the request notwithstanding, it may also be challenging to encode complex data structures into a URI as there is no commonly accepted marshalling mechanism. Inherently, the `POST` method does not suffer from such limitations.

[3]The latest version of HTTP 1.1. actually includes 8 verbs: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `TRACE`, `OPTIONS`, `CONNECT`.

[4]At the time writing, it was debated which flavor of XML or which alternative serialization formats (e.g., JSON, YAML, etc.) guarantees the best interoperability.

[5]The XForms standard also allows `PUT`, but not yet `DELETE`.

[6]HTTP Code 414 - Request-URI too long.

4. COMPARISON METHOD

Architectural decisions [39] are the key metaphor in our comparison method. Architectural decisions capture the main design issues and the rationale behind a chosen technical solution; they can informally be defined as conscious design decisions concerning a software system as a whole or affecting one or more of its core components and determining the non-functional characteristics and quality factors of the system [52]. For each architectural decision, one or more *architecture alternatives* (AAs) enumerate the design options.

In our earlier work, we proposed a structured, proactive approach to architectural decision *modeling* [52], which extends today's practices of rather informal, retrospective decision *capturing* for documentation purposes. When reviewing the SOAP vs. REST debate, it becomes clear that many of the issues discussed and arguments brought forward qualify as recurring architectural decisions that can be modeled.

4.1 Preparation of Decision Models

Architectural decision models have several use cases, for example decision support and governance during architecture design; they can also be used to facilitate technical quality assurance reviews [53]. In the context of this work, we used them as a metric for the REST vs. WS-* technology comparison. We took the following steps:

1. Screening of reference information and positioning papers as well as online resources such as blog entries of proponents of the two integration styles, identifying candidate decisions and alternatives along the way.

2. Development of several sample integration scenarios, recording the architectural decisions and development steps required when using REST and WS-*.

3. Creation of one decision model for each of the two integration styles from the results of Steps 1 and 2.

4. Comparison of the models created in Step 3, leading to a review cycle to solicit additional input to make sure that both models actually address the same design issues and are complete enough for the following assessment step.

5. Walk through the now completed decision models to determine relative complexity and cost (as indicated by required development effort and technical risk) assessments.

6. Measure and compare 1) number of decisions, 2) number of options per decision and 3) cost assessment per option.

4.2 Comparison Levels

The two decision models we prepared are organized into four levels of abstraction:

1. *Comparison of architectural principles*. The intent of an architectural style such as REST is communicated through its defining principles [3]. These principles determine how the architectural style addresses its requirements and design goals. In our comparison models, principles and requirements such as protocol layering, dealing with heterogeneity, and loose coupling are investigated.

   A comparison of these principles comprises the first step in our comparison (Table 1). This is the level of abstraction and detail on which most existing comparisons of the two styles reside. Principles such as dealing with heterogeneity and loose coupling are typically defined informally; hence, an assessment solely based on these criteria only is bound to be subjective and incomplete. Therefore, we added three more steps.

2. *Comparison of conceptual decisions*. Next, we investigate conceptual decisions required when following one of the two respective styles (Table 2): we compare how service contract design is supported and discuss the similarities and differences between the methodologies for publishing a service.

3. *Comparison of technology decisions*. We then refine the conceptual decisions and compare how the two integration styles can be technically realized (Table 3).

4. *Vendor asset-level comparison*. On this level, concrete implementations tools can be evaluated in the light of the previously discussed decisions. For instance, we surveyed how contemporary Web browsers and Web servers implement the HTTP specification, and which SOAP engines and WSDL tools exist. Due to space limitations, we could not include these parts of the decision models in this paper.

5. COMPARISON OF PRINCIPLES

We start by comparing how REST and WS-* comply with the architectural principles introduced in the previous section.

5.1 Protocol Layering

The first architectural principle clarifies whether the Web is used as a publishing medium for delivering application services to clients or as a messaging medium for application integration – in other words, whether HTTP is considered as an *application* or as a *transport* protocol.

In the context of REST, the Web is seen as the universal medium for publishing globally accessible information. Applications become part of the Web by using URIs to identify the provided resources, data, and services and by leveraging the full semantics of the four HTTP verbs (`GET`, `POST`, `PUT`, and `DELETE`) to expose operations on such resources.

Instead, from the perspective of SOAP/WS-*, the Web is seen as the universal transport medium for messages, which are exchanged between Web services endpoints of published applications. Thus, applications gain the ability to remotely interact through the Web but remain "outside" of the Web. In other words, the HTTP protocol is used as a *tunneling* protocol to enable remote communication through firewalls, but it is not used to convey the semantics of the service interaction. This can be seen from the way WS-* uses URIs to address *messaging endpoints*, which typically remain the same for all operations of a service, whereas REST URIs identify *resources* of the application domain.

In WS-*, both request and response messages are exchanged using only one HTTP verb (`POST`), the only one which allows to transfer XML payloads in both directions. This way, the selection of the operation to be performed by the service is no longer done at the HTTP level, but is pushed into the SOAP message[7].

5.2 Dealing with Heterogeneity

The Web is a rather uniform client/server environment, where all components (e.g., browsers, Web servers, proxies, clients) speak the same protocol: HTTP. Heterogeneity is more related to the competition between different browser vendors that would lead, for

[7]Unless the `SOAP-Action` HTTP header is employed, which is controversial and discouraged by the WS-I Basic Profile.

Architectural Principle and Aspects	REST	WS-*
Protocol Layering	yes	yes
HTTP as application-level protocol	✓	
HTTP as transport-level protocol		✓
Dealing with Heterogeneity	yes	yes
Browser Wars	✓	
Enterprise Computing Middleware		✓
Loose Coupling, aspects covered	yes, 2	yes, 3
Time/Availability		✓
Location (Dynamic Late Binding)	(✓)	✓
Service Evolution:		
Uniform Interface	✓	
XML Extensibility	✓	✓
Total Principles Supported	**3**	**3**

Table 1: Principles Comparison Summary

instance, to different renderings of HTML pages or incompatible JavaScript libraries. Still, all browsers support the same HTTP protocol and can process a large variety of standard document types.

SOAP and WS-* originate from a more complex and heterogeneous domain, the one of enterprise computing. This domain can be characterized as a collection of heterogeneous, autonomous, distributed software systems [7]. In organizations having a long history, which may even predate the advent of the Web, these software systems are implemented in many different kinds of technologies. Examples are legacy COBOL programs running on mainframes that require screen-scraping, transaction processing over Message Queuing (MQ)-based reliable data feeds, and distributed object-oriented programs adhering to the Common Object Request Broker Architecture (CORBA).

From these examples, it can be seen that the heterogeneity addressed by WS-* standardization efforts goes beyond the synchronous client/server protocols used in specific RPC middleware products, but extends to delivering the interoperability needed to build the plumbing for SOAs integrating technically diverse enterprise applications.

5.3 Defining Loose Coupling

Both WS-* and REST foster the development of *loosely coupled* distributed systems, albeit according to different definitions of this rather overloaded term. There are many aspects to loose coupling: time/availability, location transparency, and service evolution are important dimensions.

A very important aspect of loose coupling concerns the ability for service consumers to interact with a service provider even when the latter is not available (the *time/availability aspect* of loose coupling). Thus, in such kind of loosely coupled system, clients are not affected when services suffer from temporary downtime. When WS-* is used to implement message-based (as opposed to RPC-based) SOAs, the underlying message bus makes it possible to achieve such degree of loose coupling as messages can be transferred using persistent, reliable queues. Since RESTful Web services exclusively focus on RPC-like, synchronous interactions, this technology cannot hide such failure scenarios. In other words, when an HTTP server is down, its clients will be affected as their HTTP requests fail; they have to handle such connection timeout failures themselves.

Synchronous interactions can nevertheless show a form of loose coupling, when clients discover the actual location of service providers at runtime. This *location aspect* of loose coupling (dynamic late binding) is supported by most WS-* toolkits thanks to the standardization of service registry lookup APIs. In REST, a form of location transparency can be achieved by DNS address translation, requiring to additional efforts to properly configure the networking infrastructure.

When it comes to managing the *evolution of a Web service*, loose coupling implies the ability to make modifications to a Web service without affecting its clients. In case of RESTful Web services, it is clear that the uniform "4 verbs" `GET`, `POST`, `PUT`, `DELETE` are the same for all services and never change. Freezing the basic protocol allows for complete decoupling of clients from servers, as no change on the server will ever break a client, as such change simply does not happen. At first glance, RESTful Web services might appear to be "more" loosely coupled, as each WS-* service interface publishes a different set of operations and such interface may change over time. However, independent of whether the set of operations is fixed or not, another important part of a service interface definition is the message parameter data model, which specifies the format (i.e., the syntax, structure, and in some cases also the semantics) of the message payloads for each operation. In this case, both WS-* and RESTful POX/HTTP services share the same loose coupling properties of XML [9]. As opposed to a binary protocol, the usage of SOAP or POX messages can give the ability to *slightly* modify a service interface in non-breaking ways [42].

In summary, as shown in Table 1, both styles support all three principles, albeit with different definitions and interpretations of the aspects involved. Therefore, it is too early to reach a conclusion and we need to continue with our comparison at the conceptual and technology levels.

6. CONCEPTUAL COMPARISON

Table 2 summarizes our conceptual comparison. From a conceptual point of view, WS-* and REST support a different set of *integration styles* as it was discussed in Section 1.

WS-* and REST take a different approach to the definition of the Web service interfaces. In this section, we therefore concentrate on the most important architectural decisions that have to be made when following the respective *contract design* methodologies.

The importance of having a well-defined, machine processable interface description has been understood for a long time [6, 24]. Two different practices have emerged in the WS-* community. Contract-*first* prescribes to begin the development of a service from the specification of its interface, whereas Contract-*last* takes a bottom-up approach, where an existing service implementation is published with an automatically generated contract.

REST constrains the interface of a resource to its generic uniform interface with predefined operations. Thus, apparently no decision has to be made concerning what are the available operations (i.e., contract-*less* development). Designers are advised to concentrate their effort on defining the exposed resources. However, it is still necessary to: 1) assess whether all four verbs are applicable to each resource exposed by the RESTful Web service[8] and 2) establish the exact semantics of applying each verb on the resource.

Comparing the complexity of Web service interface design, we can conclude that REST appears to be simpler as it completely constrains the set of operations. However, the designer is still required to enumerate the set of resources (or the set of operations, for WSDL-based services) to be provided by a Web service. These are two different kinds of artifacts; however, a non-negligible design effort is still required in both cases.

[8]For which resource/verb combination should "405 Method Not Allowed" be returned to the client?

Architectural Decision and AAs	REST	WS-*
Integration Style	1 AA	2 AAs
Shared Database		
File Transfer		
Remote Procedure Call	✓	✓
Messaging		✓
Contract Design	1 AA	2 AAs
Contract-first		✓
Contract-last		✓
Contract-less	✓	
Resource Identification	1 AA	n/a
Do-it-yourself	✓	
URI Design	2 AA	n/a
"Nice" URI scheme	✓	
No URI scheme	✓	
Resource Interaction Semantics	2 AAs	n/a
Lo-REST (`POST`, `GET` only)	✓	
Hi-REST (4 verbs)	✓	
Resource Relationships	1 AA	n/a
Do-it-yourself	✓	
Data Representation/Modeling	1 AA	1 AA
XML Schema	(✓)[a]	✓
Do-it-yourself	✓	
Message Exchange Patterns	1 AA	2 AAs
Request-Response	✓	✓
One-Way		✓
Service Operations Enumeration	n/a	≥3 AAs
By functional domain		✓
By non-functional properties and QoS		✓
By organizational criterion (versioning)		✓
Total Number of Decisions, AAs	**8**, 10	**5**, ≥10

[a]Optional

Table 2: Conceptual Comparison Summary

Although the methodology used to design a RESTful Web service is rather similar to the one used to design a WSDL-based Web service, it requires architects and developers to make decisions that deal with different abstractions. To help with the comparison we outline these decisions in the following. RESTful Web service design entails making these decisions (Table 2):

1. *Resource Identification*: What are the resource abstractions (e.g., a book catalog, a purchase order, a bank account) exposing an application as a Web service?

2. *URI Design*: Each resource should be addressed using a "nice" URI scheme [34]. Properties of such URIs include: durability [4], predictability, conciseness, reification (prefer nouns to verbs), readability, consistency, and abstraction from implementation details [26].

3. *Resource Interaction Semantics*: Decide which of the four verbs are applicable to a resource. For example, should it be allowed to `POST` on a resource representing a bank account? In general, this decision can be constrained by distinguishing whether accessing a resource is a read-only operation free of side-effects – and thus `GET` should be used, or instead accessing a resource involves modifying its state – thus one of the `PUT`, `POST`, `DELETE` verbs should be used.

4. *Resource Relationships*: Resources should be connected by linking them into several kinds of relationships (e.g., ownership, reference, or containment). This is useful to incrementally discover the interface of a RESTful Web service by traversing hyperlinks between each of its resource representations. Likewise, the content of a resource can be revealed gradually, allowing interested clients to follow links to drill-down for more details. Also, relationships may be used to represent correct state transitions. Thus clients can correctly interact with a stateful resource as they navigate to the next states listed in the current state's representation.

5. *Data Representations*: The content-type of a resource must be chosen among a set of standard formats. If an XML-based representation is chosen, it is not mandatory to constrain the content using a schema.

The design process behind a WS-* Web service centers upon the definition of its interface using the WSDL language, which requires to make the following decisions (Table 2):

1. *Data Modeling*: Structure the content of the XML messages exchanged by the service using XML Schema data types.

2. *Message Exchange Patterns*: Determine whether each operation uses a synchronous or an asynchronous interaction pattern – an outgoing message is only required for synchronous message exchange.

3. *Service Operations Enumeration*: Define the set of actions exposed by the service interface. A WSDL port type can contain more than one operation; the service modeler can group related operations by functional area (business domain), by nonfunctional properties such as security requirements (e.g., authorization profile), or by organizational factors such as change frequency or version management patterns. The rationale behind these grouping criteria is to achieve high cohesion between the service operations so that as few clients as possible break when the service interface evolves.

The design of the interface of WS-* Web services is a very important decision, as each service is characterized by a specific set of operations, listed in its WSDL description. The port type for each service must be designed carefully, so that it describes the service functionality in an understandable way; no pre-defined semantics for its operations is available. In this regard, many extensions for annotating service descriptions with additional semantics metadata have been proposed (e.g., SA-WSDL, or WSDL/S). However, no standard has been agreed upon yet.

There is a tradeoff between modularity/reusability and performance [8]. It is still a challenge to design a reusable Web service interface, which requires a minimal number of interactions to accomplish a certain goal with all of its potential clients [37].

As summarized in Table 2, eight conceptual architectural decisions with only ten alternatives are required for REST. For five decisions, only one alternative exists. This freedom from choice leads to substantial design and development efforts for decisions with a single do-it-yourself alternative. Rather surprisingly, the number of decisions is lower for WS-* (five). However, there are many more alternatives for these decisions (more than ten). Hence, WS-* provides freedom-of-choice within rather strict conceptual boundaries established by the specifications. Moreover, the alternatives are easier to implement due to the standardization of the concepts and the available tool support.

Architectural Decision and AAs	REST	WS-*
Transport Protocol	1 AA	≥7 AAs
HTTP	✓	✓[a]
waka [13]	(✓)[b]	
TCP		✓
SMTP		✓
JMS		✓
MQ		✓
BEEP		✓
IIOP		✓
Payload Format	≥6 AAs	1 AA
XML (SOAP)	✓	✓
XML (POX)	✓	
XML (RSS)	✓	
JSON [10]	✓	
YAML	✓	
MIME	✓	
Service Identification	1 AA	2 AA
URI	✓	✓
WS-Addressing		✓
Service Description	3 AAs	2 AAs
Textual Documentation	✓	
XML Schema	(✓)[c]	✓
WSDL	✓[d]	✓
WADL [18]	✓	
Reliability	1 AA	4 AAs
HTTPR [38][e]	(✓)	(✓)
WS-Reliability		✓
WS-ReliableMessaging		✓
Native		✓
Do-it-yourself	✓	✓
Security	1 AA	2 AAs
HTTPS	✓	✓
WS-Security		✓
Transactions	1 AA	3 AAs
WS-AT, WS-BA		✓
WS-CAF		✓
Do-it-yourself	✓	✓
Service Composition	2 AAs	2 AAs
WS-BPEL		✓
Mashups	✓	
Do-it-yourself	✓	✓
Service Discovery	1 AAs	2 AAs
UDDI		✓
Do-it-yourself	✓	✓
Implementation Technology	many	many
...	✓	✓
Total Number of Decisions, AAs	**10**, ≥17	**10**, ≥25

[a] Limited to only the verb `POST`
[b] Still under development
[c] Optional
[d] WSDL 2.0
[e] Not standard

Table 3: Technology Comparison Summary

7. TECHNOLOGY COMPARISON

Table 3 summarizes the third step of our REST vs. WS-* comparison. Unlike in the conceptual comparison, the number of decisions required is the same (ten). However, again WS-* offers more alternatives. For REST, five decisions offer only one alternative.

7.1 Transport Protocol

The Web is defined by its protocol: HTTP, thus when choosing to build a RESTful Web service, there is no choice but to build services that communicate using HTTP[9]. Thus, given the lack of options, no decision is necessary when it comes to choosing the communication protocol.

One of the properties of WS-*, instead, is its *transport independence*, which allows SOAP messages to be exchanged using a variety of transport protocols. The WSDL binding element is used to select an appropriate transport protocol (HTTP being one possibility) to bind the operation messages. The corresponding QoS, security, and transactional policies also have to be defined. Table 3 shows a few examples of available transport protocols. With this approach, messages in a standardized format can be transported using the most suitable protocol [46]. On the one hand, both synchronous (e.g., HTTP) and asynchronous protocols (e.g., JMS) are available, making SOAP suitable for the implementation of both request-response and one-way message exchange patterns. On the other hand, this creates the need for devising mechanisms such as SOAP headers, delivering certain QoS properties at the message level. For example, since a SOAP message could be routed both over secure and non-secure communication channels, it has been suggested that the message itself should be protected in order to guarantee certain end-to-end security properties of the communication.

7.2 Payload Format

For WS-* Web services, a single standardized message format exists: SOAP. On the other hand, RESTful Web services currently do not use a single format for representing resources. Instead they rely on the flexibility provided by the content negotiation features of REST to choose between a variety of MIME document types – which may also include SOAP itself[10]. This can complicate and hinder the interoperability of a RESTful Web service, as – for example – clients expecting JSON [10] data will not be able to parse a XML payload. Also, a RESTful Web service capable of serving resources in multiple representation formats requires more maintenance effort. However, in our experience the benefit of preferring JSON over XML can outweigh the extra effort and the lack of interoperability with a significant overhead reduction.

7.3 Service Identification

RESTful Web services leverage the URI standard as the naming mechanism to address resources. The advantage of URIs is that they encapsulate all information required to identify and locate a resource on a global addressing space without the need for a centralized registry. Furthermore, URIs can be bookmarked, exchanged via hyperlinks and, given their readability, even printed on billboards for advertising [26].

WS-* initially lacked a standard addressing mechanism and also relied on URIs for identification of messaging end-points and WSDL service interface descriptions. More recently, WS-Addressing [44] was introduced to represent addressing information through the definition of "end-point references". This standard language aug-

[9] The waka [13] protocol being still under development
[10] IETF RFC 3902, `application/soap+xml`.

ments the information stored in URIs with additional metadata, but lacks the conciseness and readability properties that ensured the success of URIs.

7.4 Service Description

Whereas WS-* Web services rely on a standard, machine-processable, strongly-typed XML interface description language (WSDL), RESTful Web services have adoped a more human-oriented approach based on informal, textual descriptions, giving developers extensive documentation of the API of the provided service [32].

As a consequence, WSDL tools can automatically generate client stub code for most programming languages, masking the complexity of remotely interacting with a service. For RESTful Web services, developers have to manually write the code to assemble the resource URIs and correctly encode/decode the exchanged resource representations. Having good documentation greatly helps to reduce the amount of trial and error involved in the process, but is no substitute for using a compiler of a "real" interface description language. To address this shortcoming, the Web Application Description Language (WADL [18]) was recently proposed. Also the latest WSDL version 2.0 could be applied to describe RESTful Web services, thanks to its more fine-grained control over the HTTP binding and the possibility of supporting non-SOAP message encodings.

In general, having an interface description language to specify service contracts is not only beneficial for simplifying repetitive development tasks, but also helps to catch incompatibilities caused by changes of service interfaces early in the development process. Thanks to the strong typing features of WSDL, clients will break at compile time. This would also benefit RESTful Web services, insofar changes to the URI naming scheme[11] and resource representations are concerned, since due to the uniform interface principle the actual set of operations applicable to a resource never changes (as already discussed).

7.5 Reliability, Security, Transactions

The WS-* stack contains a set of optional specifications covering the QoS properties of messages exchanged. These help to guarantee a certain quality level in the communication, in a way which is independent of the underlying transport protocol and which can be handled automatically by the WS-* middleware. In terms of the effort involved, developers may declare the desired security/reliable messaging/transactional policies associated with a Web service interface and reuse the corresponding WS-* standard implementation to deliver them. While the WS-* specifications related to QoS indeed are complex, they should be considered as optional technology choices addressing advanced requirements in enterprise computing scenarios.

For simpler scenarios, the basic guarantees of protocols such as HTTP (best effort) and HTTPS (point-to-point SSL security) are shared by both REST and WS-* styles. A reliable extension to HTTP (HTTPR [38]) was proposed, but did not complete the standardization process. No framework for distributed transactions comparable to WS-* has been proposed in conjunction with REST.

7.6 Service Composition

Fostering service reuse by means of service composition is one of the central tenets of SOA. The standardization of the invocation path to a Web service has greatly facilitated their reuse and composition, as opposed to the traditionally fragmented markets for software components (e.g., where EJBs can not be easily mixed with .NET assemblies). This has also led to the emergence of a large number of languages and tools specifically targeting the composition of WS-* services out of already existing ones (e.g., WS-BPEL [29], JOpera [31], or XL [15]).

Given their lack of formally described interfaces and the possibility of not always using XML messages, RESTful Web services are cumbersome to compose using the WSDL-based invocation abstractions provided by WS-BPEL [30]. The composition of RESTful Web services is the main focus of so-called Web 2.0 Mashups, which are seen as a welcome improvement over screen-scraping the information to be composed out of traditional HTML Web sites [48].

7.7 Service Discovery

Do-it-yourself build-time lookup is a common choice for both RESTful and for WS-* Web services. Many successful large-scale industry deployments of Web services use groupware or Web portals supported by databases as their service repositories [54].

WS-* offers Universal Description, Discovery, and Integration (UDDI) registries as an additional technology option. While this technology has been mature and stable for several years now, it has failed to reach widespread acceptance in the industry. Several proprietary WSDL-based registry and repository products exist.

7.8 Implementation Technology

A fairly large number of application server platform options exists both for hosting RESTful and WS-* Web services implemented in any programming language. Likewise, client-side library support for consuming RESTful and WS-* Web services is also available in most programming languages and operating systems. Due to space limitations we cannot present a complete survey comparing all available implementation technologies. However, it is worth mentioning that most existing WS-* Web services tools are currently being extended to also provide support for REST.

8. RELATED WORK

SOAP vs. REST has been an ongoing discussion for some time on the blogosphere and has also recently gained attention in the academic community. None of the existing work employs a structured and detailed comparison method based on architectural decisions.

For example, the ECOWS 2005 keynote [17] focused on the reconciliation of WS-* with REST, whereas [55] gives a comparison of the two approaches from the point of view of their application to workflow systems. A good discussion on whether the Web (and in particular RESTful Web services) can fulfill the requirements of enterprise computing can be found in a recent W3C workshop [23].

A comparison of RESTful Web services and so-called "Big Web Services" is also found in Chapter 10 of [33]. In it, a critical look to the WS-* stack is given in terms of how it does not fit with the "resource-oriented" paradigm of the Web. The chapter also attempts to show how simpler RESTful techniques can be used to replace the corresponding WS-* technologies. The distinction between "resource-oriented" and "service-oriented" architectures was first introduced by [36]. Unfortunately, the book does not provide a clear definition of the terms services and resources, and its technology comparison is not based on measurable, objective criteria such as software quality attributes, design and development effort, technical risk, and QoS characteristics.

Even if HTTP is a synchronous protocol, the comparison presented in [22] argues that RESTful calls are asynchronous from an application layer perspective. Thus, REST can be seen as favorable solution for simple integration scenarios. Additional architectural concerns such as the URL design and payload format are not discussed.

[11] HTTP redirection can only work up to a certain point.

In [51] we applied architectural decision modeling concepts to another complex design issue in SOA, that of designing transactional workflows. We touched upon the transport protocol selection decision, but did not investigate it in detail. The paper focused on the rationale behind architectural decision modeling and the usage of decision models as a design method; it did not serve as a technology comparison metric as in this paper.

A framework for the comparison of pre-Web services middleware infrastructures (i.e., CORBA vs. J2EE vs. COM+) has been introduced in [49], with the goal of supporting the choice of the most suitable framework given certain application requirements.

9. CONCLUSION

In this paper we used architectural principles and decisions as a comparison method to illustrate the conceptual and technological differences between RESTful Web services and WSDL/SOAP-based "Big" Web services. On the principle level, the two approaches have similar quantitative characteristics. On the conceptual level, less architectural decisions must be made when deciding for WS-* Web services, but more alternatives are available. On the technology level, the same number of decisions must be made, but less alternatives have to be considered when building RESTful Web services. Thus, the perceived simplicity of REST now can be understood from a quantitative perspective – choosing REST removes the need for making a series of further architectural decisions related to the various layers of the WS-* stack and makes such complexity appear as superfluous. Still, if advanced functionality as delivered by WS-* is needed, it will be no simple matter to extend a RESTful Web service to support it in an interoperable manner. Furthermore, according to our experience several of the decisions that are very easy to make for RESTful services lead to significant development efforts and technical risk, for example the design of the exact specification of the resources and their URI addressing scheme.

From our comparison it can also be seen that the two styles are rather similar, as long as the same subset of technology decisions is compared, for example when ignoring the more advanced features of the WS-* stack and only comparing POX/HTTP and SOAP/HTTP. It could even be argued that these two approaches are two technology-level variants of the same conceptual design.

Assuming that the enterprise-level features of WS-* (transactions, reliability, message-level security) are not required, the key decision drivers at present are degree of flexibility and control, but also development efforts and technical risk (in implementation design, development, and maintenance), degree of open source and vendor tool support, and programming interface convenience. REST scores better with respect to flexibility and control, but requires a lot of low-level coding; WS-* provides better tool support and programming interface convenience, but introduces a dependency on vendors and open source projects. The main conclusion from this comparison is to use RESTful services for tactical, ad hoc integration over the Web (*à la Mashup*) and to prefer WS-* Web services in professional *enterprise application integration* scenarios with a longer lifespan and advanced QoS requirements.

Acknowledgements

The authors would like to thank Domenico Bianculli, Christopher Ferris, Joachim Hagger, Jana Koehler, Noah Mendelsohn, David Nüscheler, James Snell, Stefan Tramm, and Tammo van Lessen for their invaluable input and support. This work is partially supported by the EU-IST-FP7-215605 (RESERVOIR) project.

10. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures, Applications*. Springer, 2004.

[2] Apache. *Axis2*. `http://ws.apache.org/axis2/`.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 2003.

[4] T. Berners-Lee. *Cool URIs don't change*, 1998. `http://www.w3.org/Provider/Style/URI.html`.

[5] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): generic syntax*. IETF RFC 3986, January 2005.

[6] A. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2:39–59, February 1984.

[7] C. Bussler. *B2B Integration*. Springer, June 2003.

[8] W. R. Cook and J. Barfield. Web services versus distributed objects: A case study of performance and interface design. In *Proc. of the IEEE International Conference on Web Services (ICWS2006)*, Chicago, USA, September 2006.

[9] F. P. Coyle. *XML, Web Services, and the Data Revolution*. Addison-Wesley, May 2002.

[10] D. Crockford. JSON: The fat-free alternative to XML. In *Proc. of XML 2006*, Boston, USA, December 2006. `http://www.json.org/fatfree.html`.

[11] Eclipse. *Web Tools Platform (WTP) Project*. `http://www.eclipse.org/webtools/`.

[12] R. Fielding. *Architectural Styles and The Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[13] R. Fielding. waka: A replacement for HTTP. In *APACHECON US*, November 2002. `http://www.apache.org/~fielding/waka/`.

[14] R. Fielding. *A little REST and Relaxation*. The International Conference on Java Technology (JAZOON07), Zurich, Switzerland, June 2007. `http://www.parleys.com/display/PARLEYS/A\%20little\%20REST\%20and\%20Relaxation`.

[15] D. Florescu, A. Gruenhagen, and D. Kossmann. XL: An XML programming language for Web service specification and composition. In *Proc. of the 11th International World Wide Web Conference (WWW2002)*, Honululu, Hawaii, USA, May 2002.

[16] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[17] H. Haas. Reconciling Web services and REST services (Keynote Address). In *Proc. of the 3rd IEEE European Conference on Web Services (ECOWS 2005)*, Växjö, Sweden, November 2005.

[18] M. J. Hadley. *Web Application Description Language (WADL)*, 2006. `http://wadl.dev.java.net/`.

[19] D. Hansson. Keynote. In *Canada on Rails*, January 2006.

[20] G. Hohpe. *Enterprise Integration Patterns*. Addison-Wesley, October 2003.

[21] IETF. *HTTP*, 1999. `http://www.ietf.org/rfc/rfc2616`.

[22] E. Landre and H. Wesenberg. Rest versus soap: as architectural style for web services. In *5th International OOPSLA Workshop on SOA & Web services Best Practices*, 2007.

[23] K. Laskey, P. L. Hègaret, and E. Newcomer, editors. *Workshop on Web of Services for Enterprise Computing*. W3C, February 2007. http://www.w3.org/2007/01/wos-ec-program.html.

[24] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[25] R. Monson-Haefel. *J2EE Web Services*. A-W., 2003.

[26] J. Nielsen. *URI as UI*, March 1999. http://www.useit.com/alertbox/990321.html.

[27] OASIS. *Organization for the Advancement of Structured Information Standards*. http://www.oasis-open.org/.

[28] OASIS. *Web Services Resources Framework (WSRF 1.2)*, April 2006. http://www.oasis-open.org/committees/wsrf/.

[29] OASIS. *Web Services Business Process Execution Language*, April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[30] H. Overdick. Towards resource-oriented bpel. In *2nd ECOWS Workshop on Emerging Web Services Technology*, November 2007.

[31] C. Pautasso and G. Alonso. The JOpera visual composition language. *Journal of Visual Languages and Computing (JVLC)*, 16(1-2):119–152, 2005.

[32] Programmable Web. *API Dashboard*, 2007. http://www.programmableweb.com/apis.

[33] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007.

[34] P. Seebach. *Making URLs accessible*, June 2001. http://www.ibm.com/developerworks/library/us-cranky8.html.

[35] R. Sessions. Fuzzy boundaries: Objects, components, and web services. *ACM Queue*, 2(9), December/January 2004-2005.

[36] J. Snell. *Resource-oriented vs. activity-oriented Web services*. IBM developerWorks, October 2004. http://www-128.ibm.com/developerworks/webservices/library/ws-restvsoap/.

[37] T. Takase and K. Tajima. Efficient Web services message exchange by SOAP bundling framework. In *Proc. of 11th IEEE International EDOC Conference (EDOC 2007)*, October 2007.

[38] S. Todd, F. Parr, and M. Conner. *A Primer for HTTPR*, July 2001. http://www.ibm.com/developerworks/webservices/library/ws-phtt/.

[39] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, 2005.

[40] S. Vinoski. Serendipitous reuse. *IEEE Internet Computing*, 12(1):84–87, 2008.

[41] S. Vinoski. Putting the "Web" into Web services: Interaction models, part 1: Current practice. *IEEE Internet Computing*, 6(3):89–91, May-June 2002.

[42] S. Vinoski. Putting the "Web" into Web services: Interaction models, part 2. *IEEE Internet Computing*, 6(4):90–92, July 2002.

[43] W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, Nov-Dec 2003.

[44] W3C. *Web Services Addressing*, May 2006. http://www.w3.org/2002/ws/addr/.

[45] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture*. Prentice Hall, March 2005.

[46] C. Werner, C. Buschmann, T. Jaecker, and S. Fischer. Enhanced transport bindings for efficient SOAP messaging. In *Proc. of the 3rd IEEE International Conference on Web Services (ICWS'05)*, Orlando, FL, USA, July 2005.

[47] WS-I. *Web Services Interoperability*. http://www.ws-i.org.

[48] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. A Framework for Rapid Integration of Presentation Components. In *Proc. of the 16th International World Wide Web Conference. Banff, Canada*, May 2007.

[49] A. Zarras. A comparison for middleware infrastructures. *Journal of Object Technology*, 3(5):103–123, May-June 2004.

[50] O. Zimmerman, M. Tomlinson, and S. Peuser. *Perspectives on Web Services: Applying SOAP, WSDL, and UDDI to Real-World Projects*. Springer, September 2003.

[51] O. Zimmermann, J. Grundler, S. Tai, and F. Leymann. Architectural decisions and patterns for transactional worlflows in SOA. In *Proc. of the 5th International Conference on Service-Oriented Computing*, Vienna, Austria, 2007.

[52] O. Zimmermann, T. Gschwind, J. Kuester, F. Leymann, and N. Schuster. Reusable architectural decision models for enterprise application development. In *Quality of Software Architecture (QoSA) 2007*, Boston, USA, July 2007.

[53] O. Zimmermann, J. Koehler, and F. Leymann. Architectural decision models as micro-methodology for service-oriented analysis and design. In *SEMSOA Workshop*, Hannover, Germany, May 2007.

[54] O. Zimmermann, M. Milinski, M. Craes, and F. Oellermann. Second generation web services-oriented architecture in production in the finance industry. In *OOPSLA Conference Companion*, 2004.

[55] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson. Developing Web services choreography standards - the case of REST vs. SOAP. *Decision Support Systems*, 40(1):9–29, July 2005.