Technical University of Vienna
Information Systems Institute
Distributed Systems Group

DAIOS – Efficient Dynamic Web Service Invocation

Philipp Leitner, Florian Rosenberg and Schahram Dustdar
leitner@infosys.tuwien.ac.at
florian@infosys.tuwien.ac.at
dustdar@infosys.tuwien.ac.at

TUV-1841-2007-01 November 2, 2007

Systems based on the Service-Oriented Architecture (SOA) paradigm need to be able to bind to arbitrary (Web) services at run-time. However, current service frameworks are predominantly used through pre-compiled service access components, which are invariably hardwired to a specific service provider. In this paper, we present DAIOS*– a massage-based service framework that supports implementation of SOAs, enabling dynamic invocation of SOAP/WSDL-based and RESTful services. It abstracts from all internals of the target service, allowing clients to be decoupled from services they use.* DAIOS*' runtime performance is on par with the current state of the art – and is, therefore, a feasible solution for developers striving to implement dynamic SOAs.*

Keywords: Web services, Dynamic Invocation

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
URL: http://www.infosys.tuwien.ac.at/

DAIOS– Efficient Dynamic Web Service Invocation

Philipp Leitner, Florian Rosenberg, Schahram Dustdar
VitaLab, Technical University of Vienna
Argentinierstrasse 8/184-1
A-1040, Vienna, Austria
lastname@infosys.tuwien.ac.at

Abstract

Systems based on the Service-Oriented Architecture (SOA) paradigm need to be able to bind to arbitrary (Web) services at run-time. However, current service frameworks are predominantly used through pre-compiled service access components, which are invariably hardwired to a specific service provider. In this paper, we present DAIOS– *a massage-based service framework that supports implementation of SOAs, enabling dynamic invocation of SOAP/WSDL-based and RESTful services. It abstracts from all internals of the target service, allowing clients to be decoupled from services they use.* DAIOS*' runtime performance is on par with the current state of the art – and is, therefore, a feasible solution for developers striving to implement dynamic SOAs.*

1. Introduction

Software systems built on top of Service-Oriented Architectures (SOAs) [1] intend to use a triangle of three operations, publish, find and bind, to decouple all roles participating in the system. Two elements of this triangle, publish and find particularly, put requirements on the service registry and the interface definition language: to publish services, an expressive and extensible service definition language has to be available and to be supported by the service registry. These issues are pursued within the VReSCO [2] project. The third operation, bind, is independent from the service registry: binding has to be handled solely by the service consumer. It is essential for the implementation of a SOA that the consumer is able to connect to any service that he might discover during the find step, and that it is possible to change this binding at any time (specifically at run-time of the system) if the original target service becomes unavailable or services delivering a more suiting Quality of Service level are found.

In this article we will present DAIOS, a message-based dynamic service invocation framework that enables application developers to create service clients which are not coupled to any specific provider. We will detail the requirements that drove the design of our prototype, the architecture of our solution and explain how our approach solves the present issues that we currently encounter when building SOAs with state of the art technology. Finally, we will also present an evaluation of our DAIOS prototype with regard to functional and runtime performance aspects.

2. Dynamic Service Invocation

Dynamic binding is not easy with current state of the art Web service client frameworks such as Apache Axis 2 or Apache WSIF. These frameworks strictly rely on client-side stubs to invoke services, which are usually autogenerated at design-time. However, stubs are invariably hardwired to a specific service provider and cannot be changed at run-time. This is a severe problem for realizing a SOA: if service providers are hardwired into the service consumers' application code, producers and consumers cannot by any means be considered loosely coupled. The usage of client stubs does not follow the idea of SOA, since find as well as bind are in that case actually carried out by the developer. A client application relying on pre-compiled stubs cannot implement a SOA. We, therefore, conclude that the SOA triangle is currently "broken" [2].

Additionally, existing Web service client frameworks such as Apache Axis 2 and Apache WSIF often suffer from a few further misconceptions. They are often built to be as similar as possible to earlier distributed object middleware systems [3], implying a very strong emphasis on RPC-centric and synchronous Web services. SOAs, on the other hand, are centered on the notion of asynchronous exchange of business documents.

2.1. Requirements

Taking into account the fundamental maladies of currently available Web service client-side solutions we define the following requirements for a Web service invocation framework that supports the core SOA ideas:

- *Stubless service invocation*: Given that generated stubs entail a tight coupling of service provider and service consumer the invocation framework shall not rely on any static components such as client-side stubs or data transfer objects. Instead the framework should be able to invoke arbitrary Web services through a single interface, using generic data structures.

- *Protocol-independent*: Web service standards and protocols have not yet fully settled. There is still ongoing discussion about the advantages of the REST [4] architecture as compared to the more common SOAP and WSDL-based [5, 6] approach to Web services. The framework should therefore be able to abstract from the underlying Web service protocol, and support at least SOAP-based and REST-based services transparently.

- *Message-driven*: Currently Web services are often seen as collections of platform-independent remote methods. The framework shall be able to abstract from this RPC style which usually leads to tighter coupling, and follow a message-driven approach instead.

- *Support for asynchronous communication*: In a SOA, services might take a long time to process a single request. The currently prevalent request/response style of communication is not suitable for such long-running transactions. The framework shall therefore also support asynchronous (non-blocking) communication.

- *Simple API*: Current dynamic invocation interfaces are often not intuitive to use. The framework shall utilize a message-driven approach to make the API to the user as simple as possible.

- *Acceptable runtime behavior*: The framework shall not imply sizable overhead on the Web service invocation. Using the framework shall not be significantly slower than using any of the existing Web service frameworks.

Unfortunately, current Web service frameworks cannot fully live up to these requirements (see Section 5 for details).

3. Related Work

The first Java-based Web service framework that incorporated the idea of dynamic service invocation was the Apache project Web Service Invocation Framework (WSIF) [7]. The WSIF dynamic invocation interface is intuitive to use as long as the client application knows the signature of the WSDL operation to invoke. We consider this to be a unacceptable precondition for loosely-coupled SOAs – client applications should not have to care about service internals such as the concrete operation name. Another big caveat of WSIF is its notoriously weak support for complex XML Schema types as service parameters or return values. Complex types can only be used if they are "mapped" to an existing Java object beforehand, what is frequently impossible in dynamic invocation scenarios. These problems, along with the fact that the framework is not under active development since 2003 and the relatively bad runtime performance, render WSIF outdated today.

The Apache Axis 2 [8] framework incorporates a lot more SOA concepts than WSIF: it supports client-side asynchrony and works much more on a document level than the strictly RPC-based WSIF. Event though Axis 2 is still grounded on the usage of client-side stubs it also supports dynamic invocations through the `OperationClient` or `ServiceClient` APIs. The disadvantage of these interfaces is that they expect the client application to create the entire payload of the invocation (e.g., the SOAP body) itself. In that case Axis 2 does little more than transfer the invocation to the server. This is not the level of abstraction that we expect from a Web service framework used to construct a SOA client. However, we recognize that the Axis 2 SOAP and REST stacks are well developed and highly performant. We therefore created a Axis 2 service backend as part of our DAIOS prototype in order to combine the advantages of DAIOS and Axis 2: the Axis 2 backend uses the dynamic invocation abstraction of DAIOS, but utilizes the Axis 2 service stack to carry out the actual invocation. Similar problems as present in Axis 2 arise with other recent service frameworks such as Codehaus XFire [9] or XFire's successor, Apache CXF [10]. Ultimately, all of these client-side frameworks are relying on static components to access Web services, with little to no support for truly dynamic invocation scenarios.

JAX-WS (Java API for XML-based Web Services) is the latest Java-based Web service specification. JAX-WS is described in JSR (Java Specification Request) 224 [11], and is the official follow-up to the older JAX-RPC [12]. JAX-WS is implemented for instance in the Apache CXF project, and, therefore, exhibits similar problems – although the change in the naming suggests that JAX-WS is less RPC-oriented than its ancestor the specification still focuses on WSDL-to-operation mappings, ignoring the messaging

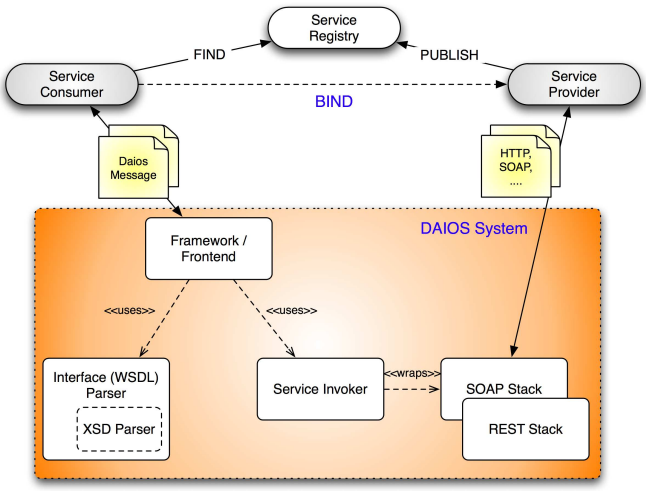

Figure 1. Daios overall architecture

ideas of SOA and Web services. REST is also not discussed explicitly in the JSR, even though the document claims to generally handle XML-based Web services in Java.

A different approach to dynamic service invocation has been introduced by Nagano et al. [13]. They propose to continue using static stubs, but bind them to more generic interfaces instead of "precise" ones. That way the same stubs could be used to invoke any service with a similar interface, thereby enabling looser coupling between client and provider. The advantage of this approach is that (unlike to our ideas) static type safety can be achieved, but there are also considerable disadvantages: the concept is only feasible for Web services defined using a formalized XML interface (what is currently not the case with most REST-based services), and the practical implementation of more generic interfaces is often a hard problem, and needs a lot of domain knowledge – it is therefore hard to create a generic framework that can be used by SOA clients in any problem domain using this approach.

4. The DAIOS Solution

With the requirements from Section 2.1 in mind we have designed and prototypically implemented the DAIOS (Dynamic and Asynchronous Invocation of Services) framework. DAIOS is a Web service invocation frontend for SOAP/WSDL-based and RESTful services. It supports only fully dynamic invocations without any static components such as stubs, service endpoint interfaces or data transfer objects.

Figure 1 sketches the general architecture of the DAIOS framework, and how it fits into the general SOA triangle of publish, find and bind. The framework internally splits up into three functional components: the general DAIOS classes which are the at core of the framework and orchestrate the individual other components, the interface parsing component which is responsible for preprocessing (binding) and the invoker component which conducts the actual Web service invocations using a REST or SOAP stack. Clients communicate with the framework frontend using DAIOS messages which are DAIOS' internal data representation format. The general structure of the framework is an implementation of the Composite Pattern for stubless Web service invocation (CPWSI) [14]. CPWSI separates the frameworks' interface from the actual invocation backend implementation, and allows for flexibility and adaptability.

DAIOS is grounded on the notion of message exchange: clients communicate with services by passing messages to them; services return the invocation result by answering with messages. DAIOS messages are potent enough to encapsulate XML Schema complex types, but much simpler to use than working directly on XML level. Messages are unordered lists of name-value pairs, referred to as message fields. Every field has an unique name, a type and a value. Valid types are either built-in types (*simple field*), arrays of built-in types (*array field*), complex types (*complex field*) or arrays of complex types (*complex array field*). Such complex types can be constructed by nesting messages – arbitrary data structures can therefore be built easily, without the need for a static type system.

Using DAIOS is generally a three-step procedure:

1. First and foremost, clients have to find a service that they want to invoke (service discovery phase). This step is external to DAIOS– the service discovery problem is mostly a registry issue and has to be handled separately [2].

2. In the second step the service has to be bound (preprocessing phase). During this phase the framework will collect all necessary internal service information, e.g., for a SOAP/WSDL-based service the service's WSDL interface will be compiled in order to obtain endpoint, operation and type information.

3. The final step is the actual invocation of the service (dynamic invocation phase). During this phase the user input (i.e., an input message) will be converted into the encoding expected by the service (for instance a SOAP operation of a WSDL/SOAP-based service, or a HTTP GET request for REST), and the invocation will be launched using a SOAP or REST service stack. When the invocation response (if any) is received by the service stack it will be converted back into an output message and returned to the client.

Once a service is successfully bound clients can of course issue any number of invocations without having to

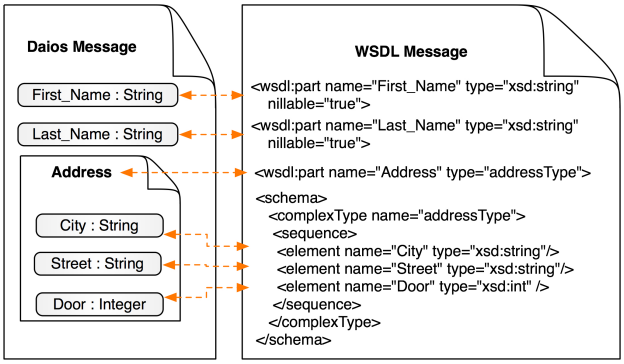

Figure 2. Equivalent DAIOS and WSDL inputs

re-bind again. Service bindings only have to be renewed if the interface contract of the service changes or the client explicitly decides to release the binding for some reason.

Most of DAIOS' important processing happens in the dynamic invocation phase. For a SOAP invocation the framework will analyze the given input and determine which WSDL input message the provided data matches best. For this DAIOS relies on a specific similarity algorithm for DAIOS and WSDL messages. The general idea of this algorithm is to calculate a structural distance metric for the WSDL message and the user input, i.e., count how many parts in a given WSDL message have no corresponding field in the DAIOS message, whereas lower values represent a better match. Figure 2 sketches a DAIOS message and a WSDL message in `RPC/encoded` style with a structural distance of 0 (a perfect match). If for instance the field "First_Name" would be removed from the DAIOS message the structural distance would increase to 1.

DAIOS will choose to invoke the operation whose input messages has the best (i.e., lowest) structural distance metric to the provided data. Only if two or more input messages are equally similar to the input the user has to specify the operation to use. If no input message is suitable at all, that is if all input messages have a similarity metric of ∞ to the input, an error will be thrown – in that case the provided input is simply not suitable for the chosen Web service. Otherwise the framework will convert the input into an invocation of the chosen operation, issue the invocation, receive the result from the service and convert the result back into a message. The backend used to conduct the actual invocation is replaceable: the DAIOS research prototype comes with two options of invocation backends, one which uses the Apache Axis 2 stack and one which utilizes a custom-built ("native") SOAP and REST stack. DAIOS also puts much emphasis on client-side asynchrony. All invocations can be issued in a blocking or non-blocking fashion.

This procedure abstracts most of the RPC-like internals of SOAP and WSDL; the client-side application does not need to know about WSDL operations, messages, endpoints or encoding. Even whether the target service is implemented as SOAP- or REST-based service is transparent to the client. All of these service details are handled solely by DAIOS, allowing the client application to be as generic as possible.

4.1. Usage Examples

```
1  // create a Daios backend
2  ServiceFrontendFactory factory =
3      ServiceFrontendFactory.getFactory
4        ("at.ac.tuwien.infosys.dsg.daiosPlugins."+
5        nativeInvoker.NativeServiceInvokerFactory");
6
7  // preprocessing - bind service
8  ServiceFrontend frontend = factory.createFrontend(
9    new URL(
10     "http://vitalab.tuwien.ac.at/"+
11     "orderservice?wsdl"));
12
13 // construct input that we want
14 // to pass to the service
15 DaiosInputMessage registration
16   = new DaiosInputMessage();
17 DaiosMessage address = new DaiosMessage();
18 address.setString("City", "Vienna");
19 address.setString("Street", "Argentinierstrasse");
20 address.setInt("Door", 8);
21 registration.setComplex("Address", address);
22 registration.setString("First_Name", "Philipp");
23 registration.setString("Last_Name", "Leitner");
24
25 // dynamic invocation
26 DaiosOutputMessage response =
27   frontend.requestResponse(registration);
28
29 // retrieve result
30 String regNr = response.getString("registrationNr");
31 // ...
```

Listing 1. DAIOS SOAP invocation

Using DAIOS is simple to the extreme. Listing 1 displays the Java code necessary to invoke a SOAP/WSDL-based Web service. The message constructed in this example corresponds to the structure depicted in Figure 2. Note that even though the target service uses nested data structures (registrations contain address data) DAIOS does not need any static components such as data transfer objects. All necessary service and type information is collected during the preprocessing phase (lines 8 to 11). When the actual dynamic invocation is fired (lines 26 and 27) the framework will use this information and convert the user-provided input to a concrete Web service invocation. In this example a blocking invocation style is used, but asynchronous communication is handled widely identically.

It is important to note that no SOAP or WSDL specifics such as operation name, endpoint address, or the WSDL encoding style used have to be specified by the client application – DAIOS will abstract from all these service internals

Requirement	DAIOS	WSIF	Axis 2	XFire	CXF
Stubless service invocation:					
Simple types	✓	✓	✓	✓	✓
Arrays of simple types	✓	✓	✓	✓	✓
Complex types	✓	✗	✓	✗	✓
Arrays of complex types	✓	✗	✓	✗	✓
Protocol-independent:					
Transparent protocol integration	✓	✗	✗	✗	✗
SOAP over HTTP support	✓	✓	✓	✓	✓
REST support	✓	✗	✓	✗	✓
Message-driven:					
Document-centric interface	✓	✗	✗	✗	✗
Transparent handling of service internals	✓	✗	✗	✗	✗
Support for asynchronous communication:					
Synchronous invocations	✓	✓	✓	✓	✓
Asynchronous invocations	✓	✗	✓	✗	✓
Simple API:					
Simple to use dynamic interface	✓	✓	✗	✓	✓

Table 1. Functional comparison with regard to the requirements from Section 2.1

and expose a uniform interface, therefore, enabling loose coupling between client and service.

```
1  String myAPIKey = ... // get an API key from Flickr
2
3  // use the native backend
4  ServiceFrontendFactory factory =
5      ServiceFrontendFactory.getFactory
6        ("at.ac.tuwien.infosys.dsg.daiosPlugins."+
7        nativeInvoker.NativeServiceInvokerFactory");
8
9  // preprocessing for REST
10 ServiceFrontend frontend = factory.createFrontend();
11
12 // setting the EPR is mandatory for REST services
13 frontend.setEndpointAddress(
14   new URL("http://api.flickr.com/services/rest/"));
15
16 // construct message
17 DaiosInputMessage in = new DaiosInputMessage();
18 in.setString("method",
19   "flickr.interestingness.getList");
20 in.setString("api_key", myAPIKey);
21 in.setInt("per_page", 5);
22
23 // do blocking invocation
24 DaiosOutputMessage out =
25   frontend.requestResponse(in);
26
27 // convert WS result back
28 // into some convenient Java format
29 DaiosMessage photos =  out.getComplex("photo");
30 // ...
```

Listing 2. DAIOS REST invocation

Listing 2 exemplifies the invocation of a RESTful Web service. In this listing the widely known Flickr REST API[1] is accessed and a list of hyperlinks to the most "interesting" photos is retrieved.

RESTful and SOAP-based services are invoked through the same interface – the code necessary to access the service is practically identical for both types of Web services. The main difference is that no interface definition language similar to WSDL has yet been established for RESTful services, leading to the problem that the user is forced to specify more service details for REST-based invocations (the endpoint address in the example).

5. Evaluation

We have evaluated our prototype against a variety of currently available Web service frameworks: Apache WSIF, Apache Axis 2, Codehaus XFire and Apache CXF. We have compared the frameworks in terms of supported functionality, response times and memory consumption. For reasons of brevity we will only present functional aspects and runtime performance data in this article.

Table 1 depicts how well the candidate frameworks are able to cope with the requirements introducted in Section 2.1. The last of these requirements, *acceptable runtime behavior*, will be presented separately below. We believe that all current service frameworks fail to meet these requirements in some important respects. What we consider to be the core problem is that no currently available solution really embraces the loosely-coupled document-centric approach of SOA – all evaluated solutions are based on an RPC processing model, demanding for explicit knowledge

[1] http://www.flickr.com/services/api/

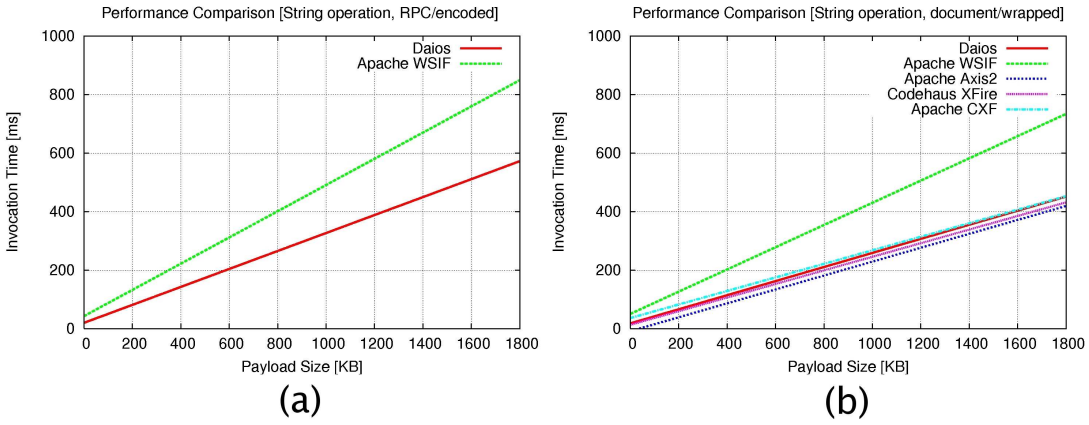

Figure 3. Comparison of invocation response times

of service internals such as WSDL encoding styles, operation signatures or endpoint addresses. Additionally, WSIF and XFire do not provide a fully expressive dynamic invocation interface – user-defined (complex) types cannot be used easily over these interfaces if the types are not known at compile-time. Support for the REST style of Web services is not uncommon today, but a transparent integration of SOAP and REST is not supported. All of these problems are solved in our DAIOS prototype: it exposes a simple messaging interface which can be used to dynamically invoke arbitrary services without knowing implementation details of the service (including whether the service is implemented as a SOAP- or REST-based service), both synchronously or asynchronously.

The last requirement from Section 2.1 is addressed in Figure 3. The figure compares the response times of the candidate frameworks in simple SOAP-based Web service invocations – Figure (a) displays the results for `RPC/encoded` invocations, while Figure (b) does the same for `document/literal` with wrapped parameters. `RPC/encoded` invocations are only evaluated for DAIOS and WSIF – Axis 2, XFire and CXF do not support this particular WSDL encoding style. Note that Apache WSIF is well behind in both test cases; all other candidate frameworks exhibit similar response times. We have also done extensive tests using different types of invocations (with binary or array payload data), but the general result was similar for all tests. According to our tests the same applies to REST-based invocations. We therefore conclude that using DAIOS does not imply a relevant performance penalty over Apache Axis 2, Apache CXF or Codehaus XFire, and that our prototype is significantly faster than Apache WSIF.

6. Conclusions

The SOA vision expects distributed systems to use a triangle of three operations, publish, find and bind, to create a loosely coupled architecture. Services in a SOA have to be selected or substituted at run-time. Unfortunately, today's state of the art client-side service frameworks are not well-suited for run-time service binding: they are strictly based on an RPC programming model and use statically generated service access components, implying a tight coupling of service provider and consumer. The DAIOS framework aims at providing a client-side service framework that is better suited for such scenarios: it provides a fully expressive and easy to use dynamic interface, works entirely message-oriented, has full support for non-blocking communication and supports SOAP and REST-based services through an uniform and simple API. DAIOS is on one level with recent Web service frameworks such as Apache Axis 2, Codehaus XFire or Apache CXF with regard to runtime performance, and is therefore a strong solution for developers facing the dynamic service invocation problem.

6.1. Future Work

Increasingly, Web service implementations use policies to describe non-functional attributes of Web services. Such attributes include security policies, transactional behavior or reliable messaging. For these situations the WS-Policy framework [15] is often employed. It is important for a client to adhere to the policies required by the service provider to allow for a successful interaction with the service. Currently, our implementation does not deal with provider-specific policies, but we plan to add WS-Policy support to our framework to support policy-enforced interactions. Furthermore, we plan to extend our evaluation of the DAIOS framework to a more sizable real-life scenario,

in order to get a more accurate picture of the runtime performance and usability of our implementation in real business applications.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 11, 2007.

[2] A. Michlmayr, F. Rosenberg, C. Platzer, and S. Dustar. Towards Recovering the Broken SOA Triangle - A Software Engineering Perspective. In *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSE'07)*, 2007.

[3] W. Vogels. Web Services Are Not Distributed Objects. *IEEE Internet Computing*, 7(6), 2003.

[4] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, CA, 2000.

[5] World Wide Web Consortium (W3C). SOAP Version 1.2 Part0: Primer. `http://www.w3.org/TR/soap12-part0/`, 2003.

[6] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer - W3C Candidate Recommendation 27 March 2006. `http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327/`, 2006.

[7] Apache Foundation. Web Services Invocation Framework. `http://ws.apache.org/wsif/`.

[8] Apache Foundation. Apache Axis 2. `http://ws.apache.org/axis2/`.

[9] Codehaus XFire. `http://xfire.codehaus.org/`.

[10] Apache Foundation. Apache CXF: An Open Source Service Framework. `http://incubator.apache.org/cxf/`.

[11] D. Kohlert and A. Gupta. Java API for XML-Based Web Services, Version 2. `http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html`, 2007.

[12] JSR-101 Expert Group. Java API for XML-Based RPC, Version 1.1. `http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec10`, 2003.

[13] S. Nagano, T. Hasegawa, A. Ohsuga, and S. Honiden. Dynamic Invocation Model of Web Services Using Subsumption Relations. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, 2004.

[14] P. Buhler, C. Starr, W. H. Schroder, and J. M. Vidal. Preparing for Service-Oriented Computing: A Composite Design Pattern for Stubless Web Service Invocation. In *International Conference on Web Engineering*, 2004.

[15] J. Schlimmer et al. Web Services Policy Framework (WS-Policy), joint specification by IBM, BEA Systems, Microsoft, SAP AG, Sonic Software, and VeriSign. `http://www.ibm.com/developerworks/library/specification/ws-polfram/`, 2006.