Daios: Efficient Dynamic Web Service Invocation

Systems based on the service-oriented architecture (SOA) paradigm must be able to bind to arbitrary Web services at runtime. However, current service frameworks are predominantly used through precompiled service-access components, which are invariably hard-wired to a specific service provider. The Dynamic and Asynchronous Invocation of Services framework is a message-based service framework that supports SOA implementation, allowing dynamic invocation of SOAP/WSDL-based and RESTful services. It abstracts from the target service's internals, decoupling clients from the services they use.

Philipp Leitner,
Florian Rosenberg,
and Schahram Dustdar
Vienna University of Technology

Software systems built on top of service-oriented architectures (SOAs)[1] use a triangle of three operations — publish, find, and bind — to decouple roles participating in the system. Publish and find put requirements on the service registry and the interface definition language. To publish services, an expressive and extensible service definition language must be available and supported by the service registry.[2] The bind operation, however, is independent from the service registry and is handled by the service consumer. In a SOA, consumers must be able to connect to any service they discover during the find step. In addition, they must be able to change this binding at any time (specifically, at runtime) if the original target service becomes unavailable or if the find operation discovers services delivering a more appropriate quality of service level.

Currently, application developers generate *stubs* (service access components, which are typically compiled from a formal service description such as the Web Services Description Language [WSDL]) to invoke services. These stubs handle the actual invocation but are specific to a service provider. If the application invokes a similar service from a different provider, it must regenerate the stubs because services from different providers in the real world never look quite the same. Even if the services provide similar functionality, they usually differ

1089-7801/09/$25.00 © 2009 IEEE

Related Work in Web Service Invocation Frameworks

The Apache Web Services Invocation Framework (WSIF; http://ws.apache.org/wsif) was the first Java-based Web service framework to incorporate dynamic service invocation. The WSIF dynamic invocation interface is intuitive to use if the client application knows the signature of the WSDL operation to invoke. This is an unacceptable precondition for loosely coupled service-oriented architectures (SOAs). Client applications shouldn't have to know service internals such as the concrete operation name. In addition, WSIF provides notoriously weak support for complex XML Schema types such as service parameters or return values. An application can use complex types only if they're mapped to an existing Java object beforehand, which is frequently impossible in dynamic invocation scenarios. These problems, together with the fact that the framework hasn't been under active development since 2003 and the relatively bad runtime performance, render WSIF outdated.

The Apache Axis 2 (http://ws.apache.org/axis2) framework incorporates more SOA concepts than WSIF. It supports client-side asynchrony and works more on a document level than the strictly RPC-based WSIF. Although Axis 2 is still grounded on the use of client-side stubs, it also supports dynamic invocations through the `OperationClient` or `ServiceClient` APIs. However, these interfaces expect the client application to create the invocation's entire payload (for example, the SOAP body) itself. In that case, Axis 2 does little more than transfer the invocation to the server. We expect a higher level of abstraction from a Web service framework for constructing SOA clients. Still, the Axis 2 SOAP and Representational State Transfer (REST) stacks are well developed and high performing. We therefore created an Axis 2 service back end as part of our Dynamic and Asynchronous Invocation of Services (Daios) prototype. The Axis 2 back end uses Daios's dynamic invocation abstraction, but the Axis 2 service stack performs the actual invocation.

Similar problems arise with other recently introduced service frameworks, such as Codehaus XFire (http://xfire.codehaus.org) or XFire's successor, Apache CXF (http://cxf.apache.org). Ultimately, all of these client-side frameworks rely on static components to access Web services, with little to no support for truly dynamic invocation scenarios.

The Java API for XML-based Web services is the latest Java-based Web service specification. JAX-WS, described in Java specification request (JSR) 224,[1] is the official follow-up to JAX-RPC.[2] JAX-WS is implemented, for instance, in the Apache CXF project, where it exhibits problems similar to Apache CXF. Although the name change suggests that JAX-WS is less RPC-oriented than its predecessor, the specification still focuses on WSDL-to-operation mappings, ignoring the messaging ideas of SOA and Web services. JSR 224 doesn't explicitly discuss REST, despite its claims to generally handle XML-based Web services in Java.

Shinichi Nagano and his colleagues introduce a different approach to dynamic service invocation.[3] They bind static stubs to generic instead of precise interfaces. Doing so lets them use the same stubs to invoke any service with a similar interface, thereby enabling looser coupling between client and provider. This approach (unlike ours) can achieve static type safety. It has considerable disadvantages, however. The concept is only feasible for Web services defined using a formalized XML interface (few REST-based services have such interfaces), and the practical implementation of more generic interfaces is often a hard problem, requiring a lot of domain knowledge. Creating a generic framework that SOA clients can use in any problem domain is therefore difficult using this approach.

References

1. D. Kohlert and A. Gupta, "Java API for XML-Based Web Services, Version 2," 2007; http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html.
2. JSR-101 Expert Group, "Java API for XML-Based RPC, Version 1.1," 2003; http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec10.
3. S. Nagano et al., "Dynamic Invocation Model of Web Services Using Subsumption Relations," *Proc. IEEE Int'l Conf. Web Services* (ICWS 04), IEEE CS Press, 2004, p. 150.

in technical details (such as operations or the data encoding used). It's therefore difficult to implement a SOA-based application using client stubs without falling back to generating and loading stubs at runtime (for example, using reflection facilities). We consider such a solution to be a workaround, which further demonstrates the need for stubless service invocation in SOA scenarios.

Our message-based Dynamic and Asynchronous Invocation of Services (Daios) framework lets application developers create stubless and dynamic service clients that aren't strongly coupled to a specific service provider. Instead, Daios's dynamic interface offers a high degree of provider transparency that lets applications exchange service providers at runtime.

Dynamic Service Invocation

Dynamic binding isn't easy with current Web service client frameworks such as Apache Axis 2 or the Apache Web Services Invocation Framework (WSIF). These frameworks rely on client-side stubs to invoke services, which are usually autogenerated at design time. (See the "Related Work in Web Service Invocation Frameworks"

sidebar for a more detailed discussion of these and other current frameworks.) However, stubs are invariably hardwired to a specific service provider and can't be changed at runtime. If service providers are hardwired into the service consumers' application code, producers and consumers can't be considered loosely coupled. The use of client stubs doesn't follow the SOA ideas because the developer performs both find and bind. A client application relying on precompiled stubs can't implement a SOA. We therefore conclude that the SOA triangle is broken.[2]

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

We define several requirements for a Web service invocation framework that supports the core SOA ideas.

The first requirement is *stubless service invocation*. Given that generated stubs entail a tight coupling of service provider and service consumer, the invocation framework shouldn't rely on 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.

Second, a Web service invocation framework should be *protocol independent*. Web service standards and protocols are not yet fully settled. Discussion continues about the advantages of the Representational State Transfer (REST)[4] architecture compared to the more common SOAP and WSDL-based[5,6] approaches to Web services. The framework should therefore be able to abstract from the underlying Web service protocol and support at least SOAP- and REST-based services as transparently as possible.

Third, the framework must be *message driven*. Web services are often seen as collections of platform-independent remote methods. The framework must be able to abstract from this RPC style, which usually leads to tighter coupling, and follow a message-driven approach instead. Additionally, the message-driven interface should be as simple as possible to facilitate the creation of complex messages.

Next, the framework should support *asynchronous communication*. In a SOA, services might take a long time to process a single request. The prevalent request-response communication style is unsuitable for such long-running transactions. The framework should therefore support asynchronous (nonblocking) communication.

Fifth, it must provide *acceptable runtime behavior*. The framework shouldn't imply sizable overhead on the Web service invocation. Using the framework shouldn't take significantly longer than using any of the existing Web service frameworks.

Unfortunately, current Web service frameworks don't fully live up to these requirements.

The Daios Solution

Given our requirements for a Web services invocation framework, we designed the Daios framework and implemented a system prototype. Daios is a Web service invocation front end for SOAP/WSDL-based and RESTful services. It supports fully dynamic invocations without any static components such as stubs, service endpoint interfaces, or data transfer objects.

Figure 1 sketches the Daios framework's general architecture. It also shows where the general SOA triangle of publish, find, and bind fits into the framework. The framework consists of three functional components:

- the general Daios classes, which orchestrate the other components;
- the interface parsing component, which preprocesses the service description (for example, WSDL and XML Schema); and
- the service invoker, which uses a SOAP or REST stack to conduct the actual Web service invocations.

Clients communicate with the framework front end using Daios messages (a Daios-specific message representation format). The framework's general structure is an implementation of the composite pattern for stubless Web service invocation (CPWSI).[7] CPWSI separates the framework's interface from the actual invocation back-end 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 mes-

sages. Daios messages are potent enough to encapsulate XML Schema complex types but are still simpler to use than straight XML. Messages are unordered lists of name-value pairs, referred to as *message fields*. Every field has a 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. Users can therefore easily build arbitrary data structures without needing a static type system.

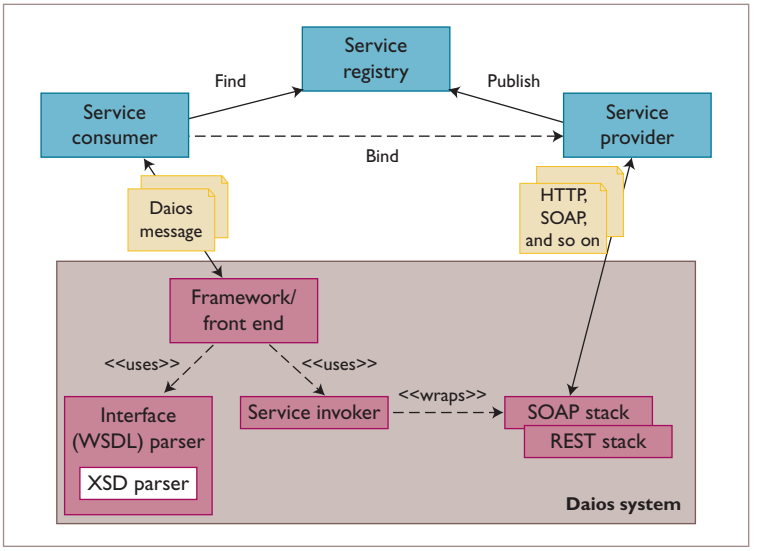

Figure 1. The Dynamic and Asynchronous Invocation of Services (Daios) framework's overall architecture. The framework supports the service-oriented architecture publish, find, and bind paradigm.

Invoking Services with Daios

Using Daios is generally a three-step procedure:

First, clients find a service they want to invoke (service discovery phase). The service discovery problem is mostly a registry issue and is handled outside of Daios.[2]

Next, the service must be bound (preprocessing phase). During this phase, the framework collects all necessary internal service information. For example, for a SOAP/WSDL-based service, the service's WSDL interface is compiled to obtain endpoint, operation, and type information.

The final step is the actual service invocation (dynamic invocation phase). During this phase, Daios converts the user input message into the encoding expected by the service (for instance, a SOAP operation for a WSDL/SOAP-based service, or an HTTP `get` request for REST), and launches the invocation using a SOAP or REST service stack. When the service stack receives the invocation response (if any), it converts it back into an output message and returns it to the client.

Once a service is successfully bound, clients can issue any number of invocations without having to rebind. Service bindings must be renewed only if the service's interface contract changes or the client explicitly releases the binding.

Most of Daios's important processing occurs in the dynamic invocation phase. For a SOAP invocation, the framework analyzes the given input and determines which WSDL input message the provided data best matches. For this, Daios relies on a similarity algorithm. This algorithm calculates a structural distance metric for the WSDL message and the user input — that is, how many parts in a given WSDL message have no corresponding field in the Daios message, where lower values represent a better match. For fields in the user message with no corresponding field in the WSDL message, the similarity is ∞. Daios invokes the operation whose input message has the best (that is, lowest) structural distance metric to the provided data. If two or more input messages are equally similar to the input, the user must specify which operation to use. If no input message is suitable — that is, if all input messages have a similarity metric of ∞ to the input — an error is thrown. Here, the provided input is simply not suitable for the chosen Web service. Otherwise, the framework converts the input into an invocation of the chosen operation, issues the invocation, receives the result from the service, and converts the result back into a message.

The back end used to conduct the actual invocation is replaceable. The Daios research prototype offers two invocation back ends. One uses the Apache Axis 2 stack, the other uses a custom-built (native) SOAP and REST stack. Daios emphasizes client-side asynchrony. All invocations can be issued in a blocking or nonblocking fashion.

This procedure abstracts most of the RPC-like internals of SOAP and WSDL. The client-side application doesn't need to know about WSDL operations, messages, end points, or encoding. Even whether the target service is implemented as a SOAP- or REST-based service is somewhat transparent to the client, although for REST services, clients need to know

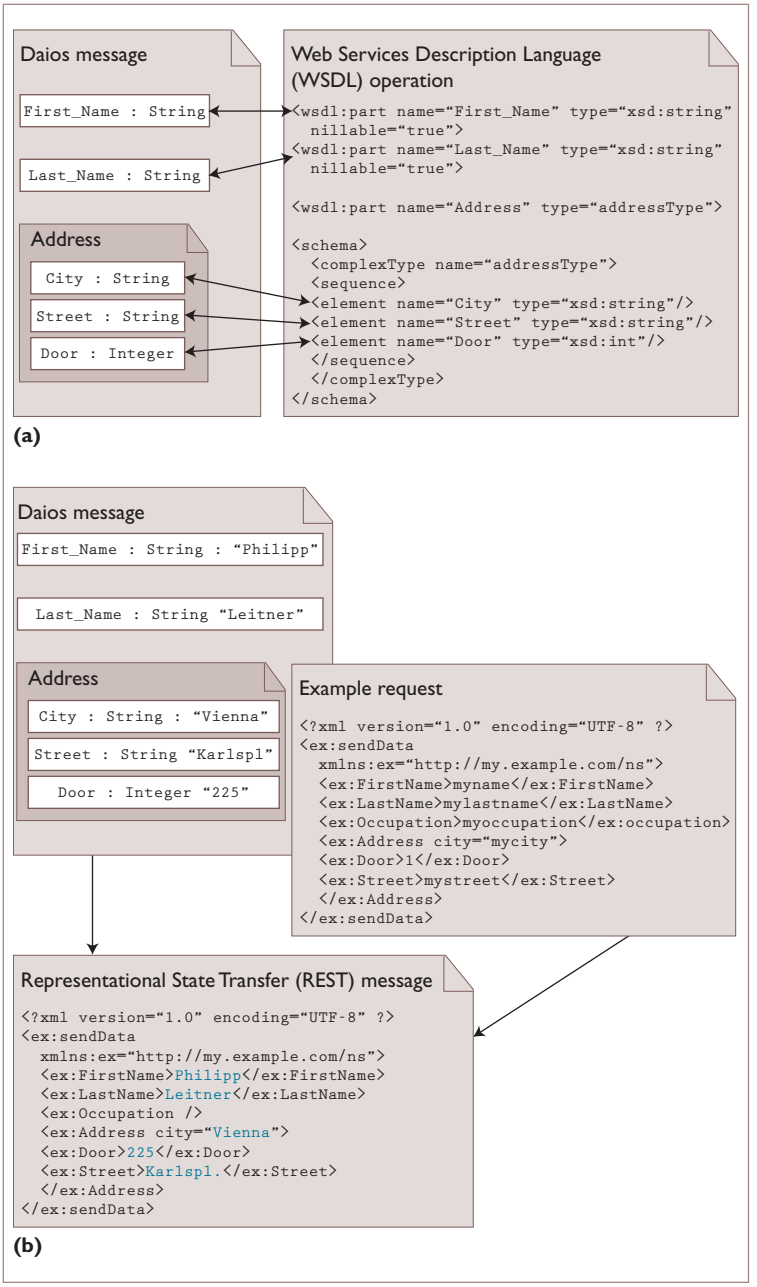

Figure 2. Matching Daios inputs to service interfaces. (a) A Daios message and a WSDL operation with a structural distance of 0. (b) When Daios receives an invocation, it fills the template (example request) with user input to produce a valid REST invocation.

the endpoint address. Daios handles all of these service details, so the client application can be as generic as possible. The service client needs to know only the names and types of mandatory service parameters (for example, WSDL operation parameters).

For REST invocations, the user can specify an example request instead of the WSDL interface, or not give further details on the service interface at all. If the user gives an example request, Daios uses the example as a template, which it fills with the user's actual input at invocation time, and issues an HTTP `post` request with the filled template as payload. If no information about the service interface is given (that is, neither WSDL description nor example request), Daios issues an HTTP `get` request with URL-encoded parameters.

Figure 2 shows the matching of Daios inputs to a WSDL description and an example request. Figure 2a shows a Daios message and a WSDL message in RPC/encoded style with a structural distance of 0 (a perfect match). Removing the `First_Name` field from the Daios message increases the structural distance to 1. Figure 2b details how the framework fills an example template with user input provided as a Daios message in a REST invocation.

Usage Examples

The message-based Daios client API is easy to use. Figure 3 shows the Java code necessary to invoke a SOAP/WSDL-based Web service. The message in this example corresponds to the structure depicted in Figure 2. Although the target service uses nested data structures (the registrations contain address data), Daios doesn't 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 uses this information and converts the user-provided input to a concrete Web service invocation. The example in the figure uses a blocking invocation style, but Daios handles asynchronous communication identically.

The client application doesn't have to specify operation name, endpoint address, or WSDL encoding style used. Daios abstracts this information from the service internals and exposes a uniform interface, allowing loose coupling between client and service.

Figure 4 (p. 78) exemplifies the invocation of a RESTful Web service. In this figure, a client application is accessing the Flickr REST API and retrieving a list of hyperlinks to the most interesting photos.

Daios invokes RESTful and SOAP-based services through the same interface (the code necessary to access the service is practically identical for both Web service types). The main

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

Figure 3. A Daios SOAP invocation. The application constructs both a new service front end to the SOAP-based Web service described by a Web Services Description Language contract and an input message in Daios message format, and issues the invocation using a blocking request response invocation.

difference is that no interface definition language similar to WSDL has yet been established for RESTful services, so the user must specify more service details for REST-based invocations (the endpoint address in the example).

Evaluation

We evaluated our prototype against various Web service frameworks: Apache WSIF, Apache Axis 2, Codehaus XFire, and Apache CXF (see the sidebar). We compared the frameworks in terms of supported functionality, response times, and memory consumption. For brevity, we present only functional aspects and runtime performance data in this article.

Table 1 (p. 79) shows how well the candidate frameworks meet our requirements for a Web service invocation framework. We present the last of these requirements — acceptable runtime behavior — later. Current service frameworks fail to meet these requirements in some important respects. The core problem is that none of them embraces SOA's loosely coupled document-centric approach. Rather, they're based on an RPC processing model, demanding explicit knowledge of service internals such as WSDL encoding styles, operation signatures, and endpoint addresses. Additionally, neither WSIF nor XFire provide a fully expressive dynamic invocation interface. User-defined (complex) types are difficult to use over these interfaces if the application doesn't know the types at compile time. Most interfaces support the REST style of Web services, but they don't support a transparent integration of SOAP and REST. The Daios prototype solves all these problems. It exposes a simple messaging interface with which applications can dynamically invoke arbitrary services without knowing the service's implementation details (including whether the service is implemented as a SOAP- or REST-based service), both synchronously and asynchronously.

Figure 5 (p. 79) addresses the acceptable runtime behavior requirement. The figure compares

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

Figure 4. A Daios Representational State Transfer (REST) invocation. The application creates a Daios service front end to the Flickr photo service's REST API and retrieves a list of the "most interesting" photos.

the response times of the candidate frameworks in simple SOAP-based Web service invocations. Figure 5a shows the results for RPC/encoded invocations, and Figure 5b shows results for document/literal invocations with wrapped parameters. We only evaluated RPC/encoded invocations for Daios and WSIF. Axis 2, XFire, and CXF don't support this particular WSDL encoding style. Apache WSIF is well behind in both test cases; all other candidate frameworks exhibit similar response times.

We also performed extensive tests using different types of invocations (with binary or array payload data), but the general result was similar for all tests. Additionally, we've gathered similar results for REST-based invocations. We therefore conclude that using Daios doesn't imply a relevant performance penalty over Apache Axis 2, Apache CXF, or Codehaus XFire, and that our prototype is significantly faster than Apache WSIF.

Increasingly, Web service implementations use policies to describe the service's nonfunctional attributes, such as security policies, transactional behavior, and reliable messaging. Often, these implementations use the Web Services Policy framework.[8] We plan to add WS-Policy support to our framework to support policy-enforced interactions. Furthermore, we'll extend our evaluation of the Daios framework to a more extensive real-life scenario to get a more accurate picture of the implementation's runtime performance and usability in real business applications.

We recently released the first version of our Daios prototype as an open source project using Google Code and are currently working on a .NET port.

Acknowledgments

The European Community's Seventh Framework Program (FP7/2007-2013) helped fund the research leading to the results reported here under grant agreement 215483 (S-Cube).

Table 1. Functional comparison of current Web service invocation frameworks (WSIFs).

Requirement	Daios	Apache WSIF	Apache Axis 2	Codehaus XFire	Apache CXF
Stubless service invocation					
Simple types	Yes	Yes	Yes	Yes	Yes
Arrays of simple types	Yes	Yes	Yes	Yes	Yes
Complex types	Yes	No	Yes	No	Yes
Arrays of complex types	Yes	No	Yes	No	Yes
Protocol independence					
Transparent protocol integration	Yes	No	No	No	No
SOAP over HTTP support	Yes	Yes	Yes	Yes	Yes
Representational State Transfer support	Yes	No	Yes	No	Yes
Message-driven approach					
Document-centric interface	Yes	No	No	No	No
Transparent handling of service internals	Yes	No	No	No	No
Support for asynchronous communication					
Synchronous invocations	Yes	Yes	Yes	Yes	Yes
Asynchronous invocations	Yes	No	Yes	No	Yes
Simple API					
Simple to use dynamic interface	Yes	Yes	No	Yes	Yes

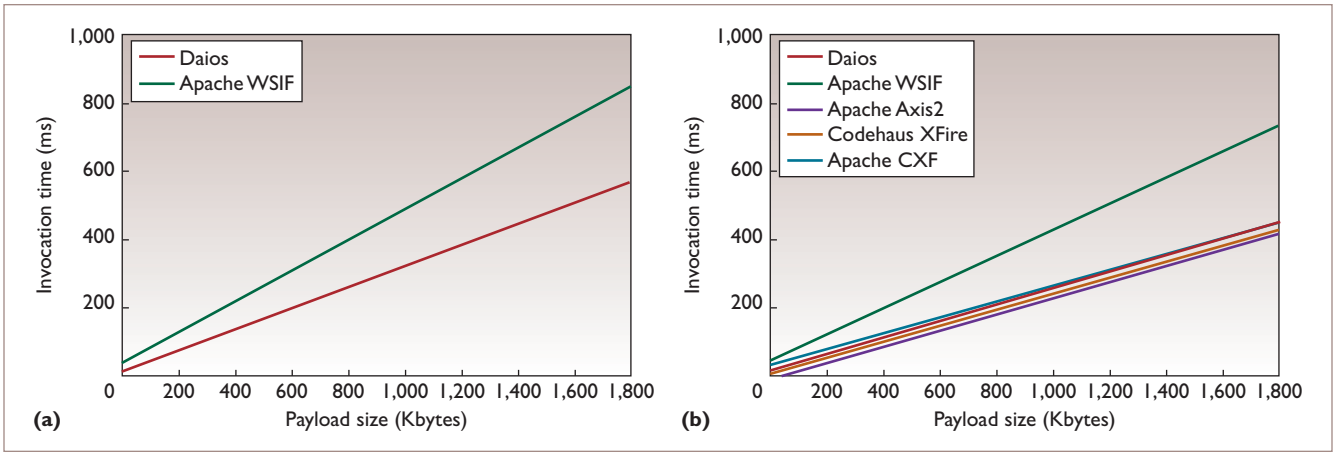

Figure 5. Comparison of invocation response times. (a) RPC/encoded invocations and (b) document/literal invocations with wrapped parameters. Only Daios and the Web Services Invocation Framework (WSIF) support RPC/encoded.

References

1. M.P. Papazoglou et al., "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, 2007, pp. 38–45.
2. A. Michlmayr et al., "Towards Recovering the Broken SOA Triangle: A Software Engineering Perspective," *Proc. 2nd Int'l Workshop on Service Oriented Software Eng.* (IW-SOSE 07), ACM Press, 2007, pp. 22–28.
3. W. Vogels, "Web Services Are Not Distributed Objects," *IEEE Internet Computing*, vol. 7, no. 6, 2003, pp. 59–66.
4. R.T. Fielding, *Architectural Styles and the Design of Network-Based Software Architectures*, doctoral dissertation, Information and Computer Science Dept., Univ. of California, Irvine, 2000.
5. *SOAP Version 1.2 Part0: Primer*, World Wide Web Consortium (W3C) recommendation, 2003; www.w3.org/TR/soap12-part0.
6. *Web Services Description Language (WSDL) Version 2.0 Part0: Primer*, World Wide Web Consortium (W3C) candidate recommendation, 27 Mar. 2006; www.w3.org/TR/2006/CR-wsdl20-primer-20060327.

7. P. Buhler et al., "Preparing for Service-Oriented Computing: A Composite Design Pattern for Stubless Web Service Invocation," *Proc. Int'l Conf. Web Eng.*, LNCS 3140, Springer, 2004, p. 763.
8. J. Schlimmer et al., "Web Services Policy Framework (WS-Policy)," joint specification by IBM, BEA Systems, Microsoft, SAP AG, Sonic Software, and VeriSign, 2006; www.ibm.com/developerworks/library/specification/ws-polfram.

Philipp Leitner is a PhD student in the Distributed System Group at the Vienna University of Technology. His research interests include general issues of distributed computing, especially service-oriented computing, service-oriented architectures, Web services, peer-to-peer computing, and network management. Leitner has a master's degree in business informatics from the Vienna University of Technology. Contact him at leitner@infosys.tuwien.ac.at.

Florian Rosenberg is a PhD candidate in the Distributed System Group at the Technical University Vienna. His research interests include software composition, service-oriented architectures, and software engineering. Rosenberg has a master's degree in software engineering from the Upper Austria University of Applied Sciences. Contact him at florian@infosys.tuwien.ac.at.

Schahram Dustdar is a full professor of computer science, director of the Vienna Internet Technologies Advanced Research Lab, head of the Distributed Systems Group of the Information Systems Institute at the Vienna University of Technology, and honorary professor of information systems in the Department of Computer Science at the University of Groning, the Netherlands. His research interests include service-oriented architectures and computing, mobile and ubiquitous computing, complex and adaptive systems, and context-aware computing. Dustdar has a PhD in business informatics from the University of Linz, Austria. He's a member of the IEEE Computer Society and the ACM. Contact him at dustdar@infosys.tuwien.ac.at.