Multi-protocol Web Services for enterprises and the Grid

N. Mukhi, R. Khalaf,
IBM TJ Watson Research, Hawthorne, New York, USA

P. Fremantle
IBM Hursley Park, Winchester, Hampshire, UK

Abstract

The Web Services paradigm is a simple model for building web-wide distributed systems using XML and HTTP. However, as the use of this technology expands to include Grid and Enterprise computing, the requirement for higher qualities of service becomes important. To address these requirements, we propose a pluggable framework for using Web services components that allows both static and dynamic invocation of services over multiple protocols. These protocols can include existing reliable technologies such as CORBA/IIOP or new emerging protocols such as BEEP. By moving Web services beyond SOAP, the approach embodied by the Web Services Invocation Framework enables a Web service to be exposed and used over any of a number of protocols whose choice can be made on the basis of the available infrastructure and the required quality of service.
A complementary piece of pluggable architecture, the Web Services Gateway, is presented to handle interception of messages coming in to a service. This allows services to be offered over protocols that they were not originally designed for, as well as performing mediation functions as routing, mapping to different protocols, and filtering. Making the gateway itself uses pluggable providers for multiple bindings enables a Web service to be invoked over multiple protocols, while still being able to benefit from the mediation services provided. Providing a Web service with both of the above frameworks enables it to both invoke and be invoked by other artefacts distributed over the network while taking full advantage of the heterogeneity that makes Web services unique. The evolution of this concept is a separation between application logic, which deals in "services" as complete units of available function, and the infrastructure, which may select the best available service over the most appropriate protocol.

1. INTRODUCTION

The Web Services framework and architecture is the subject of a high level of research and development, particularly under the W3C and OASIS bodies. A number of organisations are building a set of standards that will allow users to build connections between companies and heterogeneous systems to perform computations and exchange data - all the while using open protocols and standards such as SOAP[1] and HTTP.

At the same time the scientific community is attempting to build the next generation of high-performance distributed computing systems using a concept known as Grid computing[2, 3]. Grid computing envisages using distributed resources to attack very large computing problems, such as analysis of terabytes of data generated by particle accelerators, for example the Collision Detector at Fermilab [4].

Both these challenges require very fast efficient access to both new and existing computing resources. In some cases they also require high quality of service: such as reliability, guaranteed delivery, transactionals and security. While standards for these are under development, many existing technologies, such as CORBA[5], already have these capabilities, and re-use of existing systems is an important goal. In this paper we describe an approach to service oriented architecture that allows enterprises and Grids to use existing high-performance reliable computing technologies and also to interoperate with new Web services standards. We describe an infrastructure that allows the optimal protocol or invocation method to be chosen at runtime, and that allows existing services to be exposed over multiple protocols.

2. RELATED WORK

The ideal distributed computing world relies on the heterogeneity of its constituent subsystems. The differing needs of developers and applications, the fast rate of change of the technologies available, and the connectivity of the internet make the varying landscape of the IT infrastructure not only necessary but also very beneficial. It is also important to be able to utilise existing proven running systems, for reasons of efficiency, reliability and skills. A large number of platforms exist for distributed computing; however, most rely on deploying the same infrastructure on all parties involved. The emerging field of Web services provides an approach that leverages the heterogeneity of IT systems by using open standards.

Distributed programming languages such as Java RMI[6] and Concert/C [7, 8], were created to handle interoperation of remote applications in the same language. The interface definition language in Concert/C presents a separation of interface and "endpoint modifier" that is similar to that exhibited by Web services, but is internal, machine-generated, and has no human readable syntax. Concert/C presents an extensible multi-protocol capability for doing RPC calls over both its own RPC protocol and other "open" RPC protocols. However, the solutions presented by these systems require that the objects/applications they connect respectively use Java and ANSI C (or Concert/C which is a superset of ANSI C).

On the other hand, systems such as CORBA[5] and Mockingbird[9] enable distributed, heterogeneous, application-to-application integration . Nakajima [10] proposes an extension to CORBA interfaces that exposes multiple available protocols, such as IIOP and IIOP over SSL, thereby allowing applications to select the most appropriate protocol from those present. As opposed to Web services, these systems require that the programmer adapt the applications or components that are to be included in the distributed system so they may be used by these frameworks. Web services shift this bridging responsibility into the middleware, allowing the user to use the access method and data format that most suit the situation at hand. Web services do not aim to replace such existing systems, but instead to allow them to interoperate as a result of its looser coupling. Vinoski[11] therefore describes Web services as "middleware for middleware."

By using the Web services standards and tools, Grid computing can capitalize on the advantages presented by Web services. In [3], Foster et al define the Open Grid Services Architecture (OGSA) that proposes to align the two architectures in this manner. They stress that the de facto Web services messaging protocol SOAP must not be the only means to communicate with available services, a view which is put into practice in the technologies presented in this paper.

The Web Services Invocation Framework (WSIF)[12,13,14] and Web Services Gateway (WSGW)[15,16] presented here take advantage of the Web services specifications to enable communication over a set of available data formats and access protocols, including the native implementation of the component used.

The aim of WSIF and the WSGW is to meet the requirements of complex distributed systems. An example is a process manager. In order to execute the process, the manager must execute a number of remote and local services. If the local services are executed using SOAP/HTTP, then each execution spawns a new thread and causes XML marshalling and de-marshalling. If however, a fast in-process execution binding can be described and invoked, the process manager will be much more efficient. We describe the use of WSIF and the WSGW in the Enterprise in [17]. Similarly, the OGSA architecture and OGSi infrastructure require fast execution of local services while using an open architecture.

3. THE WEB SERVICES COMPONENT MODEL

The Web services framework aims to accommodate and leverage arbitrary applications, protocols and platforms which provide services, while placing the burden of integrating such heterogeneous artefacts away from the developer. The definition of a Web services component, provided by the Web Services Description Language (WSDL)[18], embodies this requirement by separating the abstract component description from its mappings to available deployed implementations. The formal description of a service's functionality is expressed in a normaliz platform independent fashion by defining one or more portTypes, which specify the operations supported and the input and/or output message structures. Each message is composed of one or more parts whose structure is forr described using XML Schema[19] as an abstract type system. The interface thus created may then be used by th parties aggregating components or by client code invoking operations, regardless of protocol or service implementation.

The concrete aspect of a WSDL definition, which is called a binding, includes data and protocol mappings from th above abstract description to available network protocols and data encoding formats. The extensibility of WSDL bindings allows WSDL to support an open-ended set of data formats and network protocols. Finally, ports provide locations of physical endpoints that implement the given abstract functionality and are available with a particular binding. A simplified graphical view of the WSDL structure is given in figure 1.

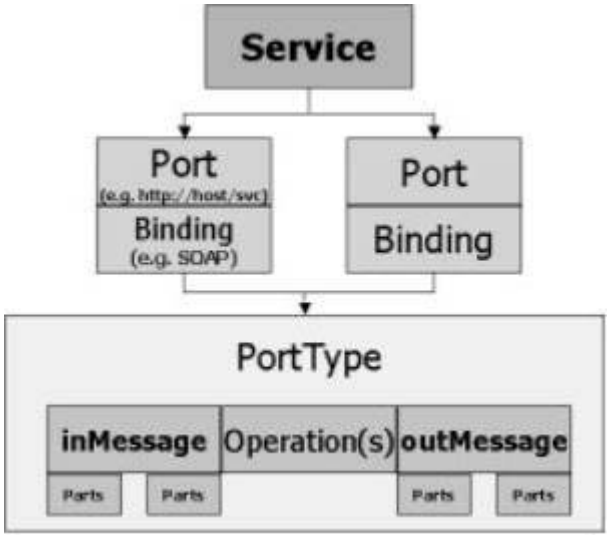

FIGURE 1: WSDL 1.1 a simple graphical representation

This separation enables dynamic selection of the access mechanism to a Web service, as a single interface/portType may be implemented with many bindings. In turn, this enables invocations to be optimised based on such factors as the native environment of the invoker (implementation, platform, location), security considerations, and network properties. WSDL 1.2 defines "standard bindings" that map abstract service descriptions to SOAP, HTTP GET/POST and MIME, of which SOAP is the most popular and often thought of as the only protocol for Web Services. Our approach is to take advantage of WSDL extensibility and make it the centrepiece of a unified meta-component model. To achieve this, we have to define mappings so that WSDL encompasses more than just SOAP services.

A new mapping needs to define how the following tasks are handled:

- Marshalling of data between the common type representation (usually schema-typed XML) and the deployment representation: This may involve defining a protocol-specific encoding for XML (as in SOAP) or mapping schema-typed structured data to some local representation.

- Communicating with the service endpoint: The data needs to be carried to a well-defined service endpoint - in the case of SOAP, protocols like HTTP and SMTP are used to communicate with the endpoint which is exposed as a URI.

We have used this process to define new mapping so that Java programs, Enterprise JavaBeans (EJBs), programs using the Java Messaging Service (JMS), CICS transactions, and applications that use the Java Connector Architecture (JCA) can all be described using WSDL. This results in a large number of software systems being viewed as WSDL-described services, hiding language and protocol related issues. Applications that interact with the software can then concentrate on the abstract interface and program against that. To make this a reality, we need a WSDL-driven programming model: the Web Services Invocation Framework (WSIF), described below, provides this.

4. THE WEB SERVICES INVOCATION FRAMEWORK

The Web Service Invocation Framework (WSIF) is a simple Java API for invoking Web services, no matter how or where the services are provided:

- It has an API that provides binding independent access to any Web service.
- It allows stubless (completely dynamic) invocation of Web services. This allows invocation of services based on examination of the meta-data at runtime - for example in a test client.
- An updated implementation of a binding can be plugged into WSIF at runtime.
- A new binding can be plugged in at runtime, using the provider concept.
- It allows the choice of a binding to be deferred until runtime.
- It is closely based on WSDL, so it can invoke any service that can be described in WSDL.

WSIF is an open-source project under the auspices of the Apache Software Foundation, and part of the Axis web services project [14].

4.1 Pluggable Invocation Framework

WSIF consists of a Java™ Application Programming Interface (API), which closely mirrors the constructs of WSDL. The API allows the user of a service to select the service (through choosing a description), and then invoke operations on that service. There are two approaches to invoking operations. The main approach is the “Stub” model. In this model, the programmer generates an “business” interface, known as the Service Definition Interface (SDI). This interface is generated from the WSDL description and mirrors the Port Type in Java. For example, if there is an operation named “getQuote” there will be a method on the interface named “getQuote(…)”.

WSIF then dynamically generates an object which implements this interface, allowing the user to treat the service as a simple remote stub object.

The second approach is called “stubless”, or the Dynamic Invocation Interface (DII). In this approach, the user of a service uses a set of factories to create objects that mirror the WSDL concepts. This API is designed for invoking services based on dynamically inspecting the WSDL document, and then generating service requests based on the metadata. For example, a test harness could inspect the WSDL, prompt the user for inputs based on the schema and input types of the operation, and then use this interface to dynamically invoke the service.

This approach is used in the Web Services Gateway that we describe below, and has also been used to build business process execution environments that dynamically invoke services based on a process description document. In this model, typically, the process engine takes data from the output of one or more operations and then manipulates it before using it as the input to the next operation. This API is ideal for that task.

Whichever aspect of the API is used, the programmer is not concerned with the underlying protocol or format that is used to execute the service request – this is shown in Figure 2. For example, the request may be using IIOP to invoke a remote EJB, or SOAP to invoke a remote Web Service. The programmer has the option to choose the named port from the WSDL that corresponds to a particular binding or access protocol, or may defer this choice to the WSIF infrastructure. In the current production implementation the first protocol that is implemented in the client code base is chosen, but more complex decision and selection approaches have been implemented in research code.

4.2 Handling different protocols

In order to handle different protocols, WSIF implements a extensibility framework. This allows new modules to be added to a client runtime to support additional protocols. These modules are known as WSIF Providers. Each provider handles all possible service requests for a given protocol, and so the deployment of one provider allows multiple services to be accessed over a given protocol. This compares well in terms of manageability with deploying multiple stubs.

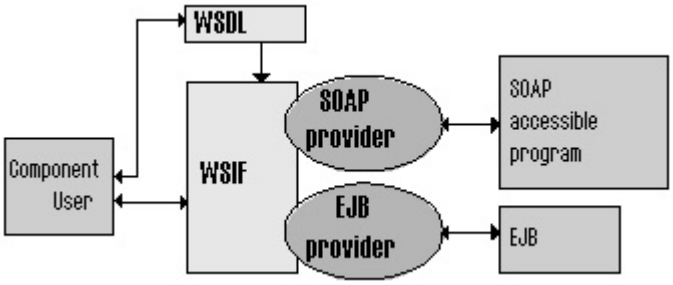

FIGURE 2: the Web Services Invocation Framework

4.3 Pluggable providers

The provider is responsible for translating the Java API call into the correct invocation using the protocols and formats supported by that provider. For example, the provider may make an invocation against a remote EJB using IIOP, or a SOAP call by creating an XML stream over HTTP. In order to do this, the provider needs metadata which describes the mapping from the abstract XML Schema based description of the interface (the port type) and the real invocation system. This is done using the extensibility capabilities of WSDL. The provider must be registered with the WSIF runtime. Each provider supports a given protocol, and this is represented by the namespace of the binding in the WSDL. A WSIF Provider is accompanied by a parser plug-in that understands the grammar of the extensibility elements in WSDL. WSIF uses the WSDL4J open source parser[20], which supports extensions.

As WSIF is implemented in Java, we chose to use the JAR file system. One aspect of this system is designed to support pluggable extensions such as the WSIF provider system. This system allows the WSIF runtime to load all the available providers. The runtime then queries each module to identify which namespaces it supports. It uses this data to then utilise the right module when presented with a specific WSDL document.

WSIF currently has providers for SOAP/HTTP, SOAP/JMS, Enterprise JavaBeans using RMI-IIOP, native JMS messaging, and legacy systems using the J2EE Connector Architecture to access systems including CICS and IMS.

4.4 In-memory Representation of Data

Messages created when accessing a Web service are often created using the native language of the invoker. However, the client may not always have access to data structures that properly represent the message definition in which case the middleware could generate these structures. In the case of compiled languages such as Java, this involves expensive code generation and class loading. A mapping from XML Schema to Java is being formally specified by the Java Architecture for XML Binding (JAXB)[21]. JAXB lays down a set mapping for most simple types, while complex types are mapped to classes that must be generated using a code-generation tool. There is a need for a lightweight, generic representation for instances of XML Schema typed data that does not require code-generation but can represent any valid Schema-typed data.

In WSDL, the message is a collection of XML Schema defined parts, so an instance representing a part may then faithfully represent the data structure as a tree with values corresponding to simple (non-structured) types at the leaves. With a straight-forward creation API and values at the leaves typed in the native language, the invoker may easily create the arguments of operations without requiring extra class generation or the creation of XML trees whose values require the conversion of everything into a string (for example, the DOM API[22]).

The Java Record Object Model (JROM)[23] presents such an abstract tree structure to represent schema types in memory. JROM values are therefore either typed simple values for first level data, or complex values that contain two collections of named JROM values, one for elements and one for attributes. The simple JROM values contain a Java value that is able to represent the given Schema type. For example, a JROMFloatValue contains a Java float and corresponds to the Schema float simple type. The mappings between Schema and Java used for the simple types are nearly the same as those defined by JAXB. However, all complex types are

mapped to the tree structured JROMComplexValue. An example is given in Figure 3.

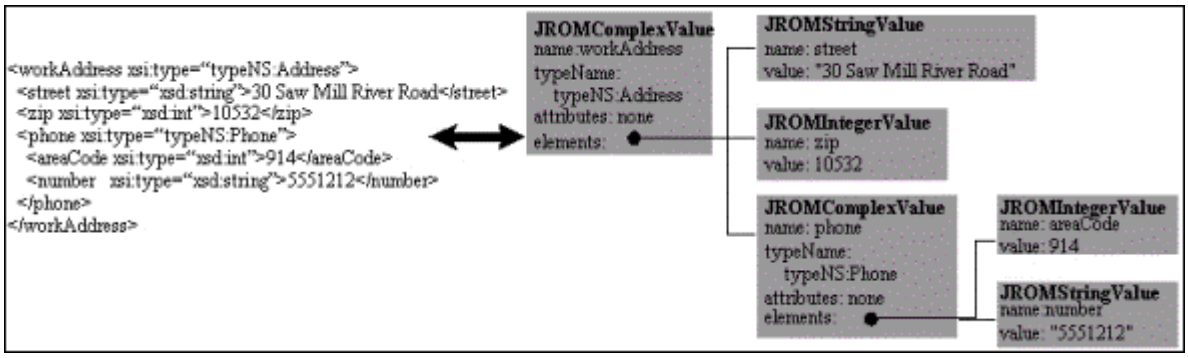

FIGURE 3: JROM example

4.5 WSIF and JROM

WSIF includes a command-line tool called the DynamicInvoker, whose first iteration only supported the use of JavaBean components to represent complex types. Because of this the command-line tool was restricted to only invoking services that use simple types, unless the developer modified the code to use generated JavaBeans. By using JROM with WSIF, dynamic invocation is allowed even if there are no Java types that match the schema complex types in the classpath. This is a very powerful approach, especially in building dynamic Web services systems such as gateways, flow engines, or test clients. WSIF supports both JROM (currently only in the the ApacheSOAP provider) as well as code-generated Java types (all providers).

5. WEB SERVICES GATEWAY

A WSDL description advertises a Web service's capabilities and describes protocol bindings supported by the web service. It is straightforward to use such a description to access the functionality offered by a web service through frameworks such as WSIF or protocol-specific client libraries. However, there are situations in which the service requestor will not have the necessary runtime to communicate with the service. For example, the service may be available within an enterprise through a proprietary protocol and a situation arises when we need to expose this service to a business partner. Or, a GRID service may be available through a high performance protocol but needs to be accessed through a lighter weight one due to lack of the necessary resources. In such situations, we need to access a service façade that appears to offer the same functionality over multiple convenient protocols, and performs the necessary protocol conversions to access the actual service through one of its advertised bindings. In this section, we describe the Web Services Gateway that solves this problem and also has some other useful capabilities:

- It can make a Web service available through bindings for which it was not originally designed
- It allows for information sent between a service requestor and the Web service to be filtered and mediated.
- It can act as a single point through which all services offered within a domain (an enterprise, university laboratory, etc.) can be accessed, thus allowing for convenient access control, logging, etc.

5.1 Internal and External views

With the Gateway acting as a mediator, there are two views of the same service: an external view, visible to service requestors that access the service through the Gateway, and an internal view, visible only to the Gateway itself. Web Services deployed in such an environment make their WSDL descriptions available to the Gateway, which then generates an external WSDL containing the same abstract description, but with different bindings and ports. The bindings offered by this external WSDL will depend on the protocols that the Gateway supports, and which bindings are chosen by the administrator. The internal WSDL specifics (other than the abstract description) are thus hidden from the eventual service requestor. This is depicted in figure 4.

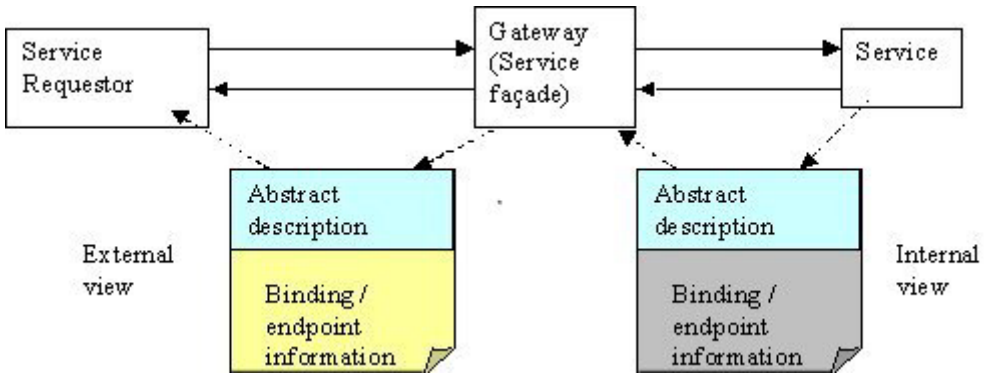

FIGURE 4: Internal and External views of a service

5.2 Channel framework

A channel is the software component that supports bindings and protocols in the Gateway, and manages communication between a service requestor and the Gateway. Incoming service requests arrive at the channel through a protocol-specific mechanism. The channel converts them into a normalized format and hands them off to the Gateway Manager. Response messages also leave via a channel, which sends the normalized message to some party interacting with the service using a specific protocol.

The Gateway is associated with a channel manager that keeps track of all the channels in the system. The administrator associates one or more of these channels with each service deployed to the Gateway. The external WSDL for the service advertises bindings and endpoints that can be handled by its associated channels.

Channels can be added to the Gateway at any time, thus allowing the future addition of support for new protocols.

5.3 Mediation and filtering

When a normalized message is delivered to the Gateway, it is mapped to a deployed service based on either a static or dynamic configuration. In the static case, there is just one target service defined for each Gateway-exposed service. In the dynamic case, multiple target services that share the same interface can be defined and a user-written component chooses which service to invoke. The Gateway then acts as a requestor for the actual service, invoking the required operation on the service's interface. Once the invocation is complete, any output message is returned to the original service requestor via the channel mechanism.

The Gateway allows message interceptors (known as filters) to be registered with each service. These filters are of two types: those that can intercept incoming messages (pre-interceptors, since they are executed prior to the invocation of the service itself) and those that intercept outgoing messages (post-interceptors, since they are executed after the service). Filters can perform tasks such as logging, authorisation, etc.

5.4 Use of WSIF and JROM

The normalized representation for all data within the Gateway is a message consisting of distinct parts that are JROM values. This is a convenient format since data conversions between JROM and message representations such as Java and XML (which most services consume) are straightforward and efficient.

As we pointed out above, once the Gateway has determined what the target service for a message is, it acts as a service requestor. It invokes the actual service using WSIF. It uses the WSIF Dynamic Invocation Interface (DII) described above, so the invocation can be done without the need for generating any code. Code generation is unnecessary in part due to the WSIF dynamic invocation model and in part due to the benefits of using JROM for representing structured data.

The architecture allows the Gateway to invoke services irrespective of their location or binding - including

locally, remotely, and even mediated through further gateways. In fact, the internal view that one Gateway sees could actually be the external view that is generated by some other Gateway (that is, an actual implementation behind the service façade may in fact be another service façade).

5.5 Applying the Framework: Business Process Execution

In this section, we present a use case in which the combination of WSIF, JROM, and the gateway is used to enable compositions of Web services to interact with the outside world over multiple protocols.

The Business Process Execution Language for Web Services (BPEL4WS, or BPEL for short) [24] is an XML specification for composing Web services. Compositions are created by defining and wiring together different specialized activities that can, for example, perform Web services invocations, manipulate data, throw faults, or terminate the process. Primitive activities are combined to form complex processes through the use of structured activities that specify execution order, for example in sequence, in parallel, or depending on certain conditions.

BPEL supports recursive composition by allowing the composition itself to be exposed as a Web service with a WSDL description. The service thus created is not only invoked by other parties, but usually invokes other Web services itself. In order to support the required interaction models, our implementation of BPEL4WS uses WSIF and a module based on the Web services gateway on top of the core engine that executes the composition semantics. In all three parts of the system, JROM is used as the internal data representation.

The implementation, BPWS4J[25], contains a Process Management Module(PMM) that handles all invocations coming to the composition's WSDL operations. The PMM's design is based on the Web Services Gateway, creating a single point of entry to all instances of the running compositions where messages can be filtered and routed to the appropriate instances.

Once a business process instance gets a message, it may manipulate it directly, send it out to make another invocation, or use its parts to evaluate conditions on the links connecting activities. Data in BPEL consists of WSDL messages, which in the BPWS4J engine become WSIF messages whose parts are JROMValues. By treating everything as a JROMValue, a business process instance is able to able to locally manipulate data from a large number of distributed Web services without requiring class generation for each of the data types involved. With both the PMM and the Invocation Module (below) using JROM for their own benefits as described earlier, minimal data conversion is required between the core engine and the its interaction infrastructure.

Another module, the Invocation Module, handles all invocations made by the composition to the Web services it uses. The Invocation Module directly uses WSIF, thereby it automatically supports multi-protocol and dynamic binding to its constituent components.

6. FUTURE WORK

Both BPEL4WS and the OGSA architecture utilise new features of WSDL and the Web Services stack. In particular, they require support for Service References and stateful services, which may have instances. For example, in a simple travel-booking scenario, the customer may pass a reference to a confirmation service that is instantiated on the customers system. Then, as the travel agent makes bookings with airlines, car rentals and hotels, it passes on the reference. Each of the parties then confirms directly with the customer, allowing the customer to record directly the signed messages from the third parties and thus prove the bookings. Support for this is seen as important for WSIF and the Gateway.

We also envisage a broad range of providers. For example, we have done work on developing providers that can invoke Microsoft COM and .NET components directly or using DCOM.

Finally, there is always work to do adding support for the latest standards, such as WSDL 1.2, SOAP 1.2, etc. As WSIF is an open-source project, any interested parties are welcome to contribute!

7. CONCLUSION

We have described an architecture that builds upon the Web Services framework in a standard and extensible way. We believe that the services oriented vision has a lot to offer small companies, enterprises, and academic and scientific environments, and the creation of framework that supports multiple protocols allows us to create services which are modular, accessible, well-described, implementation-independent and interoperable, yet still efficient, manageable and that work with existing infrastructures.

REFERENCES

[1] Simple Object Access Protocol (SOAP) 1.1, W3C Note, May 2000.
[2] Ian Foster, Carl Kesselman, and Steven Tuecke, “The anatomy of the Grid: Enabling scalable virtual organizations,” International Journal of Supercomputing Applications, 2001, 15(3).
[3] Foster, I., Kesselman, C., Nick J.M., and Tuecke, S., “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration,” 2002, draft, www.globus.org/research/papers/ogsa.pdf
[4] Collision Detector at Fermilab, http://www-cdf.fnal.gov/cdf.html
[5] Common Object Request Broker: Architecture and Specification, Revision 2.2. Object Management Group Document 96.03.04, 1998.
[6] Sun Microsystems, ”Java Remote Method Invocation: Distributed Computing for Java, ” http://java.sun.com/marketing/collateral/javarmi.html
[7] Joshua Auerbach, Ajei S. Gopal, Mark T. Kennedy, and James Russell, “Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime,” In the 14th International Conference on Distributed Computing Systems, June 1994.
[8] Joshua Auerbach and and James R. Russell, “The Concert Signature Representation: IDL as an intermediate language,” Proc. of the 1994 ACM SIGPLAN Workshop on Interface Definition Languages, January 1994.
[9] Joshua Auerbach and Mark C. Chu-Carroll, “The Mockingbird System: A Compiler-based approach to maximally interoperable distributed programming,” Research Report RC 20718, IBM T. J. Watson Research Center, February 1997.
[10] T. Nakajima, “Practical Explicit Binding Interface for Supporting Multiple Transport Protocols in a CORBA System”, Proc. of IEEE International Conference on Network Protocols (ICNP 2000), November 2000.
[11] Steve Vinoski, “Where is Middleware,” IEEE Internet Computing, March/April 2002, 6(2).
[12] Michael Beisiegel, Matthew J. Duftler, Paul Fremantle, Nirmal Mukhi, Piotr Przybylski, Aleksander A. Slominski, and Sanjiva Weerawarana, “Web Services Invocation Framework,” Oct. 2001, released on www.alphaworks.ibm.com/tech/wsif
[13] Duftler, M.J., Mukhi, N., Slominski, A., and Weerawarana, S., “Web Services Invocation Framework (WSIF),’ OOPSLA Workshop on Object Oriented Web Services, October 2001.
[14] Web Services Invocation Framework, Apache Software Foundation, June 2002, http://cvs.apache.org/viewcvs.cgi/xml-axis-wsif/
[15] Paul Fremantle, Nirmal Mukhi, William Nagy, and Sanjiva Weerawarana, “Web Services Gateway,” released on www.alphaworks.ibm.com/tech/wsgw Dec. 2001.
[16] IBM Web Services Gateway, WebSphere Developer Domain, http://www7b.boulder.ibm.com/wsdd/downloads/wsgw/wsgw.html
[17] Paul Fremantle, Sanjiva Weerawarana, and Rania Khalaf, “Enterprise Services,” Communications of the ACM (CACM), Volume 45 , Issue 10 (October 2002), http://doi.acm.org/10.1145/570907.570935
[18] Christensen, E., Curbera, F., Meredith, G. and Weerawarana., S. Web Services Description Language (WSDL) 1.1. W3C, Note 15, 2001, www.w3.org/TR/wsdl
[19] Fallside, D.C. XML Schema Part 0: Primer. W3C, Recommendation, 2001, http://www.w3.org/TR/xmlschema-0/
[20] Matthew Duftler, Paul Fremantle, WSDL4J, July 2001, http://www-124.ibm.com/developerworks/projects/wsdl4j
[21] Joseph Fialli and Sekhar Vajjhala, “The Java Architecture for XML Binding (JAXB), public draft, V0.7, ” Septmeber 12, 2002, http://java.sun.com/xml/downloads/jaxb.html
[22] Document Object Model (DOM), http://www.w3.org/DOM/DOMTR
[23] Rania Khalaf, Sanjiva Weerawarana, Nirmal Mukhi, and Matthew Duftler, “Java Record Object Model (JROM)” www.alphaworks.ibm.com/tech/jrom
[24] Curbera, F., Goland, Y., Klein, J., Leyman, F., Roller, D., Thatte, S., and Weerawarana, S., “Business Process Execution Language for Web Services(BPEL4WS) 1.0,” August 2002,

http://www.ibm.com/developerworks/library/ws-bpel.
[25] Francisco Curbera, Matthew J. Duftler, Rania Khalaf, Nirmal Mukhi, William A. Nagy, and Weerawarana S., “Business Process Execution Language for Web Services Java Runtime(BPWS4J),” August 2002, released on www.alphaworks.ibm.com/tech/bpws4j