Architectural Foundations of WSRF.NET

Marty Humphrey, University of Virginia, USA

Glenn Wasson, University of Virginia, USA

ABSTRACT

State management has always been an underlying issue for large-scale distributed systems, but only recently has it been brought to the forefront of Grid computing with the introduction of the Web Services Resource Framework (WSRF) and its companion, WS-Notification. WSRF advocates standardized approaches for client exposure to and potential manipulation of stateful services for Grid computing; however, these arguments and their long-term implications have been difficult to assess without a concrete implementation of the WSRF specifications. This paper describes the architectural foundations of WSRF.NET, which is an implementation of the full set of specifications for WSRF and WS-Notification on the Microsoft .NET Framework. To our knowledge, the observations and lessons learned from the design and implementation of WSRF.NET provide the first evaluation of the WSRF approach. A concrete example of the design, implementation, and deployment of a WSRF-compliant service and its accompanying WSRF-compliant client are used to guide the discussion. While the potential of WSRF and WS-Notification remains strong, initial observations are that there are many challenges that remain to be solved, most notably the implied programming model derived from the specifications, particularly the complexity of service side and client code and the complexity of WS-Notification.

Keywords: Grid computing; .NET framework; Web Services notification; Web Services Resource Framework

INTRODUCTION

There is probably no single best approach with regard to state management in distributed systems. The fundamental issue is not whether state exists in the services that comprise the distributed system (most people believe that the description of a non-trivial distributed system must include some representation of state) but rather what a client can assume about the state of the particular service with which the client wants to interact. For years, architects and system designers have compared the relative virtues of *stateful* services and *stateless* services. Simplistically, on one hand, it is argued that stateless services scale better and are more fault-tolerant, while, on the other hand, stateful services support terser messages that, hence, are more efficient, can be more intuitive to design, and, indeed, can scale well due to recent advances in software support for services. The general theme regards the notion of

conversation — specifically, what can and should a client say in its next request to the service?

Until recently, state management in Grid computing was not a first-class architectural concern. The Grid community largely relied on the Globus Toolkit (Globus Project, 2004), a collection of tools for wide-area, cross-domain computing. Prior to 2002-2003, Globus was not constructed as a collection of services, but rather a collection of semi-independent tools that individually facilitated remote job execution, remote file transfer, and so forth. The Globus Toolkit lacked a single architectural principle with regard to state management.

In 2002-2003, the Open Grid Services Infrastructure (OGSI; Tuecke et al., 2003), under the broader umbrella of the Open Grid Services Architecture (OGSA; Foster et al., 2002), synergized the traditional approach of performing Grid computing via Globus (Globus Project, 2004) or Legion (Grimshaw et al., 1999) with the emerging commercial approach of Web Services. Web Services would provide much of the underlying XML-based protocols for communication between services, while OGSI would provide a canonical rendering of such services. That is, OGSI constrained the appearance (to potential consumers) and behaviors of services, arguing that such constraints would make the overall service composition and subsequent execution more predictable and easier to assess and manage.

In January 2004, a team from IBM and the Globus Alliance introduced the Web Services Resource Framework (WSRF) as an attempt to refactor many of the concepts in OGSI in order to be more consistent with today's Web Services (Czajkowski et al., 2004a). In contrast to early versions of the Globus Toolkit, the central theme of WSRF is the manipulation of state. In the W3C's Web Service Architecture (WSA; Booth et al., 2004), services are either stateless, or any reference to state in the client-server protocol is an application-level concern. In WSRF, the argument is that there is great value in the canonical referencing and/or manipulation of state, paralleling the argument in OGSI that there was great value to the canonical be-

havior and appearance of services. The difference between OGSI and WSRF is that WSRF requires no modification to Web Services tooling. Of course, the significant research challenge for the community is to determine the extent to which WSRF and WS-Notification adds value above the Web Services approach. Almost immediately after the introduction of WSRF, a healthy debate emerged on this subject, particularly its similarities and differences with the Web Services Composite Application Framework WS-CAF (Little, Webber, & Parastatidas, 2004) and REST (Fielding, 2000).

WSRF.NET is an implementation of the WSRF specifications and WS-Notification on the Microsoft .NET Framework and is the first publicly available implementation of the full set of specifications for WSRF and its associated WS-Notification (WSRF Project, 2004). In this paper, we build upon and update our earlier assessment of WSRF.NET (Humphrey et al., 2004). In creating WSRF.NET, we significantly leveraged our experience designing and implementing OGSI on .NET, OGSI.NET (Wasson et al., 2004). We describe how we have interpreted the WSRF and WS-Notification suite of specifications, and, most importantly, we attempt to assess the resulting package, particularly in terms of the programming model. We do not claim that our programming model is the only programming model for WSRF and WS-Notification, but we argue that it is a logical consequence of the implementation of the specifications on the .NET Framework. Additionally, the difference between WSRF and WSRF.NET was difficult at times to discern. Overall, while the potential of WSRF and WS-Notification remains strong, initial observations are that there are many challenges that remain to be solved, most notably the implied programming model derived from the specification, particularly the complexity of service-side code, client-side code, and WS-Notification.

The outline of this paper is as follows. After providing a brief overview of the WSRF approach at the time of this writing, we describe WSRF.NET, which is our open-source implementation of the WSRF suite of specifications. Next, we present a use-case for constructing

and consuming a WSRF-compliant Web Service in WSRF.NET. We discuss a traditional Grid scenario involving remote execution. Following this, we discuss the broader issues and concerns that arise from the resulting implementation. The last section is the conclusion.

WSRF

The core of the WS-Resource Framework (WSRF) is the WS-Resource, a "composition of a Web service and a stateful resource" (Czajkowski et al., 2004a) described by an XML document (with known schema) that is associated with the Web service's port type and addressed by a WS-Addressing EndpointReference (IBM, Bea, Microsoft, 2004). WSRF defines functions that allow interactions with WS-Resources, such as querying, lifetime management, and grouping. WSRF is based on the OGSI specification (Tuecke et al., 2003) and can be thought of as expressing the OGSI concepts in terms that are more compatible with today's Web service standards (Czakowski et al., 2004a; WS-Addressing is currently being standardized in the W3C). Arguably and simplistically, it is sometimes convenient when contrasting OGSI and WSRF to think of OGSI as "distributed objects that conform to many Web Services concepts (XML, SOAP, a modified version of WSDL)," while WSRF is fundamentally "vanilla" Web Services with more explicit handling of state. One artifact of this is that OGSI did not really support interacting with these base Web Services and, instead, only interacted with Grid services (by definition, these were OGSI-compliant); WSRF fully supports interacting with these base Web Services (although the argument is made that client interactions with WSRF-compliant services are richer and easier to manage).

Currently, there are four specifications (WS-Resource Framework, 2004) in the WS-ResourceFramework, with a small number yet to be officially released. WS-ResourceProperties defines how WS-Resources are described by Resource Property (XML) documents that can be queried and modified. Note that the resource property document is a view or projection of the state of the WS-Resource, but it is not equivalent to the state. WS-ResourceLifetime defines mechanisms for destroying WS-Resources (there is no defined creation mechanism). WS-ServiceGroups describe how collections of Web Services and/or WS-Resources can be represented and managed. WS-BaseFaults defines a standard exception-reporting format. WS-RenewableReference (unreleased) will define how a WS-Resource's EndpointReference, which has become invalid, can be refreshed.
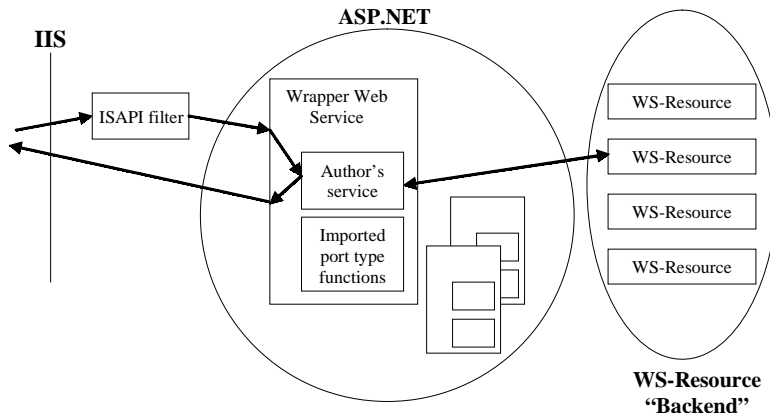
While notification is not technically a required part of the WSRF specifications, it is, nevertheless, an instrumental piece. Many of the WSRF specifications reference notification in a generic manner, so, in all likelihood, WS-Notification will be implemented alongside WSRF. WSRF separates notification into three separate specifications that are conceptually separate but which realistically tend to be grouped together (herein referred to collectively as WS-Notification or, simply, notification). These pieces are WS-BaseNotification (the simplest form of notification possible); WS-BrokeredNotification, which allows for intermediaries and an extra level of abstraction between producers and consumers; and WS-Topics, which is a description of the types of topics that can be considered part of notification. WSRF and WS-Notification are currently being standardized in OASIS (OASIS, 2004).

As mentioned earlier, the core issue regarding WSRF is whether or not state is important enough (and viewed/manipulated often enough by clients) that it should be given a canonical form in the service's interface (the WS-ResourceProperties document).

WSRF.NET

Our original motivation for OGSI.NET was to provide a familiar abstraction (the OGSI abstraction) for intra- and inter-enterprise Grids based solely on .NET. An equal motivation for OGSI.NET was to seamlessly interconnect with the Linux/UNIX OGSI world that would be supported by the Globus Toolkit v3. Upon the introduction of WSRF in January 2004, we immediately decided to implement WSRF on .NET for the same reasons; however, an additional reason was that we believed that we could not

Figure 1. Information flow in WSRF.NET

properly evaluate the core WSRF concepts without an implementation. In other words, we see an important contribution of WSRF.NET as a means by which early adopters can evaluate the WSRF approach for themselves.

It is both positive and negative that, ultimately, we did not reuse as much code from OGSI.NET as we had hoped, in part because the WSRF.NET software architecture more heavily utilized existing Microsoft tooling, as discussed later. That is, whereas OGSI.NET was forced to create a separate infrastructure, WSRF.NET heavily utilizes the Microsoft applications and tooling such as IIS, ASP.NET (Microsoft's support for Web Services/SOAP that is integrated with IIS), and Visual Studio .NET (VS.NET). Note that we still had to write our own WSDL generator. Microsoft chose not to support extensibility attributes on any of the WSDL classes. Microsoft only supports extensibility elements. While this is not WSRF's fault, the requirement of an attribute on the portType to identify the Resource Property Document's schema does make it impossible to comply with WSRF using Microsoft's tools.

In this section, we give an overview of the design of WSRF.NET. More details can be found in the WSRF.NET Programmer's Reference (Wasson 2004). Figure 1 shows how a request message is processed by WSRF.NET. A client request message is first received by IIS. IIS then sends the request to ASP.NET. Inside

ASP.NET, a "wrapper" Web service receives the message. This wrapper was generated by static tooling, based on the Web service written by the service author. Its primary purpose is to provide an ASP.NET-friendly encapsulation for both code written by the service author and functionality they wish to import (such as WSRF spec-defined port types). ASP.NET performs its normal functions of message deserialization, including running the Microsoft Web Services Enhancements (WSE; Microsoft, 2004) pipeline and invoking the correct service method.

Part of the functionality of the wrapper service includes the ability to automatically resolve the execution context specified by the EndpointReference (EPR) and provide a programmatic interface to the appropriate WS-Resource. Although there are potentially many ways in which the execution context could be resolved and many different interfaces for a Web Service to interact with a WS-Resource, we chose to implement WS-Resources using the Microsoft database support through ADO.NET (e.g., MS SQL Server, MSDE, MySQL). Before the wrapper service begins execution of the appropriate method, the object, specified by the Reference Properties element of the EPR, is loaded from the database. It is then made accessible to the Web Service method as if it were a data member of the Web Service class. When the method invocation is complete, the wrap-

per service will save this member's value back into the database. The result of the invocation is then serialized into a SOAP message by ASP.NET and returned via IIS to the client.

WSRF.NET security is and will continue to be based on WSE, with supports WS-Security, WS-SecureConversation, and WS-Policy, as well as many other emerging specifications. Since WSRF.NET Web Services are normal Web Services running under ASP.NET, all WSE features are available.

The *[Resource]* attribute is used in source code to programmatically declare which part(s) of the service state are to comprise the WS-Resource. This is used on data members of the Web Service class (both private and public). By definition, all annotated members are loaded/stored automatically on a per-EPR, per-method basis. As discussed later, processes can also be represented as WS-Resources by putting the [Resource] attribute on WSRF.NET's processHandle type. This information is then stored in a database to bind the handle to a running Win32 process.

Whereas WSRF does not define how to create new WS-Resources, WSRF.NET provides a *Create()* library method. How the service exposes this is up to the service author. The first option is the direct exposure of this method in the Web Service interface. The second option is, instead, to expose some other method, which then invokes the *Create()* operation.

After defining the WS-Resource, the service author typically must define the Resource Properties of the WS-Resource; the Resource Properties Document is the "exposed view" of the WS-Resource. WSRF.NET allows an author to define elements of this document with the *[ResourceProperty]* attribute. When the value of a Resource Property is to be computed dynamically, the service author puts the attribute on any .NET property with a "getter." In .NET, properties are a mechanism for writing accessor functions for data stored in a class's fields. This typically is done when the Resource Property is a function of WS-Resource state and can also have "setter," which is called when a client does a SetResourceProperty. The sum of all [ResourceProperty] annotations defines the Resource Property Document (RPD). WSRF.NET automatically generates the RPD schema, which is an XSD document, and places it in the service's WSDL.

Services are composed potentially of many port types, as some are defined in the WSRF specifications, by the service author, and by others. WSRF.NET provides implementations of the specification-defined port types, and the service author imports these port types into the service by using the [WSRFPortType] attribute. Composing port types into a service means composing functions, composing resources, and composing Resource Properties.
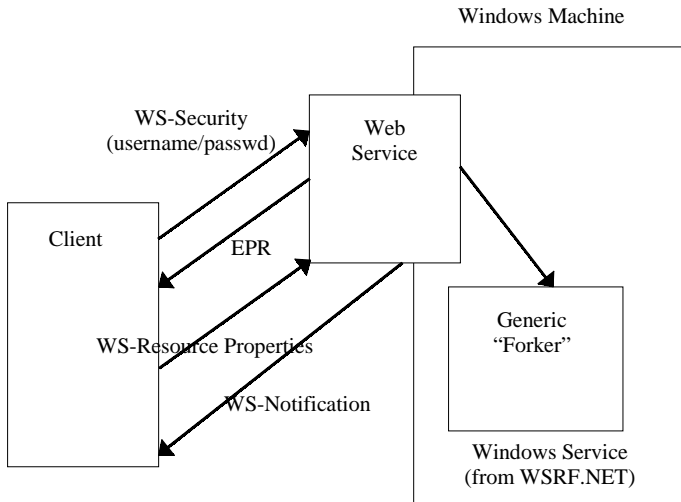
WSRF.NET does this via three mechanisms. First, [WebMethod] annotated members become part of the service's port type (recall that services must have only one port type as per the WS-I Basic Profile). Second, [Resource] annotated members become part of the service's WS-Resource. Third, [ResourceProperty] annotated members become part of the service's Resource Property Document.

The configuration of the WS-Resource is basically the WSE and ASP.NET web.config file, with a new WSRF.NET <config section> used for configuring certain specific WSRF port types. Most services need no WSRF.NET specific configuration.

USING WSRF.NET: THE REMOTE EXECUTION SCENARIO

Figure 2 shows a scenario that we use to show how a person writes a service using WSRF.NET and how a client subsequently uses the service. To support the traditional requirement of remote execution for Grids, in WSRF.NET, we have provided a generic "Proc Spawner", that, when given an authorized username and password, will spawn a process as that user (this is analogous to the procControl-D in Legion and the fork gatekeeper in Globus). In general and in this scenario, a person might choose to expose the ability to execute a particular application via a Web service (as opposed to the ability to ex-

Figure 2. Remote execution via WSRF.NET

ecute any application). This is shown at the right of Figure 2. In the code that follows, this application is called "Sample.exe", and the Web service is called "LaunchSample."

In this example, jobs are WS-Resources, and WSRF is used to expose their state via Resource Properties. That is, a client engages the remote service by securely passing the username and password on the target Windows machine via WS-Security and receives back an EPR. This EPR represents the newly started job. To monitor the job, the client uses WS-ResourceProperties functions (i.e., GetResourceProperties). When the job completes, the client receives an asynchronous notification via WS-Notification. We now describe how the service author creates the WS-Resource and then how the client is written.

Creating the WSRF-Compliant Web Service

The critical step in using any WSRF implementation is first to determine what comprises the state of the service and then to determine what this state's projection (Resource Properties) should be to the clients. Then, to instantiate the service in WSRF.NET, a service author first creates the Web Service using

VS.NET, just as they would any other Web service. Then, the author annotates the service logic with attributes that the WSRF.NET tools recognize and can use to transform the author's compiled Web service into a WSRF-compliant Web service.

Figure 3 shows the completed service for LaunchSample, which contains a representative WSRF-compliant Web service. The LaunchSample class inherits from "ServiceSkeleton" (line 2). We use the ServiceSkeleton class as the glue between the service wrapper and the portType code. The ServiceSkeleton provides a means for accessing the other portTypes that are included in the service. It also provides access to the ResourcePropertiesDocument. In lines 7 to 19, InitResource() lets a portType initialize its resource objects and ResourceProperties when a new WS-Resource is created. InitResource() is not included in the class constructor for LaunchSample, because the class constructors get invoked when the message first hits IIS, so we don't have the information that we need from processing the message at that point. A more important reason is because the class constructor is called every time a Web method invocation comes in and not just when a new WS-Resource is created. We do not want to initialize the resource every time, just once. The

Figure 3. WSRF-compliant launch sample Web service

```
1 [WebService] [WebServiceBinding] [WSRFPortType(typeof(GetResourcePropertyPortType))]
2 public class LaunchSample : ServiceSkeleton
3 {
4   [Resource]
5    private ProcessHandle proc = null;
6
7  public override void InitResource(Hashtable parameters)
8  {
9   // start up the new process that corresponds to  a new WS-Resource
10   parameters =  (Hashtable)(parameters["LaunchSamplePortType"]);
11   string executable = (string)parameters["Executable"];
12   string workingDir =(string)parameters["WorkingDir"];
13   string []args = (string[])parameters["Arguments"];
14   string username = (string)parameters["USERNAME"];
15   string passwd = (string)parameters["PASSWORD"];
16   EndpointReferenceType clientEPR =(EndpointReferenceType)parameters["clientEPR"];
17   proc = ServiceBase.createProcess(username, passwd,  executable, workingDir, args,
18            clientEPR);
19 }
20 [WebMethod]
21 public EndpointReferenceType
22 CreateJob(EndpointReferenceType clientEPR)
23 {
24    string u = null, p = null;
25 // Get the username and password from SOAP headers (Code not shown for brevity)
26 // Assume outcome of this code is u = username (string) and p = password (string)
27   Hashtable ht = new Hashtable();
28   ht["Executable"] = @"C:\Sample\Sample.exe";
29   ht["WorkingDir"] = @"C:\Sample";
30   ht["Arguments"] = new string[] { };
31   ht["USERNAME"] = u; // see comments above
32   ht["PASSWORD"] = p; // see comments above
33   ht["clientEPR"] = clientEPR;
34   Hashtable param = new Hashtable();
35   param["LaunchSamplePortType"] = ht;
36   string rp = ServiceBase.Create(null, param);
37
38 // create the new EPR to return to the client
39   EndpointReference epr = new EndpointReference(new
40            Uri("http://localhost/Sample/LaunchSample.asmx"));
41   ReferenceProperties refProp =  WSUtilities.createReferenceProperties(rp);
42   epr.ReferenceProperties = refProp;
43   EndpointReferenceType eprT =  WSUtilities.convert(epr);
44   return eprT;
45   }
46
47 [ResourceProperty]
48 public double CPUTime {
49    get {   return proc.TotalProcessorTime.TotalMilliseconds;
50    }
51 }
```

InitResource method takes a hashtable that contains the parameters for the various portTypes that make up the service. To prevent collisions, we expect the keys of the hashtable to be the fully qualified name of the class that implements the portType, and it is up to the portType to know the type of the object to which its key points. The only functionality exposed to clients is CreateJob (lines 20 to 45). This method first uses Microsoft's WSE to extract the username and password (lines 25 to 26, not shown). Next, in lines 27 to 35, we con-

struct the parameters necessary to pass to the Generic Proc Spawner included in WSRF.NET. The result of this invocation (line 36) is used in lines 38 to 43 to construct the EPR, which is returned to the client in line 44. Lines 47 to 51 contain the only Resource Property defined for this WS-Resource, which, in this case, is returning the total milliseconds that the job has executed. Note the use of the *get* in line 49 as a means to implement GetResourceProperty.

The normal compile-and-deploy mechanism of VS.NET is overloaded in WSRF.NET to also execute the PortTypeAggregator. The PortTypeAggregator pastes together all the portTypes that make up the service in order to comply with section 4.4 of the WS-ResourceProperties specification. The PortTypeAggregator will automatically invoke the WSDL generator. If the VS.NET add-in has been installed, the PortTypeAggregator gets run as part of the build process, provided that an asmx file is marked as containing a WSRF service.

Creating the
WSRF-Compliant Client

Figure 4 shows the client that interacts with service from Figure 3 in order to execute the Sample.exe application. Line 4 instantiates a proxy to the WSRF-compliant Web Service. The proxy code itself was generated via Add Web Reference option in VS.NET. Note that this is a significant improvement over OGSI.NET, in which the generic Microsoft tooling had no such ability. Line 4 shows that we will use WSE, in this case to securely pass the username and password from the client to the server (lines 6 to 15). Note that, while line 11 implies that we are sending the password in cleartext, we are actually using our X.509 certificate to encrypt the password (see line 6). Lines 17 to 29 are used to create the notification listener that will be used to receive the "*job done*" event. Note that the HandleJobDone function is set as the callback to be called when the notification message on the JobDone topic is received. Line 32 instantiates the data structure, and line 33 places the EPR of the notification listener in the data structure, which will be passed to the Web Service in Line 34. Lines 37-39 turn off en-cryption, specifically for the GetResourceProperty invocation of lines 42-45 (Line 46 writes it on the screen). When the notification message on the JobDone topic is received, the HandleJobDone function is called (lines 49-52). This function could inspect the contents of the message to see if the job completed successfully and either restart it or move on to the next job. Currently, any output produced by the execution of Sample.exe is retrievable via http; this is not shown in Figure 4, for brevity. We are planning more secure access in the near future.

It should be noted that we have experienced difficulties using the WSDL from the specifications with the Microsoft tooling. That is, arguably incorrect proxy code is generated from the WSDL in the specifications. Certain issues are inherent in the Web Services model (e.g., every service used the EndpointReference type, and so, a new version of this type is defined in each service's different and, therefore, incompatible namespace) and not specific to WSRF, but it does cause some problems in that we have to hand-patch the proxy code.

DISCUSSION

There are a number of interrelated observations/concerns we made during the implementation of WSRF.NET.

Design of State: The most important issue for a service author is clearly: What is the appropriate state for a service to expose? How this state is rendered via WSRF (and WSRF.NET) is secondary. With regard to WSRF, it is true that a client doesn't care about the real (private) service state, but to what extent will a client ever care about the projection of that state (Resource Properties)? Further, fundamentally, does there need to be a canonical way of asking for those projections of multiple independent services? These are very difficult questions that cannot be answered at this time.

Client/Service Coupling: There are four tenets to the Microsoft view of a Service-Oriented Architecture: boundaries are explicit; services are autonomous; services share schema and contract, not class; and service compatibility is determined, based on policy (Box 2004). WSRF arguably makes the client and service

Figure 4. WSRF-compliant client for WSRF-compliant Web service in Figure 3

```
1   static void Main(string[] args)
2   {
3     // create a proxy for the job launching service
4     localhost.LaunchSampleServiceWse proxy =     new localhost.LaunchSampleServiceWse();
5     // create the security tokens for the WSE headers
6     X509SecurityToken token =      GetSecurityToken(null);
7     EncryptedData ed = new EncryptedData(token);
8     SoapContext reqContext =         proxy.RequestSoapContext;
9     reqContext.Security.Elements.Add(ed);
10    // use the username and password
11    UsernameToken nt = new    UsernameToken("Fred", "passwd",  PasswordOption.SendPlainText);
12    reqContext.Security.Tokens.Add(nt);
13    reqContext.Security.Elements.Add(new      MessageSignature(nt));
14    EncryptedData ed2 = new EncryptedData(token,    "#"+nt.Id);
15    reqContext.Security.Elements.Add(ed2);
16
17    // create a notification listener
18    AsynchnonousNotificationListener listener = new  AsynchronousNotificationListener();
19    listener.start();
20    EndpointReferenceType myEPR =
21        new EndpointReferenceType(new
22            AttributedURI(null, string.Format("http://localhost:{0}/Listener",
23                          listener.ListeningPort)),
24        null, null, null, null, null);
25    TopicExpression te =
26        WellknownDialects.SIMPLE.createExpression(
27          new XmlQualifiedName("JobDone", "http://gcg.cs.virginia.edu"));
28    te.addHandler(new TopicExpressionListener(HandleJobDone));
29    listener.registerExpression(te);
30
31    // call the service's CreateJob method to launch the job
32    localhost.CreateJob cj = new localhost.CreateJob();
33    cj.clientEPR = myEPR;
34    localhost.CreateJobResponse cr = proxy.CreateJob(cj);
35    EndpointReferenceType eprT = cr.CreateJobResult;
36
37    // stop encrypting and sending Username token now that the job launch is finished
38    reqContext.Security.Elements.Clear();
39    reqContext.Security.Tokens.Clear();
40
41    // set the EPR for the proxy's <To> header to be the/ EPR returned by the CreateJob call
42    WSUtilities.setEPR(proxy, eprT);
43    localhost.GetResourcePropertyResponse grpr = new   localhost.GetResourcePropertyResponse();
44    grpr = proxy.GetResourceProperty(new XmlQualifiedName("CPUTime",
45              "http://gcg.cs.virginia.edu"));
46    Console.WriteLine(grpr.Any[0].OuterXml);
47    // continue application work
48  }
49  private void HandleJobDone(topic t, NotificationMessageHolderType msg)
50  {
51     // handle the fact that the job completed (e.g. retrieve output and start next job in sequence)
52  }
```

more tightly coupled, potentially violating these tenants. Policy assertions will certainly play an important role in WSRF (and all Web Services), but intuitively, it seems as if WSRF-based clients and services share a tighter implicit bond than in generic Web Services. In many WSRF usage scenarios, clients are responsible for the creation, destruction, and maintenance (via, for example, SetResourceProperties) of WS-Resources. The fact that the client is maintaining (or mirroring) this state associated with a service creates this tighter bond. The question is

whether or not this coupling is really any stronger than similar usage scenarios based on "pure" Web Services and whether or not the WS-Resource abstraction provides enough utility to compensate. New specifications for pure Web Services, such as WS-Transfer (Alexander et al., 2004) define a CRUD (create, read, update, delete) interface that can be used for Web Services. It also defines entities being created as resources. If this specification (and subsequent ones built upon it, such as WS-Management; Arora et al., 2004) are to be embraced, the same issues must be dealt with. It could also be argued that part of the tightness of the client-service coupling comes not from WSRF, but from the programming model provided by vendors. All major vendors (including Microsoft) provide an object-oriented model for programming Web Services in which methods are invoked on services by invoking methods on a local "proxy" object. While such proxy objects fit nicely into current programming practices, they imply that Web Services are to be treated like any other distributed object technology (e.g., J2EE [Sun Microsystems, 2005] or Corba [Object Management Group, 2005]). It can be argued that object lifetime management and conversational (typically synchronous) method invocation are not in the spirit of service-oriented architectures (SOAs) and, therefore, an improper use of Web Services. As more advanced Web service programming tools and models become available, hopefully these issues will be diminished for all Web Services (including WSRF-based services).

Complexity of Service-Side Code: First, it is not clear to what extent the service writer must understand the hosting environment. For example, when and how the state is saved/loaded will greatly impact the semantics of the WS-Resource, which inevitably is important to the service writer. This is exacerbated by the composition model, where a service author may import a port type that contains its own resource state, and therefore, what the state of their service's WS-Resource actually is may not be obvious to the service author. This is a property of WSRF, not WSRF.NET. Second, because, by definition, a WS-ResourceProperty is a

projection of the state and not the state itself, there is a decoupling between a service author declaring something to be a WS-Resource and its appearance (or projection) via WS-ResourceProperty. So, a service author has to take two steps: declaration of the state and declaration of the ResourceProperty. This can lead to a situation in which the service author forgets one of the steps altogether, or more likely forgets to change one (e.g., the ResourceProperty) after having changed the other one (the state). Having to do something twice in a programming language is never desired. Arguably, this is not unique to WSRF.NET. In the future, some fundamental link between Resources and Resource Properties (i.e., change one and the other should automatically change) would be part of a higher-level, application-specific, programming model. While some have advocated that a more direct link between resource state and Resource Properties is desirable because of its straightforward mapping onto language/environment specific constructs (e.g., EJBs), others have argued that the lack of explicit linkage is, in fact, desirable. Ultimately, WSRF should be viewed as a set of specifications on which application logic is built. The flexible connection between state and Resource Properties allows many different application-specific connections to be developed. Third, and, perhaps most important, it is not clear to what extent the WSRF rendering of state management results in an unintuitive interface between client and server. Consider a service that performs online tracking of packages. The non-WSRF rendering might have an operation with the following signature: *public bool CheckPkgIn (package pkg, string location).* The WSRF rendering might look like *public bool CheckPkgIn (string location).* The package is not an explicit parameter of the WSRF service, because the package itself is part of the resource, so the "package" is referenced in the SOAP message via a particular EPR (e.g., line 48 of Figure 4). Arguably, as the state of a Web Service becomes more complex in a particular WSRF-compliant Web service, more parameters will seem to disappear in the signature of operations, leaving an unintuitive interface. Note that this does have the advantage of mak-

ing the message smaller and, thus, faster to sign and/or encrypt and send.

Complexity of Client-Side Code: First, implicitly, there is a notion of persistence in WS-Resources, but unfortunately, the semantics of persistence are not precise. What guarantees does the service provide to the client with regard to state? What should the client do when the service has an error saving/restoring a state other than abort the potentially long, complex sequence of operations? Since the definition of persistence is, again, application-specific, WSRF is arguably not the place for it. However, any service will have to answer this question, and potentially complex clients must be built with this understanding. There is also the related issue regarding whether or not and how the client saves the EPRs that it has acquired. Must every client treat EPRs as being analogous to Kerberos tickets, which are stored in the user's file system, knowing that if the EPRs are lost, the client has no way to reengage the WS-Resource? Of course, resource discovery will be used to allow clients to find (and refind) service EPRs, but the question is whether, in any practical application (such as the remote job execution scenario in Wasson and Humphrey [2004]), it will require the discover/indexing infrastructure to prevent the client from having to maintain large amounts of state. Second, by definition, clients must treat EPRs as opaque data structures. EPRs cannot be tested for equality, although they can be tested for inequality. An EPR is not a persistent name, because the service can change it arbitrarily and still refer to the same state. For example, assume that a client gets a service group membership list (SG1). Later, a service S leaves SG1, so the same client gets a notification of an updated list (SG2). While it would seem that SG1 - S = SG2, it could be the case that SG1 and SG2 have no members in common, by the definition of EPRs. In this case, should the client stop doing everything with regard to EPRs in SG1, because it has no guarantee that whichever service it had previously engaged is still in the service group (perhaps S was kicked out of the service group because it is no longer trusted)? Similarly, a client could create a WS-Resource and get EPR1 back. Subsequently, the client could receive a notification containing EPR2 stating that it died. How does the client know if (1) EPR2 and EPR1 are the same resource, or (2) the client was never supposed to receive this message? More abstractly, it seems that "names" can change arbitrarily, and the client must handle it. Many of these issues may be resolved as both the WSRF and WS-Addressing standards progress.

A related, seemingly application-specific issue is the handling of errors. WSRF is silent on failures (except to define the WS-BaseFaults messages that carry fault information), preferring to have WSRF-based applications define their own error semantics. However, it is an open question whether stateful resources assist or inhibit systems architects in handling faults. On the one hand, properly persistent WS-Resources represent a recoverable entity, allowing a service to return to an appropriate state in the event of a server crash (client crashes and EPR discovery are still an issue, as mentioned previously). However, the very fact that a client interacts with a projection of a service's state precludes a set of failure recovery mechanisms that might address the problem by changing internal service operations. In other words, the service must continue to expose its state through the same WSDL-defined schema that the WSRF client was built against. Any additional information that a service must expose limits its ability to change its internal function. As large-scale systems are built on WSRF and traditional Web Services, these issues will become increasingly important.

Discovery: The Service Group is overloaded, in that it supports discovery of WS-Resources, but it also supports general grouping. How this may or may not interact with other Web Services discovery mechanisms (e.g., UDDI) is not clear. Also, we note that there may be scalability issues with service groups, because they must support the "list" operation.

Security Implications of EPRs: If one client passes an EPR to a trusted peer, how would the service that originated the EPR know if this second client has obtained the EPR legitimately? It could be said that the composable nature of

Web Services, specifically with regard to security, will address this. However, the tendency of WS-* authors to rely on this will eventually make the security aspects of these messages too complex to process and/or assemble (and the security architecture overburdened).

Reliability in WS-Notification: Reliability isn't required in the WS-Notification specifications, instead relying on the (optional) WS-ReliableMessaging. Every other asynchronous messaging framework that we have worked with has touted reliable messaging as its best and most important feature (e.g., JMS; Hapner et al., 2002). An example of problems that arise because of this is the use or reference to WS-Notification in specifications like WS-ResourceLifetime, where it is all but stated that death notifications can be used for cleanup. Without reliable messaging, this kind of use is questionable, at best. While it is certainly true that not all notification messages need to be reliable (and so requiring reliability would be too heavyweight), it is not clear how often real systems will not require any delivery guarantees.

Interoperability in WS-Notification: Two issues in WS-Notification significantly impact the potential for interoperable implementations. First, the raw method delivery of a notification message is particularly problematic, as the specifications states: "In this case the NotificationConsumer referred to in the wsnt:ConsumerReference element MUST implement all the NotificationMessages associated with the Topic or Topics referred to by the TopicExpression, and include corresponding operations in its portType definition" (Graham et al., 2004). However, the NotificationMessages associated with the Topic or Topics is ambiguous. One can assume that this refers to messages that, via some well-defined pattern-matching scheme, are of well-known name and type in the consumer's interface. Even if this pattern is assumed to be extremely straightforward, for example, the name of the method is the name of the topic and in the same namespace; then, the question of parameters is still undefined. To our knowledge, none of the specifications states the information passed with a notification, thus making interoperable raw message delivery challenging. Second, interoperability will be difficult to achieve, given the lack of sufficient definition for the SubscriptionManager and PublisherRegistrationManager port types in the WS-Notification spec (and indeed, this problem also crops up in other specs, such as WS-ServiceGroup). These two port types store, manipulate, and reference subscriptions that consumers have made to producers and registrations of publishers to brokers, respectively. In particular, if you take subscriptions and SubscriptionManagers as an example, when a consumer subscribes to a producer on a given topic, a new WS-Resource for a SubscriptionManager port type is created to represent that subscription. It is this WS-Resource that clients then later can pause, unpause, destroy, and so forth. However, notably lacking in the definition of this port type are how the subscriptions are created (there is no explicit factory mechanism defined in WSRF), and how those subscriptions are retrieved by the notification producer, when it is time to send out notifications. In both cases, this detail is considered implementation-specific. However, this means that all notification producers and brokers must be implemented with a specific, non-standard way of creating and retrieving subscriptions and registrations. This is indicative of a larger problem regarding interoperability in WSRF, which is the lack of standardized mechanisms to create WS-Resources.

Complexity and Atomicity in WS Notification: WS-Notification, in particular, has a rather large amount of complexity built in. Take, for example, the process by which, in the limit, the notification broker must go through in order to create a demand-based publisher. The broker receives a registration from a publisher and, as a result, must make a subscription back to the publisher, based on the registered topic/topics. This subscription is maintained, as always, by a subscription manager, but now the broker is also responsible for pausing and unpausing it, based on the state of the subscriptions that other consumers have to their own resource on the given topics. If no subscriptions currently exist to the broker on a

given topic, then all subscriptions for demand-based publishers on the same topic must, according to the spec, be paused. In total, when you consider the interactions between these various services and resources, a demand-based publisher registration interaction can involve as many as six separate Web Services and WS-Resources. More messages are generated in response to a demand-based publisher scenario than in any other spec by what we estimate to be an order of magnitude at a minimum. Further, the WSRF specification does not address the topic of atomicity in its state transitions, while this is perfectly acceptable in many of the simpler interactions between the various Grid services, the need for some kind of transactional semantics becomes increasingly clear in the more complicated scenarios hinted at by WS-Notification and WS-BrokeredNotification. In response to a subscription being created or being destroyed in a demand-based publisher scenario, messages must be sent out, which may further cause other messages to be produced, any of which may fail for any reason, including the more common one of network failure. These failures by specification must affect all the other messages, as well, and, while best effort has been made to achieve this, the fine granularity of the port types and the large number of services involved makes this incredibly difficult and not nearly as reliable, as the same situation would be in the case where the broker kept all of the subscriptions and registrations as part of an internal table.

CONCLUSION

Some in the Web Services community have argued that we're entering the contraction phase of Web Services, whereby people realize that too many moving parts potentially compromise the core interoperability story. However, we cannot ignore the fact that complex application logic may require complex infrastructure in order to support it. In the end, WSRF and WS-Notification should be viewed as building blocks on which applications or other "higher-level" infrastructure can be built selectively. WSRF's importance as a building block comes from its argument that canonical exposure to and manipulation of state by clients is important. Given that most Web Services contain state, this argument may be valid, even if it comes with the risk of added complexity. We have implemented WSRF.NET to enable our project and others to evaluate this approach through hands-on experience. While the potential of WSRF remains strong, a number of concerns have been raised. It can be difficult to decouple WSRF and WS-Notification from the higher-level Grid functionality that we know we want to build, using these specifications. In general, there is a balancing act between creating specifications that provide functionality that can be composed into many different usage patterns and creating specifications that are too vague to be used effectively. While the WSRF specifications may imply a (possibly complex) programming model, we feel that this complexity may not be unwarranted. The Technical Committees of OASIS for WSRF and WS-Notification will undoubtedly address and improve many of our concerns, and more experience with implementations, such as WSRF.NET, will provide insight into understanding the general usability of WSRF.

REFERENCES

Alexander, J., et al. (2004). Web Services transfer (WS-Transfer). Retrieved from *http://msdn. microsoft.com/library/en-us/dnglobspec/ html/ws-transfer.pdf*

Arora, A., et al. (2004). Web Services for management (WS-Management). Retrieved from *http://www.intel.com/technology/manage/downloads/ws_management.pdf*

Booth, D., et al. (Eds.). (2004). Web Services architecture. W3C Web Services architecture working group. Retrieved February 11, 2004, from *http://www.w3.org/TR/ws-arch*

Box, D. (2004). A guide to developing and running connected systems with indigo. *MSDN Magazine, 1*(1).

Czajkowski, K., et al. (2004a). From open grid services infrastructure to Web Services resource framework: Refactoring and evolution. Retrieved from *http://www-106.ibm.com/developerworks/library/ws-resource/grogsitowsrf.html*

Czajkowski, K., et al. (2004b). The WS-resource framework. Retrieved from http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf

Fielding, R. (2000). *Architectural styles and the design of network-based software architectures* (doctoral thesis). Irvine, CA: Department of Computer Science, University of California at Irvine.

Foster, I, Kesselman, C., Nick, J., & Tuecke, S. (2002). The physiology of the grid: An open grid services architecture for distributed systems integration. Retrieved June 22, 2002, from *http://www.Gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf*

Globus Project. (2004). Retrieved from *http://www.globus.org*

Graham, S., et al. (2004). Web Services base notification (WS-base notification). Retrieved March 5, 2004, from *http://ifr.sap.com/ws-notification/WS-BaseNotification.pdf*

Grimshaw, A., Ferrari, A., Knabe, F., & Humphrey, M. (1999). Wide-area computing: Resource sharing on a large scale. *IEEE Computer*, *32*(5), 29-37.

Hapner, M., Burridge, R., Sharma, R., Fialli, J., & Stout, K. (2002). Java message service. *Sun Microsystems Inc.*, (p. 14). Retrieved from *http://java.sun.com/products/jms/docs.html*

Humphrey, M., Wasson, G., Morgan, M., & Beekwilder, N. (2004). An early evaluation of WSRF and WS-notification via WSRF.NET. In *Proceedings of the 2004 Grid Computing Workshop (Supercomputing)*.

IBM, BEA, & Microsoft. (2004). WS-Addressing. Retrieved from *http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-addressing.asp*

Little, M., Webber, J., & Parastatidas, S. (2004). Stateful interactions in Web Services: A comparison of WS-context and WS-resource framework. *Web Services Journal.* Retrieved from *http://www.sys-con.com/story/?storyid=44675&DE=1*

Microsoft Corporation. (2004). Web Services enhancements, version 2.0. Retrieved from *http://msdn.microsoft.com/webservices/building/wse/default.aspx*

OASIS Web Services Resource Framework (WSRF) Technical Committee. (2005). Retrieved from *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf*

Object Management Group. CORBA. (2005). Retrieved from *http://www.corba.com*

Sun Microsystems. (2005). Java 2 platform enterprise edition. Retrieved from *http://java.sun.com/j2ee/*

Tuecke, S., et al. (2003, June 27). Open grid services infrastructure (OGSI) version 1.0. *Global Grid Forum (GFD-R-P.15)*.

Wasson, G. (2004). WSRF.NET programmer's reference. Retrieved from *http://www.cs.virginia.edu/~gsw2c/ WSRFdotNet/WSRFdotNet_programmers_reference.pdf*

Wasson, G., Beekwilder, N., Morgan, M., & Humphrey, M. (2004). OGSI.NET: OGSI-compliance on the .NET Framework. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*.

Wasson, G., & Humphrey, M. (2005). Exploiting WSRF and WSRF.NET for remote job execution in grid environments. In *Proceedings of the International Parallel and Distributed Processing Symposium*.

WS-ResourceFramework and WS-Notification Specifications. (2004). Retrieved from *http://devresource.hp.com/drc/specifications/wsrf/index.jsp*

WSRF.NET Project. (2005). Retrieved from *http://www.ws-rf.net*

Marty Humphrey received a BS in electrical and computer engineering in 1986, an MS in electrical and computer engineering in 1988 from Clarkson University, Potsdam, New York, and a PhD in computer science from the University of Massachusetts in 1996. From 1998 to the present, he has been with the Department of Computer Science at the University of Virginia, Charlottesville, Virginia, where he was first a research assistant professor and is currently (2002-) an assistant professor. His areas of research include many aspects of Grid computing,

including security, programming models, performance, Grid testing, and Grid usability. He is active in the Global Grid Forum, where he just completed a term as co-director of the Security Area and also as a member of the steering committee.

Glenn Wasson received a BS in electrical engineering from Tufts University in 1993 and an MS in computer science from the Computer Science Department at the University of Virginia in 1995. He received a PhD from the Computer Science Department of the University of Virginia in 1999. He then served as a research assistant professor at the Medical Automation Research Center in Charlottesville, Virginia, until 2001, when he joined the Grid Computing Group in the Computer Science Department at the University of Virginia. He is currently a senior scientist. Wasson's research interests include Grid computing, distributed and complex systems, and system architectures.