

An Early Evaluation of WSRF and WS-Notification via WSRF.NET*

Marty Humphrey, Glenn Wasson, Mark Morgan, and Norm Beekwilder
Department of Computer Science
University of Virginia
Charlottesville, VA 22904
humphrey@cs.virginia.edu

Abstract

The Web Services Resource Framework (WSRF) and its companion WS-Notification were introduced in January 2004 as a new model on which to build Grids. This paper contains early observations made while implementing the full suite of WSRF and WS-Notification specifications on the Microsoft .NET Platform. 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.

1. Introduction

In 2002-2003, the Open Grid Services Infrastructure (OGSI) [1], under the broader umbrella of the Open Grid Services Architecture (OGSA) [2], synergized the traditional approach of performing Grid Computing via Globus [3] or Legion [4] 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.

The problem with OGSI is that it is not a pure subset or constraining of Web Services, whereby a client could as easily engage an OGSI-compliant Web

Service as engage a *non*-OGSI-compliant Web Service. First, OGSI required a modification to the Web Services Description Language (WSDL) that was eventually called the Grid-WSDL (or GWSDL). This meant that special tools were needed to parse and process WSDL for these *Grid Services* and that commercial and non-commercial WSDL tools could not be used. Second, OGSI created a "factory" for the creation of Grid Services, thereby supporting transient, potentially short-lived services. In pure Web Services, the creation of a new Web Service is largely outside the basic model. The focus in Web Services was *not* how these services came into existence, but rather on the *messages* (i.e., wire format) sent between them. Many people interpreted the OGSI factory as the imposition of an object model on Web Services, whereas the commercial Web Services community was trying to move into a much more loosely-coupled model.

In January 2004, a team from IBM and the Globus Alliance introduced the Web Services Resource Framework (WSRF) as an attempt to *re-factor* many of the concepts in OGSI to be more consistent with today's Web Services [5]. The central theme of WSRF is the manipulation of state. In the W3C's Web Service Architecture (WSA) [6], 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 was that there is great value to the canonical behavior 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

* This work was supported in part by the National Science Foundation under grants ACI-0203960 (Next Generation Software program), ANI-0222571 (NSF Middleware Initiative), the National Partnership for Advanced Computational Infrastructure (NPACI), SCI-0123937 (through a subcontract to SURA), and Microsoft Research.

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 [7]) and REST [8].

The contribution of this paper is that it is the first work to attempt to assess the value of WSRF and WS-Notification through a concrete implementation, in this case based on the Microsoft .NET platform (WSRF.NET). In creating WSRF.NET, we significantly leveraged our experience designing and implementing OGSi on .NET, OGSi.NET [9]. In this paper, we describe how we have interpreted the WSRF and WS-Notification suite of specifications and most importantly attempt to assess the resulting package, particularly in terms of the programming model. We don't 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 platform. 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. Section 2 contains a brief overview of the WSRF approach as of the time of this writing. Section 3 contains a description of WSRF.NET, which is our open-source implementation of the WSRF suite of specifications. Section 4 describes a use-case scenario for constructing and consuming a WSRF-compliant Web Service in WSRF.NET. We discuss a traditional Grid scenario involving remote execution. Section 5 contains a discussion of the issues and concerns of the resulting implementation. Section 6 is the conclusion.

2. WSRF

The core of the WS-Resource Framework (WSRF) is the WS-Resource, a "composition of a web service and a stateful resource" [10] 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 [11]. WSRF defines functions that allow interactions with WS-Resources such as querying, lifetime

management and grouping. WSRF is based on the OGSi specification [1] and can be thought of as expressing the OGSi concepts in terms that are more compatible with today's web service standards [10] (a notable exception is the use of WS-Addressing, which is not a standard as of the time of this writing). 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 4 specifications [12] 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 which 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.

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).

3. WSRF.NET

Our original motivation for OGS.NET was to provide a familiar abstraction (the OGS abstraction) for intra- and inter-enterprise Grids based solely on .NET. An equal motivation for OGS.NET was to seamlessly interconnect with the Linux/UNIX OGS 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 properly evaluate the core WSRF concepts without an implementation. In other words, we see an important contribution of WSRF.NET is 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 re-use as much code from OGS.NET as we had hoped, in part because the WSRF.NET software architecture more heavily utilized existing Microsoft tooling, as discussed below. That is, whereas OGS.NET was forced to create 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 isn't 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 [13]. Figure 1 shows how a request message is processed by WSRF.NET. A client request message is first received by IIS. An ISAPI filter² is run to ensure that requests to arbitrarily named endpoints will be correctly

² ISAPI filters provide a hook to insert custom actions very early in IIS's processing of messages. The proper extension (.asmx) is added to the request endpoint name in order to cause IIS to dispatch the request to the ASP.NET worker process.

dispatched. IIS then sends the request to ASP.NET. Inside ASP.NET, a "wrapper" web service receives the

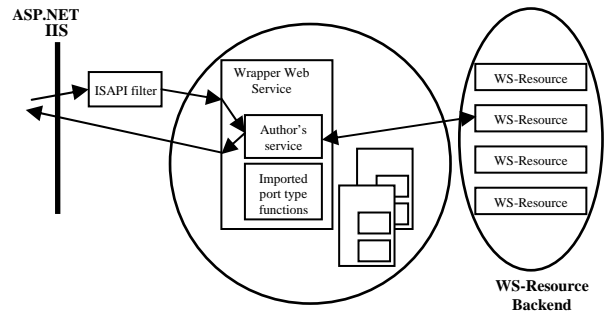


Figure 1. Information Flow in WSRF.NET

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) [14] 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 wrapper 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 of the service

state that is to comprise the WS-Resource. This is used on class-level data members (both private and public). By definition, all annotated members are loaded/stored automatically on a per-EPR, per-method basis. As shown in Section 4, 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 to instead expose some other method, which then invokes the *Create()* operation.

After defining the WS-Resource, the service author must typically 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 is typically done when the Resource Property is a function of WS-R state, and can also have "setter" which is called when a client does a *SetResourceProperty*. [*ResourceProperty*] can also be put on the class, when the value does not have to be computed dynamically. 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 of potentially many port types, as some are defined in the WSRF specifications, and some are defined by the service author and 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. Second, [*Resource*] annotated members become part of service's WS-Resource. Third, [*ResourceProperty*] annotated members become part of 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

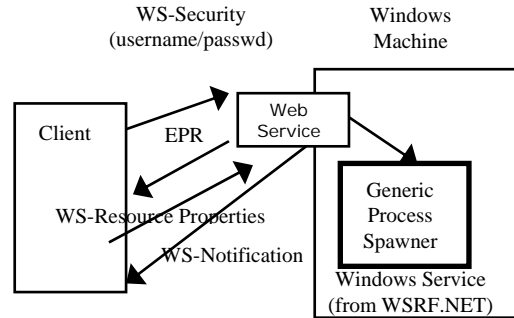


Figure 2. Remote Execution via WSRF.NET

configuring certain specific WSRF port types. Most services need no WSRF.NET specific configuration.

4. 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 execute *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".

WSRF is used to expose the state of the running job via the Resource Properties of a WS-Resource. 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 an EPR back. To monitor the job, the client uses WS-ResourceProperties. 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.

4.1 Creating the WSRF-compliant Web Service

The critical step in using any WSRF implementation is to first determine what comprises

the state of the service, and then to determine what this state's

```

1 [WebService]
2 [WebServiceBinding]
3 [WSRFPortType(typeof(GetResourcePropertyPortType))]
4 public class LaunchSample : ServiceSkeleton
5 {
6
7 [Resource]
8 private ProcessHandle proc = null;
9
10 public override void InitResource(Hashtable parameters)
11 {
12 // start up the new process that corresponds to
13 // a new WS-Resource
14 parameters =
15 (Hashtable)(parameters["LaunchSamplePortType"]);
16 string executable = (string)parameters["Executable"];
17 string workingDir = (string)parameters["WorkingDir"];
18 string []args = (string[])parameters["Arguments"];
19 string username = (string)parameters["USERNAME"];
20 string passwd = (string)parameters["PASSWORD"];
21 EndpointReferenceType clientEPR =
22 (EndpointReferenceType)parameters["clientEPR"];
23
24 proc = ServiceBase.createProcess(username, passwd,
25 executable, workingDir, args,
26 clientEPR);
27 }
28 [WebMethod]
29 public EndpointReferenceType
30 CreateJob(EndpointReferenceType clientEPR)
31 {
32 string u = null, p = null;
33
34 // Get the username and password from SOAP headers
35 // Code not shown for brevity
36 // Assume outcome of this code is u = username (string)
37 // and p = password (string)
38
39 Hashtable ht = new Hashtable();
40 ht["Executable"] = @"C:\Sample\Sample.exe";
41 ht["WorkingDir"] = @"C:\Sample";
42 ht["Arguments"] = new string[] { };
43 ht["USERNAME"] = u; // see comments above
44 ht["PASSWORD"] = p; // see comments above
45 ht["clientEPR"] = clientEPR;
46
47 Hashtable param = new Hashtable();
48 param["LaunchSamplePortType"] = ht;
49
50 string rp = ServiceBase.Create(null, param);
51
52 // create the new EPR to return to the client
53 EndpointReference epr = new EndpointReference(new
54 Uri("http://localhost/Sample/LaunchSample.asmx"));
55 ReferenceProperties refProp =
56 WSUtilities.createReferenceProperties(rp);
57 epr.ReferenceProperties = refProp;
58 EndpointReferenceType eprT =
59 WSUtilities.convert(epr);
60
61 return eprT;
62 }
63
64 [ResourceProperty]
65 public double CPUTime {
66 get {
67 return proc.TotalProcessorTime.TotalMilliseconds;
68 }
69 }
70

```

Figure 3: WSRF-compliant "Launch Sample" Web Service

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 4). 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 10-26, 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 at that point. Another reason is because the class constructor is called every time a web method comes in and not just the first time. We don't want to initialize the resource every time, just once. This method takes a hashtable which 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 that its key points to. The only functionality exposed to clients is

```

1  static void Main(string[] args)
2  {
3  // create a proxy for the job launching service
4  localhost.LaunchSampleServiceWse proxy =
5      new localhost.LaunchSampleServiceWse();
6  // create the security tokens for the WSE headers
7  X509SecurityToken token =
8      GetSecurityToken(null);
9  EncryptedData ed = new EncryptedData(token);
10 SoapContext reqContext =
11     proxy.RequestSoapContext;
12 reqContext.Security.Elements.Add(ed);
13 // use the username and password
14 UsernameToken nt = new
15     UsernameToken("Fred", "passwd",
16     PasswordOption.SendPlainText);
17 reqContext.Security.Tokens.Add(nt);
18 reqContext.Security.Elements.Add(new
19     MessageSignature(nt));
20 EncryptedData ed2 = new EncryptedData(token,
21     "#" + nt.Id);
22 reqContext.Security.Elements.Add(ed2);
23
24 // create a notification listener
25 BlockingNotificationListener listener = new
26     BlockingNotificationListener();
27 listener.start();
28 EndpointReferenceType myEPR =
29     new EndpointReferenceType(new
30     AttributedURI(null,
31     string.Format("http://localhost:{0}/Listener", 32
32     listener.ListeningPort)),
33     null, null, null, null, null);
34
35 // call the service's CreateJob method to launch the job
36 localhost.CreateJob cj = new localhost.CreateJob();
37 cj.clientEPR = myEPR;
38 localhost.CreateJobResponse cr = proxy.CreateJob(cj);
39 EndpointReferenceType eprT = cr.CreateJobResult;
40
41 // stop encrypting and sending Username token
42 // now that the job launch is finished
43 reqContext.Security.Elements.Clear();
44 reqContext.Security.Tokens.Clear();
45
46 // set the EPR for the proxy's <To> header to be the
47 // EPR returned by the CreateJob call
48 WSUtilities.setEPR(proxy, eprT);
49 localhost.GetResourcePropertyResponse grpr = new
50     localhost.GetResourcePropertyResponse();
51 grpr = proxy.GetResourceProperty(new
52     XmlQualifiedName("CPUTime",
53     "http://gcg.cs.virginia.edu"));
54 Console.WriteLine(grpr.Any[0].OuterXml);
55
56 ArrayList messages = listener.waitForMessages(0,
57     1200000);
58 NotificationMessageHolderType[] nmht =
59     messages[0] as
NotificationMessageHolderType[];
60 Console.WriteLine("Topic = " + nmht[0].Topic.topic);
61 Console.WriteLine("Message = " +
62     nmht[0].Message.OuterXml);
63 Console.WriteLine("Producer = " +
64     nmht[0].ProducerReference.Address.Value);
65 }
66

```

Figure 4: WSRF-compliant Client for "Launch Sample" WSRF-compliant Web Service

CreateJob (lines 28-62). This method first uses Microsoft's WSE to extract the username and password (lines 34-37, not shown). Next, in lines 39-49, we construct the parameters necessary to pass to the Generic Proc Spawner included in WSRF.NET. The result of this invocation (line 50) is used in lines 52-59 to construct the EPR, which is returned to the client in line 61. Lines 64-69 contain the only Resource Property defined for this WS-Resource, in this case returning the total milliseconds that the job has executed. Note the use of the "get" in line 66 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, to comply with section 4.4 of the WS-ResourceProperties specification. The PortTypeAggregator will automatically invoke the

WSDL generator. If the VS.NET addin 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.

4.2 Creating the WSRF-compliant Web Service

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 OGSINET, in which the generic Microsoft tooling had no such ability. Line 5 shows that we will use WSE, in this case to securely pass the username and password from the client to the

server (lines 6-22). Note that line 16 implies that we are sending the password in cleartext; however, we are actually using our X.509 certificate to encrypt the password (see line 7). Lines 24-33 are used to create the notification listener that will be used to receive the "job done" event. Line 36 instantiates the data structure---and line 37 places the EPR of the notification listener in the data structure---that will be passed to the Web Service in Line 38. Lines 41-44 turn off encryption, specifically for the GetResourceProperty invocation of lines 46-53 (Line 54 writes it on the screen). Lines 56-64 receive the consumer---that this client is written knowing that Sample.exe will complete within a certain time (lines 56-57); a more unpredictable environment would require more sophisticated code in lines 56-64. 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. This is inherent in the Web Services model and not specific to WSRF but it does cause some problems in that we have to hand-patch the proxy code.

5. 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? 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 service state, but to what extent will a client ever care about the projection of 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 tenants to the Microsoft view of a Service-Oriented Architecture: boundaries are explicit; services are autonomous; services share schema and contract, not class; service compatibility is determined based on policy [15]. WSRF arguably makes the client and service 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. Certainly the client must know about creation and destruction of relevant WS-Resources.

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, in which case it may not be obvious *what* this extra state actually is. This is a property of WSRF, not WSRF.NET. Second, because by definition the WS-ResourceProperty is a *projection* of the state, and not the state itself, there is a decoupling between the 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 then 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 (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. It has been observed that some proponents of WSRF want to enforce a link between the two because they want Resource Properties to be the properties of EJBs. Third, and perhaps most importantly, 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 on-line 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 making the message smaller, 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 state other than abort the potentially long, complex sequence of operations? 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 re-engage the WS-Resource? Second, by definition, clients must treat EPRs as opaque data structures. EPRs cannot be tested for equality. An EPR is not a persistent name, because the service can arbitrarily change it 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 SG2, 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 in 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 "it died". How does the client know if (a) EPR2 and EPR1 are the same resource, or (b) the client was never supposed to receive this message? More abstractly, it seems that "names" can change arbitrarily, and the client must handle it.

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 Service Groups are not scalable in that 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 [16]). 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.

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." [17] However, the "NotificationMessages associated with the Topic or Topics" is ambiguous. One can assume that this refers to messages which, 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 straight-forward, for example, the name of the method is the name of the topic and in the same namespace, etc., then the question of parameters is still undefined. To our knowledge, none of the specifications state 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, and new WS-Resource for a SubscriptionManager port type is created to represent that subscription. It is this WS-Resource that clients can then later pause,

unpause, destroy, etc. 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 his 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 6 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 must by specification 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.

6. 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 compromises the core interoperability story. However, undeniably, most Web Services contain state, and WSRF is an argument that canonical exposure to and manipulation by the client is needed, at perhaps the risk of adding complexity. We have implemented WSRF.NET to enable our project and others to evaluate this approach through hands-on experience. While we believe the potential of WSRF remains strong, a number of concerns have been raised. It is difficult to decouple WSRF and WS-Notification from an implied (arguably complex) programming model, although this decoupling is *exactly* what WS specifications are trying to achieve. We believe that the Technical Committees of OASIS for WSRF and WS-Notification will address and improve many of our concerns, but more experience with implementations such as WSRF.NET are necessary to evaluate some of the more complex issues such as general usability.

7. References

- [1] S. Tuecke et. al. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum. GFD-R-P.15. Version as of June 27, 2003.
- [2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Draft of 6/22/02. http://www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf
- [3] Globus Project. <http://www.globus.org>
- [4] A.S. Grimshaw, A.J. Ferrari, F.C. Knabe and M.A. Humphrey, "Wide-Area Computing: Resource Sharing on a Large Scale," IEEE Computer, 32(5): 29-37, May 1999.
- [5] K. Czajkowski, Ferguson, D., Foster, I., Frey, J., Graham, S., Snelling, D., Tuecke, S., From Open Grid Services Infrastructure to Web Services Resource Framework: Refactoring and Evolution, 2004. <http://www-106.ibm.com/developerworks/library/ws-resource/gr-ogsitowsrf.html>

- [6] D. Booth *et. al.*, editors. Web Services Architecture. W3C Web Services Architecture Working Group. Version of 11 February 2004. Available at: <http://www.w3.org/TR/ws-arch/>.
- [7] M. Little, J. Webber, and S. Parastatidas. Stateful Interactions in Web Services: A Comparison of WS-Context and WS-Resource Framework. *Web Services Journal*. April 30, 2004. Available at: <http://www.sys-con.com/story/?storyid=44675&DE=1>
- [8] R.T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. Ph.D. Dissertation. Dept. of Computer Science. University of California at Irvine. 2000.
- [9] G. Wasson, N. Beekwilder, M. Morgan, and M. Humphrey. OGS.NET: OGS-compliance on the .NET Framework. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*. April 19-22, 2004. Chicago, Illinois.
- [10] K. Czajkowski., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W. 2004. The WS-Resource Framework. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf>
- [11] IBM, BEA, and Microsoft. WS-Addressing. 2004. <http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-addressing.asp>
- [12] WS-ResourceFramework and WS-Notification Specifications. <http://devresource.hp.com/drc/specifications/wsrf/index.jsp>
- [13] G. Wasson. WSRF.NET Programmer's Reference. Available at: http://www.cs.virginia.edu/~gsw2c/WSRFdotNet/WSRFdotNet_programmers_reference.pdf
- [14] Microsoft. Web Services Enhancements version 2.0. Available at: <http://msdn.microsoft.com/webservices/building/wse/default.aspx>
- [15] D. Box. "A Guide to Developing and Running Connected Systems with Indigo." *MSDN Magazine*. January 2004. Volume 19, number 1.
- [16] Hapner, Mark. Burridge, Rich. Sharma, Rahul. Fialli, Joseph. Stout, Kate. Java Message Service. . Version 1.1 April 12, 2002. Sun Microsystems Inc. pp. 14. <http://java.sun.com/products/jms/docs.html>
- [17] Graham, Steve. Niblett, Peter. Chappell, Dave. Lewis, Amy. Nagaratnam, Nataraj. Parikh, Jay. Patil, Sanjay. Samdarshi, Shivajee. Sedukhin, Igor. Snelling, David. Tuecke, Steve. Vambenepe, William. Weihl, Bill. Web Services Base Notification (WS-Base Notification). Version 1.0. 3/5/2004. <http://ifr.sap.com/ws-notification/WS-BaseNotification.pdf>