Preliminary version; final version to appear in *Proceedings of Supercomputing 2005*, Seattle, WA, Nov 12-18, 2005

Alternative Software Stacks for OGSA-based Grids*

Marty Humphrey, Glenn Wasson, Yuliyan Kiryakov, Sang-Min Park, David Del Vecchio, Norm Beekwilder
University of Virginia
Jim Gray
Microsoft Research

Abstract: *The Open Grid Services Architecture (OGSA) has been a major step forward for Grid Computing, but its de facto reliance on the Web Services Resource Framework (WSRF) and WS-Notification have left some in the community questioning if the Grid software and protocols have become too complex. In this paper, we consider the feasibility of an "alternative" software stack, based on WS-Transfer and WS-Eventing. We compare and evaluate WSRF/WS-Notification vs. WS-Transfer/WS-Eventing, first via a "hello world" service and then via a more complete Grid scenario for remote execution called "Grid-in-a-Box". By qualitatively and quantitatively evaluating each approach, we find that subtle differences exist (e.g., the naming of resources, the creation of resources) but they are overwhelmingly equivalent in their functionality and implied performance. Overall, this paper uniquely shows that there could be alternative software stacks for OGSA-based Grids.*

1 Introduction

While few people argue against the potential of Web services as the infrastructure upon which to re-factor and extend traditional Grid Computing such as Legion [1] and Globus [2], some people in the community believe that the *de facto* reliance in the Open Grid Services Architecture (OGSA) [3] on the Web Services Resource Framework (WSRF) [4] and WS-Notification (WSN) [5] will lead to fragile and non-scalable systems. A second, alternative approach that is increasingly discussed is based on WS-Transfer [6] (with its REST-style four basic operations of *Get, Put, Delete,* and *Create)* and WS-Eventing [7]. Because there is a no concrete comparison of the two approaches, the community discussion regarding the pros/cons of each is often largely based on intuition. Without a specific comparison, this discussion will continue to polarize the Grid community, with implications on the broader Web services community.

The contribution of this paper is the first tangible, qualitative and quantitative comparison of WSRF/WSN and WS-Transfer/WS-Eventing, first via a "hello world" service and then via a more complete Grid scenario for remote execution we refer to as "Grid-in-a-Box". By evaluating each approach, we find that subtle differences exist (e.g., the naming of resources, the creation of resources) but they are overwhelmingly equivalent in their functionality and implied performance.

The *goal* of WSRF/WSN, which is easier client-driven state management in Web services with specific applications to Grid computing, is itself controversial (in fact, resource-orientation is not the only approach to building Grid applications -- see WS-GAF [8] for a non-resource approach). WSRF and WS-Transfer at their core expose a simple get/set interface to resource state (and appear to be an excellent replacement for SNMP) but it is not clear if it is a general architectural framework upon which to build applications. In this paper, we only indirectly address the value of client-side state management because it is difficult to assess definitively. Instead, to make a concrete contribution based on measurable experiments, we largely assume that this goal of state management is appropriate and accomplishable, and that the issue is whether or not WSRF/WSN is the (only) feasible approach by which to do this. Overall, by analyzing: (a) the two sets of specifications, (b) our implementation of the two sets of specifications, and (c) the use of these implementations for a single service and a more complex combination of services, this paper uniquely shows that there can be alternative software stacks for OGSA-based Grids.

This paper is organized as follows. In Section 2, we describe the two sets of specifications: first WSRF and WS-Notification and then WS-Transfer and WS-Eventing. Section 3 describes our experiences gained

* The University of Virginia authors are supported in part by the US National Science Foundation under grants ACI-0203960, SCI-0438263, SCI-0426972, the Department of Energy Early Career program (to Humphrey), and the San Diego Supercomputing Center.

while implementing the two sets of specifications: WSRF/WSN in Section 3.1 and WS-Transfer in Section 3.2 (a free implementation of WS-Eventing is also discussed here). Section 4 contains a qualitative and quantitative comparison of the "hello world" application and the more full-featured "Grid-in-a-box" application. Section 5 contains a discussion of our findings and Section 6 concludes the paper with some implications for the community.

2 Specifications for Managing State and Asynchronous Notifications

2.1 WS-Resource Framework (WSRF) and WS-Notification

WSRF [4] was introduced in January 2004 with the goal of defining conventions for representing, abstracting, and manipulating state in a Web services framework. WSRF defines the WS-Resource construct, a "composition of a Web service and a stateful resource" described by an XML document (with known schema) that is associated with the Web service's port type and addressed by one of the WS-Resource Access Patterns--the most common of which is to use a WS-Addressing Endpoint Reference (EPR) [9]. The four WSRF specifications being standardized in OASIS [10] define how to represent, access, manage, and group WS-Resources:

- **WS-ResourceProperties** defines how WS-Resources are described by XML documents that can be queried and modified. This document is a view or projection of the state of the WS-Resource and is typically not equivalent to the state.
- **WS-ResourceLifetime** defines mechanisms for destroying WS-Resources. ("Create" is not defined.)
- **WS-ServiceGroup** describes how collections of Web services and/or WS-Resources can be represented and managed.
- **WS-BaseFaults** defines a standard exception reporting format.

WSRF specifications are compliant with the WS-Interoperability Basic Profile [11], meaning that any WS-I-compliant Web services client can interact with any service that supports WSRF specifications. From the client's perspective, WSRF defines conventions for the message exchanges used to interact with state -- the goal of which is to make services that follow these conventions easier to use and manage.

A WSRF implementation typically also implements at least some functionality defined in the three "WS-Notification" (WSN) specifications (also separately being standardized in OASIS [12]): WS-BaseNotification, WS-BrokeredNotification, and WS-Topics. In WS-BaseNotification, Notification Consumers send subscribe messages to Notification Producers to request asynchronous delivery of messages. A subscribe request may contain a set of filters that restrict which notification messages are delivered. The most common filter specifies a message topic using one of the topic expression dialects defined in WS-Topics (e.g., topic names can be specified with simple strings, hierarchical topic trees, or wildcard expressions). Additional filters can be used to examine message content as well as the contents of the Notification Producer's current Resource Properties. Each subscription is managed by a Subscription Manager Service (which may be the same as the Notification Producer). Clients can request an initial lifetime for subscriptions, and the Subscription Manager Service is used to control subscription lifetime thereafter. When a client wishes to unsubscribe, they delete their subscription through the Subscription Manager service. When a Notification Producer generates a message, it will send that message wrapped in a <Notify> element (though unwrapped "raw" delivery is also possible) to all subscribers whose filters evaluate to "true". WS-BrokeredNotification provides for intermediaries between Notification Producers and Notification Consumers. These intermediaries receive messages from Notification Producers and broadcast them to their own set of subscribers, allowing for architectures in which Notification Producers do not want to or cannot know who is subscribed.

2.2 WS-Transfer and WS-Eventing

WS-Transfer [6] defines a mechanism for acquiring XML-based representations of entities using the Web service infrastructure. *"Resources"* in WS-Transfer are entities addressable by an endpoint reference that provide an XML representation and *"Resource factories"* are Web services that can create a new resource from an XML representation. WS-Transfer assumes best effort support for resources on the server -- for example, clients can reasonably expect that, after receiving an acknowledgement of a successful operation in response to a request to create a resource, the resource exists on the server although there is no guarantee that this is true. In particular, the specification notes that "the server may change the representation of a resource, may remove a resource entirely, or may bring back a resource that was deleted." WS-Transfer also relies on WS-Addressing [9].

WS-Transfer has only four operations (in the REST or CRUD pattern: "Create, Retrieve, Update, Delete"). *Get* fetches a one-time snapshot of the

representation of a resource. *Put* updated a resource by providing a replacement representation. This is not required to be the same XML representation as in the "Get"; in this case, the semantics of this operation are defined by the resource. *Delete* deletes the resource -- a successful *delete* operation invalidates the current representation associated with the targeted resource. *Create:* The resource factory that receives a Create request will allocate a new resource that is initialized from the presented representation.

WS-Eventing [7] is a notification subscription protocol that allows clients to request asynchronous delivery of event messages generated by Web services (called "event sources"). Standardized messages are defined to allow clients to subscribe to unsubscribe from these sources. Clients may optionally specify a filter predicate which will be evaluated against messages generated by the source. If the predicate evaluates to "true", the message will be sent to the client. These filters can be used to create topic-based subscriptions or to examine message content (e.g., with an XPath query) to see if it is relevant to the client. Subscriptions may also contain an expiration time after which messages will not be delivered. Finally, subscriptions may request a "delivery mode" specifying how the client would like to receive notification messages. These modes are viewed as an extension point by WS-Eventing in which application-specific ways of sending messages can be defined. Only a single delivery mode, "push", is defined by the specification to describe simple asynchronous messaging.

WS-Eventing also defines the concept of a subscription manager service. This service maintains information about the subscriptions themselves and allows subscriptions to be manipulated by clients. For example, the GetStatus message can be used to retrieve the expiration time of a subscription and the Renew message can be used to extend that time. The subscription manager service may be the same web service as the event source, or a separate service.

2.3 Discussion/Comparison of the Specifications

Neither set of specifications particularly address the need for a data model by which to represent state (e.g., no SQL Data Definition Language) and the need for a transaction model and a data manipulation language (DML). (For a specific example of a design that explicitly considers this, see CasJobs [13]--a multi-server, multi-queue batch job submission, execution, and tracking system for the Sloan Digital Sky Survey via Web services--and see [14] for the general issues regarding the impact of a Service Oriented Architecture on the treatment of data.)

There are a number of observations that can be made given our analysis of the specifications alone:

- The lack of "Create" in WSRF is problematic (WS-Transfer has a "Create"). In WSRF, every resource must come into existence via an application-specific protocol, causing interoperability issues.
- The lack of input/output schema for WS-Transfer is problematic. That is, every client must know the "type" of objects that the service understands; in WSRF, this is contained in the WSDL. In WS-Transfer, only an <XSD:any> tag exists, which means that the client must get this schema by other [nonstandard] means.
- WS-Transfer is a less complex specification than WSRF (in terms of the number and scope of functions defined), but the implications of this are not clear. For example, it could be that ease of implementing WS-Transfer (see Section 3) might eventually lead to more independent implementations. But it's not clear that two WS-Transfer implementations are more apt to facilitate interoperability. Although large specs with many moving parts often make it harder to get all of the parts working, a smaller set of spec-defined functionality creates its own problems. Specifically, with WS-Transfer (and WS-Eventing) an implementation is more apt to use functionality outside of the scope of the spec, causing interoperability headaches among custom extensions
- WSRF encourages each service to operate on a single "type" of resource (defined by the ResourceProperty document schema in the service's WSDL). WS-Transfer is silent on this issue, potentially allowing multiple types of resources to be associated with a single service. This may lead to a style of interaction in which resource "names" (e.g., EPRs) are no longer opaque to clients and are created by clients using service-specific rules to differentiate between resource types. While it can be argued that EPR opaqueness is an architectural principle of WS-Addressing (and therefore need not be addressed directly by WS-Transfer), the W3C WS-Addressing working group has argued that there may be legitimate reasons for breaking this principle and that it is difficult to prevent clients from adding ad-hoc information to resource identifiers. While WSRF may encourage resource names to be generated only by services, this may result in unintuitive client/service interactions

and/or additional messages. However, it may also prevent service implementation details from "leaking" through in the service-specific way clients are expected to construct these names.

- At first glance, the explicit mention of "best-effort" semantics in the WS-Transfer *Create* implies that WSRF may be "more robust", but this belief is probably artificial -- it's just that WS-Transfer is more upfront with its lack of guarantees.
- Although WS-Notification has been argued to be more complicated than WS-Eventing, we observe that much of this additional complexity is optional.

3 Implementing the Specifications

We first present a generic implementation architecture that illustrates the basic structure adopted in the resource-aware containers for both WSRF/WSN and WS-Transfer/WS-Eventing (Figure 1). Both systems were implemented on top of the .NET Framework.

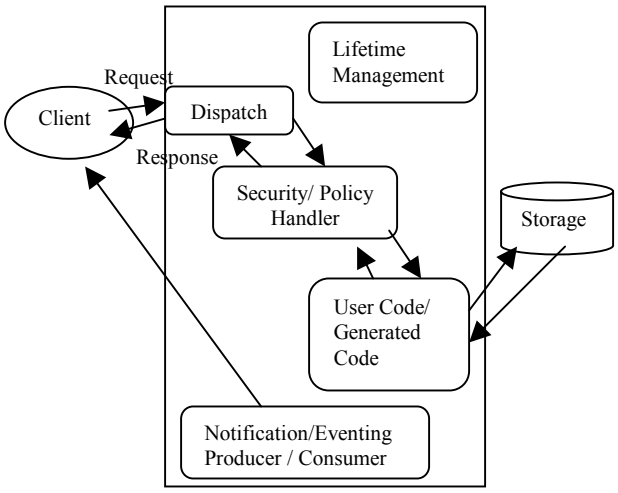

Figure 1. Generic Architecture for Resource-Aware Containers

The overview of processing is as follows. The outer box of Figure 4 is a "container" that is roughly Microsoft's application-hosting environment ASP.NET with additional functionality provided by our implementations. A request from a Client enters the container environment, where the Dispatch mechanism routes the request to the correct service. The Security/Policy Handler for the service examines the request and selects the protocol that will be used, authenticates the client, and if necessary creates a security context (e.g., session key). For message-level security protocols (WS-Security), the Security Handler verifies the signature on requests. This processing is provided by Microsoft's Web Services Enhancements (WSE) package. If security requirements are satisfied, the request is then forwarded to the service code. The state associated with the client is retrieved from "storage" for the invocation and placed back into storage once the request is satisfied. Once the service functionality is complete, the message generally passes back through the security handler, for example to digitally sign the response. The Lifetime Management component keeps track of the resources created by the client requests -- as will be described, this functionality is different in the two implementations. The Notification/Eventing Producer/Consumer component can be viewed as an independent activity within the container.

3.1 WS-ResourceFramework and WS-Notification: WSRF.NET

We have previously implemented the WSRF and WS-Notification on the .NET Framework. We describe only those features here that are relevant to the comparison. For more information, see [15][16][17]. Here we focus on the run-time processing, the programming model, and implementing WS-Notification.

Run-time Processing. Internally, WSRF.NET models Resources as XML documents that can be persisted to various backend stores. WSRF.NET contains built-in support for using an XML database, such as Microsoft's SQL Server 2005 ("Yukon"), SQL Server Express, or Xindice, as a backend, or an in-memory document collection backend. An interface to allow custom backends to be used (useful for legacy systems) is also provided. This model of Resources allows WSRF.NET to perform rich queries over that state of multiple resources using query languages such as XPath or XQuery.

The "user code/ generated code" in Figure 1 for WSRF.NET involves a "wrapper" that we wrote for use 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). Part of the functionality of the wrapper service includes the ability to automatically resolve the execution context specified by an EndpointReference, or EPR, (the embodiment of the WS-Resource Access Pattern supported by WSRF.NET).

Before the wrapper service begins execution of the appropriate method, the Resource specified by the EPR is loaded from the database and deserialized into appropriate data members of the web service class (see Programming Model section below). When the method invocation is complete, the wrapper service will

serialize the members' value back into a document that is then stored in the database. The result of the invocation is then serialized into a SOAP message by ASP.NET and returned via IIS to the client.

Although WSRF does not define how to create new WS-Resources, WSRF.NET provides a *Create()* library method that programmers can use to handle details of interaction with the storage backend. How the service exposes this functionality is up to the service author. One option is the direct exposure of this method in the Web Service interface. A second option is to instead expose some other method, which then invokes the *Create()* operation internally.

Programming Model. The goal of WSRF.NET is to make programming a WSRF.NET service as easy as programming any other Web service, with the added value of easier state management. WSRF.NET provides an attribute-based programming model that allows service authors to easily define both the stateful resources and the Resource Properties used by their services. This model also allows programmers to easily "import" functionality defined in the WSRF or WSN specifications. Consider the following code fragment.

```
[WSRFPortType
   (typeof(GetResourcePropertyPortType))]
public class MyService : ServiceSkeleton
{
    [Resource]
    int v;

    [ResourceProperty]
    public int DoubleValue
    {
      get { return v * 2; }
    }

    public MyService() { // constructor }

    // service's other methods, including
    //  the initialization of "v"
    [WebMethod]
    public int MyMethod() {
        // body
    }
}
```

The [Resource] attribute annotates class-level data members whose values should be persisted in the database as part of a WS-Resource. This means that a unique value of *"v"* will be loaded, based on the EPR in the request headers, for each method invocation. The method may use/manipulate this value as any other data member. When the invoked method completes, *v* will be saved back to the database. The [ResourceProperty] attribute annotates a C# Property whose "get" method will be called whenever a client uses one of the WS-ResourceProperty functions for retrieving resource property values (a similar "set" method can be defined for client invocations of the SetResourceProperties method). Note that the ResourceProperty value can be computed dynamically, using a portion of the WS-Resource state if required. Finally, the [WSRFPortType] attribute allows the service author to easily allow his service to support the WS-ResourceProperty defined method GetResourceProperty. All port types defined in all the WSRF and WSN specifications can be similarly imported, causing the importing service to export both their methods and their ResourceProperties. A tool called the PortTypeAggregator takes the user-defined service and creates the deployable service based on these attributes.

Implementing WS-Notification on .NET. Two issues in WS-Notification significantly impact the potential for interoperable implementations. First, the "raw" method delivery of a WS-Notification message is particularly problematic. The information passed with a notification (the parameters) is not well-defined, thus making interoperable raw message delivery challenging. Second, the lack of a standardized "create" method will result in idiosyncratic interfaces, particularly for the SubscriptionManager and PublisherRegistrationManager port types. These two port types store, manipulate, and reference subscriptions that consumers have made to producers, and registrations of publishers to brokers respectively. All notification producers and brokers must be implemented with a specific, non-standard way of creating and retrieving subscriptions and registrations.

Implementing WS-Notification reinforced our belief that WS-Notification, arguably, is very complex. For example, in demand-based publishing, 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 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 six separate Web services. More messages are generated in response to a demand based publisher scenario then 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.

3.2 WS-Transfer and WS-Eventing

Implementing WS-Transfer on .NET. Because there are only four operations in WS-Transfer, we implemented WS-Transfer very quickly and with relatively few lines of code. To facilitate a fair comparison, the people who implemented WS-Transfer for this project were *not* the same people who implemented WSRF.NET. Because of the lack of time, we have not yet developed a programming model based on WS-Transfer -- instead, the current implementation relies on the service author and client author directly manipulating XML documents. We factor out this limitation from any comparison of the two implementations whenever possible.

The fundamental issue we encountered was how to interpret persistence requirements from the WS-Transfer specification; we chose to use an XML database, Xindice, which is also supported in WSRF.NET. Our WS-Transfer implementation is basically the same as the Figure 1, except for there is no lifetime management functionality since it is not defined in the spec. The *Create()* operation creates a new XML-based resource representation based on the parameter received from the client. The service may or may not modify the XML-based resource representation (parameter) sent by the client, and stores the XML document into a Xindice database. In the service, the *Create()* operation names the resource by assigning a new resource id (by default, GUID) to it. Subsequent client references use this name, which is embedded into a returning EPR as a reference property. Together with the EPR of the new resource, *Create()* returns a new resource representation to the client if the resource representation is modified from the user's input. The *Get()* operation retrieves an XML document, which is identified by the resource id (i.e., reference property of SOAP header), from Xindice. Depending on the semantic of *Get()*, it may run query on database or pull out an overall document identified by resource id. The *Put()* operation identifies the XML document in Xindice by the resource id, and updates the document according to the semantics of the service. The XML document in Xindice is removed when the client calls *Delete()*.

We encountered three additional issues when implementing WS-Transfer. First, for simplicity, by default, the resource and its representation are equivalent -- there is no generic mechanism to attempt to hide/abstract the resource from the client. However if the resource is some kind of active entity such as running process or data transfer, we distinguished the notion of resource from its representation. The representation of the resource may remain even when the resource (e.g., process) does not exist anymore. The behavior of *Delete()* was not clear: do we terminate the process when the client calls a *Delete()* operation or delete only the XML representation from the database? WS-Transfer spec does not explicitly distinguish between these two notions while WSRF differentiates resource properties from resource.

Second, while the operation is required, WS-Transfer does not mandate that *Create()* is the *only* way to create a new resource. There is a possibility that a resource is created by an out of band mechanism. It can still be identified by EPR in *Get(), Set(), and Delete()*. Our service-side implementation had to be a little more sophisticated to deal with legitimate operations on resources (e.g., Get()) for which a corresponding *Create()* had not been previously issued to the service (e.g., the *Get()* is legitimate, although the corresponding entry in Xindice is not added by calling *Create()*).

Third, implementing WS-Transfer reinforced the perceived problems resulting from WS-Transfer lacking schema definition for input and output of operations. Our prototyping of services/clients based on our WS-Transfer implementation relied on hard-coding of common schemas within the client and service. We determined no elegant mechanism by which the client could easily discover the schemas (although emerging specifications like WS-MetadataExchange do seem promising). Clearly, this hinders independent development of the client and service in the long term.

WS-Eventing on .NET. We did not implement WS-Eventing; instead, we utilized the implementation of WS-Eventing by Plumbwork Orange [18]. The most important features in the implementation are the Subscription Manager Service, Event Source Service, and Filtering facility, all of which are defined in WS-Eventing. Additionally the implementation includes Notification Manager, which can be used to trigger a notification to subscribers. Subscribers can subscribe for an event by calling *Subscribe()* operation of Event Source Service. It may specify a filter which can be used for a topic-based notification. Unlike WS-Notification, a subscription is not associated with a resource, but only with a service. Thus, a filter can be used for registering a subscription per resource. The Subscription Manager Service implements operations including *Unsubscribe(), GetStatus(), and Renew()*. It maintains the subscription lists in a flat XML file. The Notification Manager, which is not defined in the spec,

is a convenient tool for an event source to trigger notifications by using operations implemented in it.

3.3 Discussion/Comparison of the Implementations of the Specifications

Having implemented both sets of specifications (and used an implementation of WS-Eventing), we make additional observations:

- Arguably, both approaches rely on efficient storage of XML-based resources, so it is not surprising that the same XML database (Xindice) was used. In some cases this may be overkill and a standard database or even in-memory might make more sense.
- WS-Transfer was easier to implement than WSRF. This is partly because little time was spent on a programming model or optimization for WS-Transfer. Even if you were to develop a similar programming model for each implementation, the WS-Transfer implementation would likely require less effort, because of WSRF's larger feature set.
- Based on our experience with WS-Notification, we think that most implementations are not going to implement all of the specification. Even if they did, there might be only partial interoperability with anyone else's, as we believe that that the more complicated a specification is, the more difficult interoperability becomes.

4 Qualitative and Quantitative Comparison

We now utilize the implementations of the two sets of specifications in both a simple service and a more complex multi-service scenario.

4.1 "Hello World"

A key determinant in the utility of each approach is the effectiveness, ease-of-use, etc. of the simplest application that (arguably) needs the functionality of the specifications, namely the manipulation of state and/or asynchronous notifications. The "hello world" application that we designed and implemented to evaluate each approach is the "counter service" that keeps track of some integer counter. This service optionally delivers an asynchronous notification to a consumer when the value of the counter is changed (for example, when another client updates the counter's value). From the viewpoint of a distributed object system, we believe this is the simplest case of when a client might want to instantiate an object on the server.

We do not show the code for the service in each implementation because we believe that this would be an unfair comparison and generally misleading. As discussed in the previous section, WSRF.NET has benefited from a longer development time, in which we have created higher-level, programming-level abstractions, which are not present in the WS-Transfer implementation.

4.1.1 "Hello World" via WSRF.NET

Fundamentally, the "resource" is simply a single variable, as the service class defines one data member “cv” (for "counter value"), which WSRF.NET gets/stores whenever a client message arrives with an EPR that refers to the specific WS-Resource in question. The service author has only had to define a single WebMethod, *create*, as part of this service, inheriting all other WS-Resource behavior (for getting and setting the counter value and for destroying a resource) from the WSRF.NET base libraries. The author-defined Create uses the WSRF.NET function ServiceBase.Create() to place a new resource in the backing store (Xindice in this case). For the counter service, this means storing *cv* with an initial value of 0.

4.1.2 "Hello World" via WS-Transfer/WS-Eventing

The design challenge was to map (or *not* map, as the case may be) the operations on a counter into the 4 operations of WS-Transfer: *Create()*, *Get()*, *Put()*, and *Delete()*. The functionality of these operations mostly overlaps the four WSRF operations used in hello world, *Create(),GetResourceProperty(), SetResourceProperty(), and Destroy()*. *Create()* stores this XML document without modification into Xindice. When it is called, *Get()* retrieves the XML document and returns the document without any manipulation. The client expects the schema of the return value from *Get()* to be the same as the document given to *Create()*. *Put()* updates the corresponding XML document in Xindice with newly received value. Finally, *Delete()* remove the XML document from Xindice.

4.1.3 Evaluation

We now compare the performance of each service for each of its operations: Get: The duration to retrieve the current value of the counter; Set: The duration to set the value of the counter; Create: The duration to create a new counter as a resource; Destroy: The duration to delete the resource corresponding to a counter; Notify: A client first subscribes to the “CounterValueChanged” event for a particular counter. Then, we measure the duration to first set the value of the counter and then receive a message indicating that the counter value has changed. We ran each of the five tests in six scenarios:

1. No security, client and service on same machine

2. X.509-based signing of request and response, client and service on same machine
3. https, client and service on same machine
4. No security, client and service on different machines
5. X.509-based signing of request and response, client and service on different machines
6. https, client and service on different machines

We used two identically-configured machines: Dual AMD Opteron 240 - 1.4GHz w/1MB L2 Cache, 2GB (4x512MB) PC2700 DDR333 Reg. ECC, 1x Seagate 120GB EIDE 7200 RPM, Windows Server 2003. Figures 2 through 4 present the results in pairs, with each pair comprising a particular non-distributed scenario along with its distributed counterpart. All numbers are in milliseconds for a single request.

Manipulating simple counter resources should be straightforward, and for the most part, this is true of both counter service implementations. Counter interactions (create, get, set, delete) map cleanly to both the WSRF ResourceProperty operations and the WS-Transfer operations. Both the WS-Notification and WS-Eventing specifications and implementations seem to support simple subscriptions and asynchronous notifications equally well for our counter service. Although in the WSRF.NET implementation of the counter service, many of the more sophisticated aspects of the WSRF and WS-Notification specifications (resource property querying, resource lifetime management, service groups, brokered notification, etc.) go unused, the service implementation does not seem complicated by the toolkit's support for these extras. Effectively with WSRF it does seems possible to "buy only what you need." Clearly the WS-Transfer version of the counter makes use of all the features of the WS-Transfer specification.

From a client perspective, engaging either counter service is similar to invoking web methods on any other Web service -- in .NET, via a Web service proxy object with methods corresponding to those on the service. The methods for get, set, etc. are fairly similar for both WSRF and WS-Transfer. Probably the biggest difference is the arguments to these methods. Since WS-Transfer deals in terms of raw XML, the arguments and return values for the WS-Transfer proxy methods are arrays of XML elements. Since WSRF does define the schemas for its method parameters, the WSRF.NET proxies are able to automatically deserialize the XML into C# run-time objects. In both toolkits, however, the representation of resources or resource properties (for instance returned from a get operation), must be manually deserialized into application specific types. In terms of subscribing to and receiving notifications in a simple counter client, this too turns out to be fairly straightforward. Subscribing is just a method invocation in both cases.

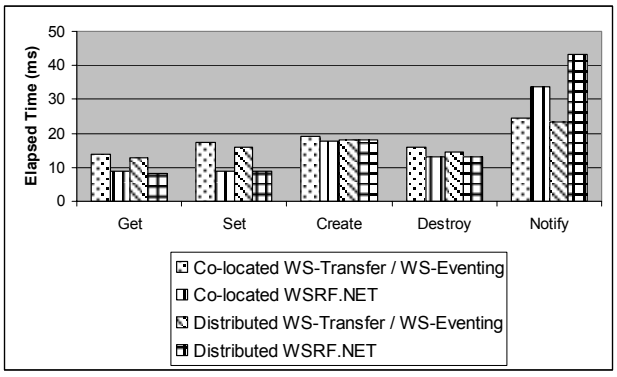

Figure 2: Testing "Hello World" with no security

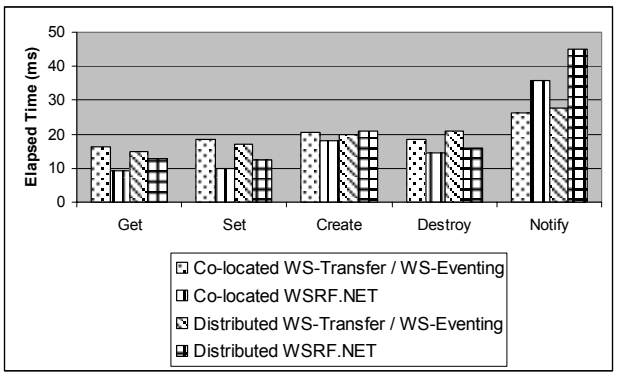

Figure 3: Testing "Hello World" over HTTPS

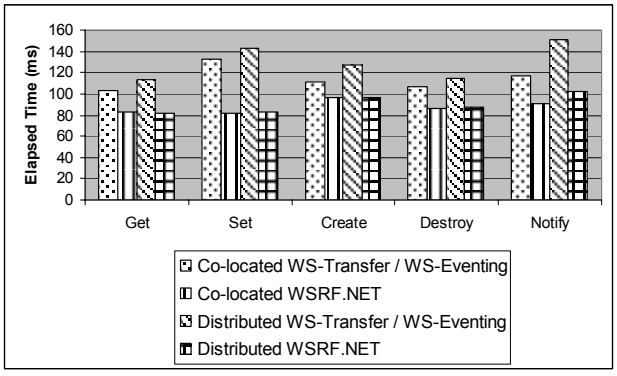

Figure 4: Testing "Hello World" with X.509 Signing

The tools for receiving notifications are slightly different, WSRF.NET uses a custom HTTP server that clients include, Plumbwork Orange uses a WSE SoapReceiver to handle notifications via TCP. Including this functionality in client applications is simple and straightforward for both toolkits.

Comparing the performance of the two implementations, we see that they perform similarly in most cases. While WSRF.NET may tend to appear somewhat faster, we attribute these differences to the more extensive optimization effort (particularly write-through resource caching) that has been invested. The

WS-Transfer implementation is largely unoptimized. Both counter implementations' performance is dominated by Xindice. Creating resources (and adding them to the database) in particular is always slower than reading or updating them. In the case of the WS-Transfer implementation of the counter service, setting the counter's value, causes the old representation of the counter's resource to be read from the database and updated with the new value before being stored. The WSRF.NET implementation through use of its resource cache is able to avoid this extra database read and thus performs faster for set operations.

Notification performance does appear to be considerably better for the WS-Eventing implementation than for WSRF.NET because of the TCP vs. HTTP issue.

Adding security to the counter example does little to alter the overall performance trends we have already seen. If anything, it makes percentage wise differences in performance between the two implementations even less notable. This is especially true of the X.509 signing tests, where the overhead of the security processing is so large that the performance differences between the two underlying systems tend to fade in significance. Due to socket caching, HTTPS performance is much faster.

4.2 "Grid-in-a-Box"

Grid-in-a-box consists of a set of Web services that provide remote job execution capabilities in a grid environment. The services are inspired by the OMII 1.0 services [19] although they do not literally match their interfaces or precise functionality. As shown in Figure 5, a deployment of grid-in-a-box represents a single virtual organization (VO) and consists of five types of services:

- ***AccountService****:* maintains a mapping between user identity and VO privileges.
- ***ResourceAllocationService***: takes client requests for particular applications and returns lists of available hosts/services in concert with the ReservationService.
- ***ReservationService***: holds and manipulates reservations for ExecServices and DataServices.
- ***DataService***: co-located with an ExecService and provides the ExecService with any data needed for user jobs running on the ExecService's host.
- ***ExecService:*** Clients can start and stop jobs and either poll for or subscribe to receive asynchronous notifications of job status.

Typically, there will be one AccountService, ResourceAllocationService and ReservationService for the entire VO and one ExecService and DataService for each machine in the VO. To conserve space in this paper, we omit the details of each service. For more, see [16], which describes an earlier version of the WSRF.NET-based implementation.

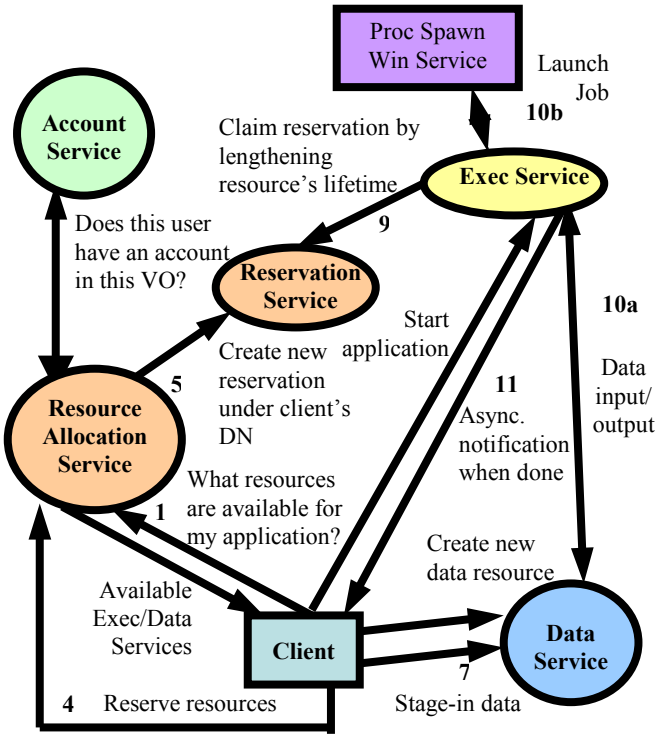

Figure 5. Grid-in-a-Box Services

4.2.1 Grid-in-a-Box via WSRF.NET

The critical issue for both implementations was the determination of the resources themselves and how they would be managed. For the DataServices, WS-Resources are directories. Clients create new directory resources (although do not *name* them), upload data to them, and pass the EPRs for these WS-Resources to the ExecService when starting a new job. The ExecService uses the associated directory as the working directory for the new job. For the ReservationService, WS-Resources are the reservations themselves. When a client creates a reservation, they receive an EPR representing that reservation which is also passed as one of the parameters to the ExecService when starting a job. An ExecService uses the reservation EPR to verify that the client has, in fact, reserved that ExecService. WS-Resources for the ExecService, then, represent jobs. When a new job is started, an EPR is returned to the client. The client can use this EPR to poll for the job's status, kill the job, etc.

The WS-Resources in the Account and ResourceAllocationServices are less straight-forward.

Although a system could be architected in which "accounts" and "available resources" were the WS-Resources for these services, and information about individual accounts, or available resources was accessed by EPR, such functionality is not used by Grid-in-a-box as described above. All interaction with these services uses the same state information (the mapping of users to privileges or the mapping of installed applications to ExecServices) and so the WS-Resource concept is not utilized.

Each of the ExecService, DataService and ReservationService use WSRF's Resource Properties to express information about their resources. The ExecService's job resources provide Resource Properties for process status, e.g. whether the job is currently running, how long it has been running, when it exited and the exit code. The DataService resources use Resource Properties to expose the files contained within each directory resource. This can be used to survey a job's output. The ReservationService uses Resource Properties to expose the details of a particular reservation that are relevant to the ExecService, i.e. which hosts are part of the reservation and the reservation's owner. WSRF's Resource Lifetime functions are also used for lifetime management of resources. For the ExecService, WSRF's Destroy method will kill a job if it is running and then cleanup the information about the process' exit state. The DataService uses the Destroy method to remove a directory and its contents from the remote filesystem. The ReservationService uses Scheduled Termination of reservations. When a client initially makes a reservation, the termination time of the reservation resource is set to the current time plus an administrator specified delta (e.g. 4 hours). When a job begins executing on a host, the ExecService on that host will send a message to the ReservationService to "claim" that reservation. Receiving this "claim" causes the ReservationService to lengthen the time until the reservation terminates. While the current Grid-in-a-box sets the termination time to infinity (i.e. never terminate this reservation, only allow it to be explicitly destroyed when jobs on associated hosts have completed), this mechanism could be used to give particular jobs/users a limited amount of run time. In that case, the ReservationService would likely send out a notification message to the ExecService when a reservation terminates, so that the ExecService may stop the user's job.

WS-Notification is used to asynchronously notify subscribed clients when their jobs exit. This notification message will contain the job's EPR so that the client knows which of the potentially many jobs they are currently running, has ended. The client may then examine the DataService resource associated with that job to retrieve output, and then cleanup both ExecService and DataService resources using the Destroy method.

4.2.2 Grid-in-a-Box via WS-Transfer/WS-Eventing

We have built a WS-Transfer/WS-Eventing compliant implementation of Grid-in-a-Box. There are four services (Account, Data, Resource Allocation/Reservation and Execution) and two clients (grid user and admin client). Our services use a Xindice XML database to store the resources. The only exception is the Data Service that stores the files on the file system. An explicit design decision was to attempt to map onto the CRUD operations as much as possible.

Due to the relative simplicity of the account service the mapping of its functionality to the corresponding WS-Transfer operations is very intuitive. Our WS-Transfer *Create()* operation creates a new account and the new account is stored as a resource, with the EPR containing the X509 DN of the user. *Get()* queries the account service whether a particular user can perform a certain action. *Delete()* removes all the privileges of a particular user. *Create()* and *Delete()* are administrative functions and can be called only from the administrative client, and *Get()* can be called from any other service.

Data service deals with file transfer to and from a computing site. A WS-Transfer *Create()* operation is invoked whenever a user wants to upload a file. The EPR of the resource (file) is in the format user's DN/filename. All the files of a particular user are stored into the same directory, so if a directory for this user does not exists yet it is created automatically. The directory created is a hash of the user DN. *Put()* overrides an existing file with a newer version. *Delete()* removes a file permanently from the file system of the server. Our utilization of the WS-Transfer *Get()* operation returns a different type of resource depending on the type of the EPR requested. If the EPR ends with "/", the *Get()* operation returns a listing of all the files in the directory specified. Otherwise *Get()* interprets the request as a download for the file specified.

As WS-Transfer allows different types of resources to coexist in the same service, we have a unified Resource Allocation service -- compared to Resource allocation service and a Reservation service in WSRF Grid-in-a-Box. (WS-Transfer is more flexible with the number of different types of resources a service can store. It's not clear how valuable this extra flexibility is.) Our WS-Transfer *Create()* operation creates the representation of a new computing site. *Delete()* permanently removes a

computing site from the database. Depending on the initial character of the EPR the WS-Transfer *Get()* operation does different things. If the EPR starts with "1", the get is interpreted as a get available resources query and all of the unreserved resources with the requested application name are returned. This mode is needed by the grid client. Otherwise, the *Get()* is a request to check which user has a reservation to a particular computing site and a user DN is returned. This mode is used by the Data service and the Execution service to make sure that the user who wants to use them has a reservation. The WS-Transfer *Put()* operation has 3 modes of operation depending on the initial symbol of the EPR. They are used to make a reservation, remove a reservation or change the time to which a site is reserved.

4.2.3 Evaluation

The Grid-in-a-Box services are considerably more complicated than the Hello World counter service, and as a result the Grid-in-a-Box implementations built on each software stack vary more significantly, both in design and performance (Figure 6). Some of these design decisions were motivated by the underlying specifications, some by the software toolkits used and some are best classified as differences of opinion. Although for fairness of comparison we attempted to eliminate those in the last category, it is often difficult to discern the precise motivations behind designs and whether certain decisions are truly idiosyncratic or in fact have their origins in the specifications and toolkits we are examining here. The net result of this is that each Grid-in-a-Box implementation retains something of a unique character, on purpose.

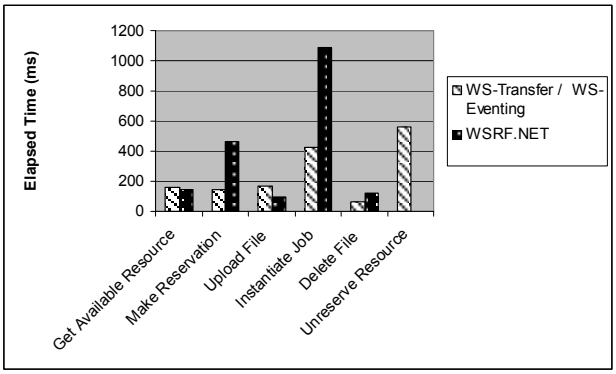

Figure 6: Grid-in-a-Box Performance Comparison

Inspection of the performance results (Figure 6) reveals more about the designs of the two Grid-in-a-Box implementations than the toolkits and specifications used. The greatest factor influencing the performance of individual operations is the number of web service outcalls (and message signings) triggered on the server. The Delete File operation involves a single call in both implementations while Upload File requires a pair of calls in both. Unsurprisingly, the results of these operations are comparable. In contrast, due to the design of its services the WSRF implementation requires several more outcalls to Instantiate a Job than the WS-Transfer version. Un-reserving a resource also happens automatically in the WSRF version (so no time is reported). The performance differences between individual spec-defined operations (Section 4.1) are small enough, that the overall design of a system dictates how fast it will run.

Among the largest differences between the two implementations concerns how much can be modeled as resources and interactions with these resources. The WS-Transfer version is entirely resource driven; everything from accounts to files are presented as resources and all interactions with the Grid-in-a-Box services map to one of the Create, Retrieve, Update, Delete (CRUD) set of operations. The WSRF version of these services takes a slightly different approach as accounts and available resources are not modeled as WS-Resources. Additionally, interactions with the Account and ResourceAllocation services are not mapped to the CRUD operations (instead opting for operations like addAccount, accountExists, etc.). There is an appealing simplicity to having a small subset of operations (four in the case of WS-Transfer) in your system. The trade-off of course is that a single operation may have to accomplish different behaviors (depending on their arguments) and the meaning of individual methods becomes less clear. It's the difference between invoking an accountExists() method to determine if a user has an account and invoking a get operation with the right XML header content so that the account is checked and a yes/no answer is returned. A client constructed in terms of meaningful application specific methods (like accountExists) is likely to be easier to understand than one that only uses CRUD operations. Still, it is worth noting that neither the WSRF nor WS-Transfer specifications preclude you from having regular Web service methods outside of the resource framework they define.

An interesting related issue involves how collections of resources can be modeled. Consider the basic Data Service which manages directories and files (on a per user basis) and their related properties. The WSRF.NET version of the Data Service models directories as resources and individual files (or more accurately the list of files) are properties of that resource. No information for individual files is actually stored as resources, instead these resource properties are generated dynamically by examining the contents

directory in response to a GetResourceProperty() call. WS-Transfer lacks this resource vs. resource property distinction. The WS-Transfer Data Service effectively exposes two types of resources: directories and files, a directory resource being a collection of file resources. Content in the WS-Addressing EPR is used to determine which type of resource is referred to in a specific Get request.

Another distinguishing aspect of the alternative implementations involves the Resource Allocation/Reservation service. In the WS-Transfer implementation, both the collections of reservable CPU/data resources and individual reservations are handled by a single Resource Allocation Service. In contrast, the WSRF.NET implementation splits this functionality into two services, one to manage each type of resource (one type of resource per service is a WSRF requirement). The WSRF.NET implementation makes use of the ResourceLifetime specification to determine when reservations should expire and be deleted. Since WS-Transfer lacks such concepts, reservation lifetimes must be managed manually. A failure to destroy a reservation after a job is finished would prevent the subsequent use of that execution resource.

5 Discussion

Perhaps the most important question that this paper is attempting to address is: *I'm starting from scratch; which alternative (spec and/or implementation) do I choose?* To address this, we must answer a number of related questions:

- *Is one spec/implementation faster?* No. The performance numbers for WSRF.NET and WS-Transfer/WS-Eventing are comparable (and actually dominated by X509 processing). We believe this is not idiosyncratic to .NET -- we assert that other implementations will experience the same behavior.
- *Is one {spec, implementation} easier to program services? Clients?* In general, no, as both are affected by the resource vs. representation issue. Client programming is very similar for both stacks (inherited from .NET and WSE). Ultimately, the programming model -- which is independent of specification -- probably has a greater influence on programming ease.
- *Should I ignore both specs/implementations when I build something like Grid-in-a-Box?* Arguably we saw that mapping everything to CRUD might be a little awkward. While not quantifiable, we saw in the WSRF.NET case that using the WSRF.NET programming model and the CRUD verbs where they fit naturally made things a little easier.
- *If one is "more full-featured" than another*, *are the extra features useful? Will the alternative stack be likely to implement these as extensions or is it more likely that such extra features will be ignored?* WSRF does have additional functionality WS-Transfer lacks (brokered notification, service groups, lifetime management, resource property queries, etc.) The utility of these features is an open question. The WSRF Grid-in-a-Box implementation only made use of one of these extra features (lifetime management), suggesting perhaps limited utility. On the other hand, implementing and/or using one complete stack or the other is not necessary, using only a small subset of features is perfectly acceptable.
- *How easy is it to switch from one stack to the other? Suppose that I have built a system (clients/services) based on stack A. My system is running just fine, and then B becomes the clear favorite of the community (i.e., my implementation of A's specs won't be around forever). I now need to add a new service using B or write a new client to consume a service written in B. Does my choosing one stack now better prepare me if this scenario happens in the future?* Switching from WS-Transfer/WS-Eventing to WSRF/WS-Notification is likely easier, as applications built using the additional functionality in WSRF would have to re-invent these extras if switching to WS-Transfer. From a client perspective that since both stacks are WS-I+ compliant, it should be possible to build client proxies with commercial tools right now (although an existing WSRF-speaking client *cannot* simply be aimed at the "corresponding" WS-Transfer-based services). Both suffer from the need to add the correct WS-Addressing header content.

6 Conclusion

Could the Open Grid Services Architecture (OGSA) have multiple stacks? While we acknowledge that there are multiple ways in which to interpret the results presented in this paper, we believe both the WSRF-based software stack *and* the WS-Transfer-based software stack have positives and negatives and thus neither should be excluded. While some argue that standardizing one approach would make things simpler and less confusing for users, especially since they both provide the same basic feature sets, others argue that choice is good and in fact core to open source development (e.g., see the Linux "Gnome vs. KDE" argument). The benefits of having independent

alternatives with slight feature differences (that perhaps cater to slightly different user bases) and that can innovate independently outweigh what would be gained by having one winner. As OGSA above all should be *inclusive* in order to be successful, we believe that based on this analysis there could be alternative software stacks for OGSA-based Grids.

Acknowledgements

We are thankful to Savas Parastatidis for helping us clarify the presentation of some of the concepts in this work.

References

[1] 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.

[2] I. Foster and C. Kesselman. "Globus: A Metacomputing Infrastructure Toolkit." *International Journal of Supercomputer Applications*. Vol. 11, Issue 4, 1997.

[3] I. Foster, C. Kesselman, J. Nick, S. Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6), 2002. *Computer*, 35(6), 2002.

[4] I. Foster, K. Czajkowski, D. Ferguson, J. Frey, S. Graham, D. Snelling, S. Tuecke., Modeling and Managing State in Distributed Systems: The Role of OGSI and WSRF, Proceedings of the IEEE, 93(3), 2005.

[5] S. Graham, P. Niblett, D. Chappell, A. Lewis, N. Nagaratnam, J. Parikh, S. Patil, S. Samdarshi, I. Seduhkhin, D. Snelling, S. Tuecke, W. Vanbenepe, and B. Weihl. Publish-Subscribe Notification for Web services. 03/05/2004. http://www-106.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf

[6] J. Alexander, D. Box, L. Cabrera, D. Chappell,G. Daniels, A. Geller, R. Janecek, C. Kaler, B. Lovering, D. Orchard, J. Schlimmer, I. Sedukhin, J. Shewchuk. Web Service Transfer (WS-Transfer). September 2004. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-transfer.pdf

[7] D. Box *et. al.* Web Services Eventing (WS-Eventing). August 2004. http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf

[8] S. Parastatidis, J. Webber, P. Watson, T. Rischbeck. WS-GAF: a framework for building Grid applications using Web Services. Concurrency and Computation: Practice and Experience. Vol. 17, issue 2-4. p 391-417.

[9] IBM, BEA, and Microsoft. WS-Addressing. 2004. http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-addressing.asp

[10] OASIS Web Services Resource Framework (WSRF) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf

[11] K. Ballinger, D. Ehnebuske, M Gudgin, M. Nottingham, and P. Yendluri, eds. Web Services Interoperability Organization (WS-I) Basic Profile Version 1.0. Final Material. 2004/04/16. http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

[12] OASIS Web Services Notification (WSN) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn

[13] W. O'Mullane, N. Li, M. A. Nieto-Santisteban, A. Thakar, A. S. Szalay, and J. Gray. "Batch is back: CasJobs, serving multi-TB data on the Web". In *Proceedings of the 2005 International Conference on Web Services (ICWS 2005)*, Jul 11-15, 2005. Orlando, FL.

[14] P. Helland, "Data on the Outside Versus Data on the Inside", In *Proceedings of the 2005 CIDR Conference*, Jan 4-7, 2005. Asilomar, CA. pp. 144-154.

[15] M. Humphrey, G. Wasson, M. Morgan, and N. Beekwilder. An Early Evaluation of WSRF and WS-Notification via WSRF.NET. *2004 Grid Computing Workshop (associated with Supercomputing 2004)*. Nov 8 2004, Pittsburgh, PA.

[16] G. Wasson and M. Humphrey. Exploiting WSRF and WSRF.NET for Remote Job Execution in Grid Environments. *2005 International Parallel and Distributed Processing Symposium (IPDPS 2005),* Denver CO, April 4-8, 2005.

[17] M. Humphrey, G. Wasson, K. Jackson, J. Boverhof, M. Rodriguez, J. Gawor, S. Lang, I. Foster, S. Meder, S. Pickles, and M. McKeown. State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations. *14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14),* Research Triangle Park, NC, 24-27 July 2005.

[18] Project Plumbwork Orange. http://sourceforge.net/projects/plumbworkorange/

[19] Open Middleware Infrastructure Institute (OMII). http://www.omii.ac.uk/