14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14), Research Triangle Park, NC, 24-27 July 2005.

State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations

Marty Humphrey, Glenn Wasson
Department of Computer Science, University of Virginia, Charlottesville, VA USA

Jarek Gawor, Joe Bester, Sam Lang, Ian Foster
Mathematics & Computer Science Division, Argonne National Laboratory, Argonne IL USA

Stephen Pickles, Mark Mc Keown
Manchester Computing, University of Manchester, Oxford Road, Manchester UK

Keith Jackson, Joshua Boverhof, Matt Rodriguez
Lawrence Berkeley National Laboratory, Berkeley, CA USA

Sam Meder
Computation Institute, University of Chicago, Chicago, IL, USA

Abstract

The Web Services Resource Framework defines conventions for managing state in distributed systems based on Web services, and WS-Notification defines topic-based publish/subscribe mechanisms. We analyze five independent and quite different implementations of these specifications from the perspectives of architecture, functionality, standards compliance, performance, and interoperability. We identify both commonalities among the different systems (e.g., similar dispatching and SOAP processing mechanisms) and differences (e.g., security, programming models, and performance). Our results provide insights into effective implementation approaches. Our results may also provide application developers, system architects, and deployers with guidance in identifying the right implementation for their requirements and in determining how best to use that implementation and what to expect with regard to performance and interoperability.

1. Introduction

An airline reservation system, a CPU management system, and a workflow system all have in common that they provide their clients with access to some (typically abstracted) view of their internal "state." In varying ways, each such system allows its clients to refer to stateful entities (reservations, CPUs, jobs), to access those entities' properties, and (at least in the case of the reservation and workflow systems) to manage their lifetime. This commonality of purpose has motivated the Open Grid Services Architecture (OGSA [1]) to identify state modeling and management as a fundamental requirement for service-oriented architectures.

Such considerations have led to the development of four specifications known collectively as the Web Services Resource Framework (WSRF [2]), which define conventional interfaces and behaviors for representing, abstracting, and manipulating state in a Web services framework. Three related WS-Notification (WSN [3]) specifications define interfaces and behaviors that allow clients to subscribe to changes in state, thus providing for push-mode access to state components.

While final WSRF and WSN specifications were still being finalized at the time of writing within OASIS, the importance of these specifications has motivated multiple groups to develop implementations. The availability of these implementations offers the opportunity to gain insights into the merits of these specifications and different implementation approaches.

To this end, we report here on a study in which we compared and contrasted the following five implementations from the perspectives of architecture, functionality, performance, and standards compliance:

- **GT4-Java**, the Java Web Services Core of the Globus Toolkit v4 [4];
- **GT4-C**, the C Web Services Core of the Globus Toolkit v4 [4];

- **pyGridWare**, a Python WSRF implementation [5], which is also distributed with GT4 as its Python Web services Core ("GT4-Python");
- the Perl-based **WSRF::Lite** [6]; and
- **WSRF.NET**, an implementation of WSRF and WS-Notification on the .NET Framework [7].

These systems are developed by different teams and differ in terms of implementation language, programming model, and, in several regards, overall goals. Thus, we believe that they provide a good basis for studying general WSRF/WSN implementation approaches.

We find that, because WSRF/WSN is consistent with the recommendations of the WS-Interoperability Basic Profile, the five systems achieve a base level of interoperability with regard to XML, HTTP, SOAP, and WSDL. We also see significant commonalities with regard to dispatching and SOAP processing techniques. On the other hand, we see significant differences in security, programming models, and performance. We describe these differences and relate them to design goals and performance measured in a set of benchmark experiments.

We organize the rest of this paper as follows. In Sections 2 and 3, we introduce the specifications and describe basic WSRF/WSN implementation techniques, respectively. In Section 4, we compare and contrast the five systems. In Section 5, we present and discuss the results of our performance experiments and in Section 6 we discuss interoperability. We conclude in Section 7.

2. WSRF and WSN Background

The WSRF and WSN specifications were introduced in January 2004, building on experience gained with the Open Grid Services Infrastructure (OGSI [8]).

The WSRF specifications define 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 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** [11] defines how WS-Resources are described by XML "Resource Property" documents that can be queried and modified. A Resource Property document is a view or projection of the state of the WS-Resource, but is not equivalent to the state.
- **WS-ResourceLifetime** [12] defines mechanisms for both explicit destruction and implicit (lease-based) destruction of WS-Resources. (There is no defined creation mechanism.)
- **WS-ServiceGroup** [13] describes how collections of Web services and/or WS-Resources can be represented and managed.
- **WS-BaseFaults** [14] defines a standard exception reporting format.

The five WSRF specifications are compliant with the WS-Interoperability (WS-I) Basic Profile [15], meaning that any WS-I-compliant Web services client can interact with any service that supports WSRF specifications. From the client's perspective, WSRF simply defines conventions for the message exchanges used to interact with state, thus making services that follow these conventions easier to use and manage.

Notification is not part of WSRF, but several WSRF specifications reference notification in a generic manner. Thus, a WSRF implementation typically also implements at least some functionality defined in the three "WS-Notification" (WSN) specifications: WS-BaseNotification [16], the simplest form of notification possible; WS-BrokeredNotification [17], which allows for intermediaries and an extra level of abstraction between producers and consumers; and WS-Topics [18], a description of the types of topics that can be considered part of notification. WSN is also being standardized in OASIS [19].

We use a simple example to illustrate how the interfaces and behaviors defined in WSRF and WSN can be used to advantage when developing service-oriented architectures.

In this example, a "job factory" that supports requests to create computational tasks defines an interface via which each job is modeled as a WS-Resource. Creation of a job returns an EPR to a WS-Resource corresponding to the job's status; subsequent requests to monitor job status can then be handled via WS-ResourceProperties mechanisms, while job lifetime can be managed via WS-ResourceLifetime mechanisms.

In a different context, the same mechanisms might be used to manage resource reservations or data transfers. This uniform treatment of similar concepts in different contexts can simplify implementation of both clients and services.

3. Implementing WSRF and WSN

We present a canonical implementation architecture that illustrates the basic structure adopted in all five WSRF/WSN implementations. Figure 1 illustrates this architecture. The large box represents the **Service Hosting Environment**. This "WS-Resource-aware container" consists of one or more WSRF-compliant

user-supplied services, typically based in part on code that the WSRF/WSN implementation provides to facilitate service development and deployment. We introduce its various components by stepping through the stages involved in processing a client request.

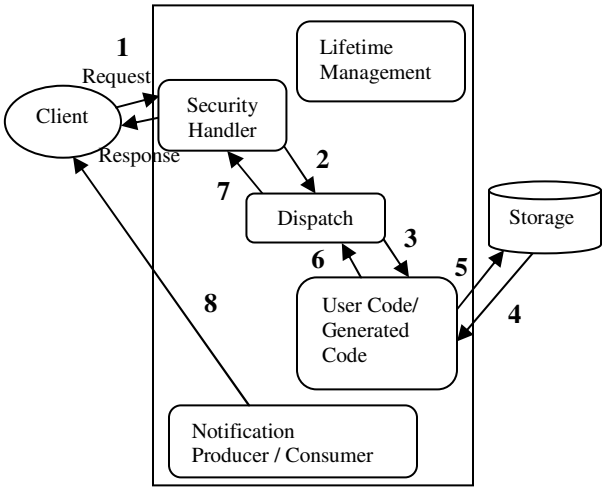

Figure 1. Generic service hosting environment architecture

A request from a WSRF-compliant **Client** generally enters the service hosting environment (1), where the **Security Handler** examines the request and selects the protocol that will be used, authenticates the client, and if necessary creates a security context. (Some implementations perform dispatching before all security processing is completed—e.g., to allow per-service authorizations.)

For message-level security protocols, the Security Handler verifies requests and signs responses. If security requirements are satisfied, the message is generally passed to a **Dispatch** mechanism (2) that routes the message to the correct WS-Resource (3). This WS-Resource is, again, a combination of "static" service functionality and WS-Resource-specific state. The associated state is typically retrieved from "storage" for the invocation (4) and placed back into storage once the request is satisfied (5). Once the service functionality is complete (6), the message generally passes back through the security handler (7), for example to digitally sign the response.

Table 1: Summary of key features of the five WSRF/WSN implementations

	GT4-Java	GT4-C	pyGridWare	WSRF::Lite	WSRF.NET
Languages supported	Java	C	Python	Perl	C# / C++ / VBasic / etc.
WS-Security password profile	Yes	No	In progress	In progress	Yes
WS-Security X.509 profile	Yes	In progress	Yes	In progress	Yes
WS-SecureConversation	Yes	No	Yes	No	Yes
TLS/SSL	Yes	Yes	Yes	Yes	Yes
Authorization	Multiple	Multiple	Callout	None	Callout
Persistence of WS-Resources	Yes	Yes	Yes	Yes	Yes
Memory Footprint	JVM + 10MB	22 KB	12 MB	12 MB	Depends on persistence mechanism
Works with unmodified hosting environment	Yes (Apache)	N/A	Yes (Twisted)	Yes (Apache)	Yes (ASP.NET)
Supports WS-I Basic Profile	Yes	Yes	Yes	In progress	Yes
Supports WS-I Basic Security Profile	Yes	Yes	Yes	No	Yes
Logging	Yes (Log4J)	Yes	Yes	Yes	Yes (WSE diagnostics)
WS-ResourceLifetime	Yes	Yes	Yes	Yes	Yes
WS-ResourceProperties	Yes	Yes	Yes	Yes	Yes
WS-ServiceGroup	Yes	Yes	Yes	Yes	Yes
WS-BaseFaults	Yes	Yes	Yes	Yes	Yes
WS-BaseNotification	Yes	Consumer	Yes	No	Yes
WS-BrokeredNotification	Partial	No	No	No	Yes
WS-Topics	Partial	Partial	Partial	No	Partial
CVS access	Yes	Yes	Yes	Read only	In progress
Bug tracking (e.g., bugzilla)	Yes	Yes	Yes	Yes	Yes

The **Lifetime Management** component keeps track of the WS-Resources created by the client requests. It monitors each WS-Resource Property and updates the Resource Property state following a set Resource Property request. This component is also responsible for cleaning up WS-Resources when their termination time has expired.

Similarly, the **Notification Producer/Consumer** can be viewed as an independent activity within the service hosting environment. The WS-BaseNotification component handles subscription requests to monitor a particular resource's state. When a resource changes to a state that matches a subscription request, a Notification response is returned to the client (8).

4. The Five Systems Described

We now describe the five systems, focusing in particular on notable differences in functionality provided and implementation approach. Table 1 summarizes the key features that we examine in this discussion.

4.1. Transport and SOAP Processing

Each system requires machinery for processing the HTTP protocol messages used to transport requests (and responses) and for deserializing (and serializing) the SOAP messages carried on that transport.

With the exception of GT4-C, each system uses existing code for this purpose: Apache Axis in the case of GT4-Java, Zolera SOAP Infrastructure (ZSI) for pyGridware, SOAP::Lite for WSRF::Lite, and Microsoft Internet Information Services (IIS) and the Web Services Enhancements (WSE) for WSRF.NET. GT4-C builds on libxml2 for XML parsing but otherwise provides its own implementation of SOAP and HTTP processing, via an HTTP driver that plugs into the XIO transport stack [23].

These different transport and SOAP processing systems differ in their performance and robustness, but all offer similar capabilities.

4.2. Security Issues

The Security Handler of Figure 1 can provide for *client authentication* and also for *message integrity* and *message privacy*. (We discuss authorization in Section 4.8 below.) These three functions can be provided at either the message level or the connection level, with different performance characteristics. (We examine performance in detail in Section 4.) Message-level security is mandated by the WS-I Basic Security Profile [20] and has advantages when routing individual messages; transport-level security can provide higher performance when many messages must be dispatched between a client and a server.

We encounter three security protocols in the systems studied here: two message-level protocols, SecureConversation and SecureMessage, and the TLS/SSL transport-level security protocol.

SecureConversation is based on the WS-Secure Conversation specification [21] and creates a security context using the X.509 proxy certificates of the client and container. The security context is used to sign and verify the body and necessary header elements in each of the messages. Depending on the implementation, creating a context requires an initial cost of at least three request/response messages.

SecureMessage does not create a security context and incur this cost; instead, necessary elements are signed in each method in conformance with the OASIS standard for SOAP Message Security [22] by using either the X.509 Token Profile or the Username Token Profile. SecureMessage is useful for single request/response interactions, as the client does not have the cost of creating the security context.

GT4-Java, GT4-C, pyGridWare, and WSRF.NET support all three security protocols. However, WSRF.NET's implementation of Secure Conversation will not interoperate with the other three systems' SecureConversation implementations because WSRF.NET inherits its SecureConversation from WSE. While the SecureConversation spec defines message formats for the exchange of cryptographic data necessary to establish a secure session, it does not define a single algorithm for computing that data, and WSE and GT4/pyGridWare implement different algorithms. WSRF::Lite only supports transport-level security.

4.3. Web Service Dispatch and Container

Having received a SOAP request and performed security processing, the relevant operation must be identified and dispatched, with message contents passed in an appropriate manner. Then, any reply is constructed and returned for SOAP processing and transport.

Each system's WS-Resources exist inside a container process within which these operations take place. Each system has similar functionality in this layer. WSRF.NET uses ASP.NET, GT4-Java can use Tomcat/Axis, and pyGridWare uses Twisted. Some systems provide their own containers: GT4-Java and pyGridWare can run stand-alone, GT4-C has its own container, and WSRF::Lite provides two "Container"

scripts to host WS-Resources: a standard Container and a secure Container that uses SSL.

4.4. Persistence

Once created, a WS-Resource must persist between service invocations until the end of its lifetime, meaning that the "state" of a WS-Resource after one invocation should be the same as its state before the next. Persistence can be achieved by holding the resource in memory, writing it to disk, or storing it in a database. Each system provides an interface that allows service authors to use custom (or pre-existing) persistence mechanisms, but each system also provides a different default configuration.

GT4-Java, GT4-C, WSRF::Lite and pyGridware all persist WS-Resources in memory by default. This approach provides the best response-time performance but is the least fault-tolerant. These systems also come with modules that allow resources to be saved to disk, providing the ability to survive server failure at the cost of some performance.

WSRF.NET uses a database by default. This approach is slower than in-memory storage (although write-through caching makes it competitive), but provides fault-tolerance and access to powerful query/discover mechanisms that are not present in the file system approach.

4.5. Finding / Discovering WS-Resources

Most requests to a WS-Resource require access to the resource's state. The five systems differ in how resources are indexed and retrieved.

GT4-Java and pyGridWare implement a ResourceHome interface, which allows pluggable discovery mechanisms. This interface contains the find() method, which discovers resources based on a supplied key, such as the resource name.

WSRF.NET uses a database query to find requested resource(s). This query mechanism makes it easy for services to provide functions that access multiple resources, such as management functions. In other words, queries can not only lookup resources based on unique key values, but also based on the data contained within the resource, for example "find all WS-Resources owned by Bob."

GT4-C defines interfaces for manipulating resources and provides a default implementation of those interfaces. Alternative implementations can be provided that, for example, do not require that any resource and state information be maintained within the hosting environment.

WSRF::Lite provides a number of ways for a WS-Resource to find its state. In some cases, the state is implicitly in the services execution context, while in other cases the Container provides a key to the developer which he uses to find the state: for example, the key could be a database index.

4.6. Lifetime Management

Lifetime management for WS-Resources includes both resource creation and resource destruction. Creation involves adding a new resource to the resource storage system, while destruction involves removing resources in response to immediate destruction requests as well as lifetime expiration.

Several systems handle creation similarly, providing a create() method that a service can call to place a new WS-Resource in the service's chosen store. For example, GT4-C's create() method adds a new resource to whichever store is used by the service's ResourceHome, while WSRF.NET's create() adds the new resource state to a database. GT4-Java does not define a specific create() operation since creation behavior can be different among various WS-Resources. For example, a resource can be created by some out-of-band process instead of through an explicit create() operation.

Different implementations remove expired resources via different mechanisms. GT4-Java and pyGridware leverage timing mechanisms in their container environments to schedule periodic resource deletion tasks. This approach exploits the container's ability to manage memory footprint and not inspect resources that are not resident in memory. In GT4-Java, a user can also provide its own method of removing expired resources. WSRF.NET uses a separate Windows service (not Web service) to run through the database periodically, because ASP.NET does not easily handle this periodic scheduling. This approach can be computationally efficient because it can be performed with a database update, and so does not require the loading of resources into memory to check if they are expired.

WSRF::Lite deals with lifetime in different ways depending on how the service developer has chosen to implement the WS-Resource. In some cases, the WS-Resources rely on the timing mechanism of the underlying operating system. In other cases, for example when the state is stored in a file or database, the state is only garbage collected if someone tries to access a stale WS-Resource. A disadvantage of this approach is of course that resources that are never accessed are never discarded.

GT4-C uses the event-handling architecture built into the Globus Toolkit's C common libraries, which provide time-based event triggering and polling. Callbacks are used to manage the lifetime of a given resource.

4.7. Programming and Tooling

A WSRF/WSN implementation's programming model defines the interface seen by developers of services that implement WSRF/WSN interfaces. The primary issue here is how the corresponding Web service implementation is constructed, which involves both architecture and tooling. The five implementations differ greatly in their approaches.

The **GT4-Java** programming model decouples the Web service (business logic) from the resource (state). The Web service implementation is usually a plain, stateless Java object. A service can be composed from several independent "operation providers," thus enabling reuse of common Web service operations among services. For example, a single implementation of the WS-ResourceLifetime explicit destruction operation can be reused in any service. The resource implementation is a Java object that implements a set of appropriate callback interfaces. For example, a resource can choose to implement a *ResourceLifetime* interface to enable its destruction through soft state.

Resources are managed by a *ResourceHome*, which is responsible for resource discovery, destruction, and/or creation. The GT4-Java design addresses flexibility and scalability as follows. The core functionality is defined as interfaces or abstract classes so that custom and more optimized implementations of these interfaces can be plugged in. For example, a custom *ResourceHome* implementation could offload most or all of its operations to a database, minimizing the overhead of keeping the resource representation in memory.

GT4-Java provides two basic *ResourceHome* implementations. One, for persistent resources, relies on the JVM garbage collector and a caching algorithm (e.g., least-recently-used) to remove unused resource objects from memory. The other provides a basic persistence helper API for serializing and deserializing resources to and from a file using Java and XML serialization methods. A resource can of course choose to provide its own way of storing and retrieving its state.

GT4-C generates C-language stubs for a WSRF-enabled service that implement functions for each operation defined in the service schema, and that support EPR encapsulation by allowing EPR handles to be passed directly to each stub. Thus, for example, the createCounter operation of the CounterService is usually called with just the service endpoint: e.g., "http://.../CounterService" in Figure 2.

```
result = CounterPortType_createCounter(
    client_handle,
    "http://.../CounterService",
    createCounterInput,
    &createCounterResponse,
    &fault_type, &fault_value);
```

Figure 2. Creating a GT4-C service endpoint

On the other hand, the preferred function call for the add operation takes a handle to the EndpointReference, thus using the EPR as an opaque reference to the WS-Resource (Figure 3).

```
result = CounterPortType_add_epr(
    client_handle,
    createCounterResponse->
            EndpointReference,
    add_value, &add_response,
    &fault_type, &fault_value);
```

Figure 3. Client sall for GT4-C Add operation

The C bindings generated include both blocking client stubs and also asynchronous event-driven functions that allow client implementations to perform many invocations at once, instead of waiting for each response. This machinery allows the underlying bindings to take advantage of multithreading on SMP architectures, and also provides a simple mechanism for performing many operations asynchronously.

```
globus_result_t
CounterPortType_add_impl(
    globus_service_engine_t engine,
    globus_soap_message_handle_t message,
    globus_service_descriptor_t *
        service,
    xsd_int * add,
    xsd_int * addResponse,
    const char ** fault_name,
    void ** fault);
```

Figure 4. GT4-C service skeleton binding

The passed-in callback for the asynchronous stub only gets called when the response has been received and deserialized. The service skeleton bindings (Figure 4) consist of a function per operation that must be filled in by the service implementor. Once these service implementation functions are filled in, the service can be considered implemented. Instead of carrying state via object encapsulation as many other languages choose to do, any state information is maintained by the parameters passed to the function.

pyGridWare seeks to provide as simple an interface as possible while still exposing full WSRF/WSN functionality. The combination of the quick development time of Python and the ease of use of pyGridWare make it an ideal platform for

rapid prototyping. Figure 5 shows the simplest possible client. First the generated Counter Service code is imported. A locator and a port are instantiated and used to locate the service and create an instance. From the create response we get the EPR, which is used to access our newly created counter service instance. Next an add request is instantiated using the generated Counter Service code. We get a new port object from the locator object using the EPR; the new port corresponds to the counter service instance previously created. Using the port object we send our add request to the Counter Service.

```
import Counter_Service as COUNTER_SERVICE

locator =
   COUNTER_SERVICE.CounterServiceLocator()
port =
   locator.getCounterPortType(
                         portAddress=url)
request =
   COUNTER_SERVICE.CreateCounterRequest()
response = port.createCounter(request)
epr = response._EndPointReference
request =
    COUNTER_SERVICE.AddInputMessage(10)
port = locator.getCounterPortType(
      portAddress=url,
      endPointReference=epr)
response = port.add(request)
```

Figure 5. pyGridWare counter client

On the server side, the developer must edit two automatically generated files to implement their service. The server file contains the implementation of the service logic, and the properties file contains the associated Resource Properties. This division allows for a clean separation between the service state as represented by the Resource Properties and the stateless service.

The goal of **WSRF.NET** is to make programming a WSRF.NET service as easy as programming any other Web service. WSRF.NET provides an attribute-based programming model that allows service authors easily to 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.

For example, consider the code fragment shown in Figure 6. 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).

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

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

    public MyService() { // constructor }

    [WebMethod]
    public int MyMethod()
    { // service's methods }
}
```

Figure 6. WSRF.NET service code

Note that the ResourceProperty value can be computed dynamically, using a portion of the WS-Resource state if required. Finally, the [WSRFPortType] attribute makes it straightforward for the service author to allow the service to support the WS-ResourceProperty 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.

Finally, Perl's type-less nature, strong support for text manipulation, and dynamic nature make **WSRF::Lite** useful for rapid prototyping of Web services. On the minus side, there is little support for automatic WSDL generation, and Perl implementations of certain important Web service specifications such as WS-Security [22] are lacking as of May 2005. Figure 7 shows a sample Perl module that provides a counter WS-Resource.

```
package Counter;
use strict;
use vars qw(@ISA);
use WSRF::Lite;

@ISA = qw(WSRF::WSRL); #inherit WS-RF ops

# Declare our ResourceProperty count
$WSRF::WSRP::ResourceProperties{count}    =
0;

#add operation
sub add {
    my ($class, $val, $envelope ) = @_;

 #increment counter
  $WSRF::WSRP::ResourceProperties{count} =
    $WSRF::WSRP::ResourceProperties{count}
       + $val;

 #return a SOAP Header and the new value
 #for count
 return WSRF::Header::header($envelope),
   $WSRF::WSRP::ResourceProperties{count};
}

1;      #end of module
```

Figure 7. WSRF::Lite counter service code

4.8. WS-Notification

A system that supports WS-BaseNotification must provide for the registration and processing of subscriptions. The implementations support notification to varying degrees. WSRF.NET implements all three WS-Notification specifications. GT4-Java and pyGridWare do not implement WS-BrokeredNotification and only support flat topic spaces and basic subscriptions: the precondition, selector, and the subscription policy elements of the subscription are ignored. GT4-C does not implement producer-side notification (NotificationProducer, SubscriptionManager). WSRF::Lite does not support any Notification specifications.

4.9. Authorization

The systems differ significantly in the sophistication of the authorization mechanisms provided. In brief, GT4-Java, pyGridWare, and WSRF.NET define an authorization callout that allows the service developer to provide custom authorization behavior.; GT4-Java also provides several build-in implementations for this callout. GT4-C implements three built-in mechanisms, while WSRF::Lite client security information to the WS-Resource through environment variables in a similar manner to CGI scripts in Apache, leaving the WS-Resource implementation to implement its own authorization if desired.

pyGridWare's callout interface provides the user function with the security context and requested operation name; the user function returns true or false depending on whether the requester is authorized to call the operation. This interface can be used to implement a simple ACL list or to interact with a policy decision point (PDP). No default implementation is provided, but an authorization module that can parse SAML tokens is planned. WSRF.NET is similar to pyGridWare with regard to its authorization support.

GT4-Java provides a flexible infrastructure level framework for making authorization decisions. This framework defines a PDP interface, which PDPs implement to provide the framework with authorization decisions. PDPs may be chained to arrive at a final authorization decision. The evaluation is performed in a "permit overrides" fashion, i.e., if any PDP in the chain returns denied the whole chain evaluates to denied. GT4-Java also provides four different PDP implementations: (a) self authorization, which evaluates to permit if the client identity and the identity of the target match; (b) identity authorization, which evaluates to permit if the client identity matches the identity specified when the PDP was created; (c) gridmap authorization, which evaluates to permit if the client identity is found in a ACL (and also returns a local identity for the client, such as a local UNIX user name), and (d) OGSA SAML-based authorization callout, which evaluates to permit if the authorization service that this PDP interacts with authorizes the client.

GT4-C supports basic host, identity, and self authorization. The integration of an authorization callout interface is planned.

5. Performance Evaluation

To compare the performance of the five systems, we defined WSDL for a "counter service" and created five service/client implementations based on this WSDL. Using this service we then compared round-trip times for common WSRF/WSN operations.

The example counter service has a single resource property which expresses the value of the counter. The property can be set and retrieved using standard WSRF functions (GetResourceProperty and SetResourceProperty). Once created, counters can be destroyed by using WS-ResourceLifetime functions. Interested parties can also be notified of changes in the counter's value using WS-Notification.

We defined five performance tests: four that evaluate key primitive operations, and one that evaluates WS-Notification.

1. **GetRP**: The average duration over 10000 invocations for client to invoke

GetResourceProperty (getting the value of the counter).
2. **SetRP**: The average duration over 10000 invocations for client to invoke SetResourceProperty (setting the value of the counter).
3. **CreateR**: The average duration over 1000 invocations for client to create a counter as a WS-Resource. We use 1000 invocations in this test and the next to achieve a manageable duration.
4. **DestroyR**: The average duration over 1000 invocations for client to destroy a WS-Resource counter.
5. **Notify**: A client first subscribes to the "ResourcePropertyValueChanged" event for a particular counter. Then, we measure the average over 100 times of first setting the counter to a new value (via SetResourceProperty) and then waiting for the notification to arrive.

We ran each of these 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 four identically configured machines: Dual (2x) AMD Opteron 240 - 1.4GHz w/1MB L2 Cache, 2GB (4x512MB) PC2700 DDR333 Reg. ECC, 1x Seagate 120GB EIDE 7200 RPM, 8MB cache. Two machines ran Windows Server 2003 and were used only for the WSRF.NET tests. The other two machines ran redhat 8.0 (Linux kernel 2.6.9-1.667smp) and were used by every other project. For the GT-4 Java tests, Sun JVM 1.4.2_04-b05 was used. The JVM was started with "–Xms64m –Xmx256M" options. (Scenarios 3 and 6 were tested with GT 4.0.1, with connection persistence; Scenarios 1, 2, 4, and 5 were tested with GT 4.0, without connection persistence). In the C tests, the gcc compiler was used with –O3 optimizations.

Tables 2-4 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.

Table 2: No security (co-located/distributed). See text for details.

	GT4 Java	GT4 C	pyGridWare	WSRF::Lite	WSRF.NET
GetRP	10.05 / 10.05 ms	2.24 / 2.34 ms	25.65 / 25.57 ms	17.6 / 17.1 ms	8.87 / 8.23 ms
SetRP	10.06 / 10.12	2.26 / 2.34	26.17 / 25.83	20.4 / 19.5	8.91 / 8.95
CreateR	16.34 / 15.82	2.22 / 2.29	28.50 / 27.86	15.0 / 15.0	17.68 / 17.91
DestroyR	14.04 / 13.97	2.15 / 2.21	24.65 / 24.24	18.0 / 17.0	13.22 / 13.13
Notify	27.83 / 26.25	8.78 / 9.03	46.52 / 47.38	N/A	33.85 / 43.12

Table 3: X.509 signing of request and response (co-located/distributed). See text for details.

	GT4 Java	GT4 C	pyGridWare	WSRF::Lite	WSRF.NET
GetRP	182.66 / 181.96 ms	15.77/14.77 ms	139.65/140.50 ms	N/A	82.4 / 81.39 ms
SetRP	182.47 / 182.04	15.88/14.99	140.74/142.21	N/A	81.84 / 82.48
CreateR	188.21 / 188.46	15.95/14.98	133.41/132.26	N/A	96.88 / 96.22
DestroyR	182.47 / 182.03	17.10/15.76	137.12136.12	N/A	86.42 / 86.89
Notify	221.28 / 219.51	N/A	152.10/244.93	N/A	100.01 / 101.57

Table 4: HTTPS (co-located/distributed). See text for details.

	GT4 Java	GT4 C	pyGridWare	WSRF::Lite	WSRF.NET
getRP	11.81 / 11.46 ms	2.75 / 2.85 ms	151.35/149.67 ms	80.7 / 55.6 ms	9.37 / 12.91 ms
setRP	11.80 / 11.47	2.77 / 2.86	152.27/150.79	81.7 / 96.9	9.76 / 12.3
createR	18.35 / 18.00	2.74 / 2.82	158.33/132.60	78.0 / 53.0	18.17 / 20.84
destroyR	15.48 / 14.92	2.63 / 2.71	151.33/149.21	81.0 / 56.0	14.55 / 16.05
Notify	31.22 / 29.26	9.26 / 9.67	172.60/169.07	N/A	35.84 / 45.0

As might be expected, GT4-C was the fastest in every test. WSRF.NET and GT4-Java were comparable with "no security." WSRF.NET is faster with https because the Microsoft IIS used by WSRF.NET implements TLS session caching, allowing a new connection to re-use a previously-established TLS session key for the client/server—thus avoiding the expensive session set-up via the TLS handshake protocol. GT4-Java and GT4-C also implements HTTP connection caching, but pyGridWare did not at the time of this writing (although they are currently implementing it). One interesting effect, particularly observable with pyGridWare, is that a client and a service sometimes run faster when on different machines than when co-located, because of the CPU-intensive nature of some of the tests. The data collectively provide a nice assessment of the state of the art with regard to WS-Security and TLS implementations -- message-level security with X.509 signing is an order of magnitude slower than transport-level security via TLS.

6. Interoperability

One important aspect of the WSRF/WSN specifications is that they provide interoperable formats for common message exchanges between clients and services. In assessing the interoperability of the five systems, we used each project's performance test client against the other projects' performance test services in the "no security" scenario. Given the significant progress and effort that each project continues to make with regard to interoperability, there are relatively minor issues involving all of the WSRF/WSN implementations. Most importantly, our observations to date reinforce that interoperability is not necessarily a trivial concern for today as well as for the future (as is the case for all Web services specifications).

While the state of interoperability is not what was naively hoped for (if everyone implements the same specifications, they will interoperate by default), it is interesting to examine the reasons for failure. In some cases, HTTP headers (i.e. transport-specific details outside the WSRF/WSN specifications) were at issue. In other cases, application-specific portions of messages (again, outside the scope of WSRF/WSN) caused problems. Namespace incompatibilities were the key interoperability concern that does fall within the scope of the specs. However, some of these incompatibilities arise from different versions of the evolving WSRF/WSN specs having different namespaces (OASIS uses release dates in the namespaces for example). While more work is needed in order to have end-to-end interoperability between the five systems, it is encouraging that many of the interoperability issues are not due to the specifications themselves, but rather idiosyncrasies of the projects' toolkits.

7. Summary and Future Directions

We have presented a detailed analysis of five different implementations of WSRF and WSN, noting numerous areas of commonality and also significant differences of approach in some key areas. We have noted, in particular, differences in programming model and in overall performance.

The five teams also all plan further development, as we now review. In addition to these specific tasks, each team intends to update their implementations to meet the final WSRF and WSN specifications.

The **GT4-Java** team will introduce advanced service management functions such as service isolation and hot deployment, continue to increase performance, and improve handling of low memory conditions. The bulk of Java WS Core's functionality was submitted to the Apache Software Foundation's new Apollo and Hermes projects, which are now undergoing the incubation process. Future Java WS Core development will occur within these projects.

The **pyGridWare** team will implement SSL session caching and HTTP connection caching in pyGridWare to increase performance, and provide tools to automate wrapping command line applications and legacy codes as WSRF-compliant Web services.

The **GT4-C** team will fill out WSRF/WSN support by adding producer-side Notification support for ConcreteTopicPath and FullTopicPath expressions (only SimpleTopics are currently supported) and complete support for WS-ResourceProperties: specifically, "Query Resource Properties". They are also working to improve the performance of their marshalling infrastructure and on infrastructure and usability improvements, including more robust authorization mechanisms and better runtime deployment.

The **WSRF::Lite** team is focused on WS-Security, after which they will address issues of usability, interoperability with other WSRF implementations, and notification.

The **WSRF.NET** team will further develop the programming model for Web and Grid services, and focus on the use of WSRF.NET for building higher-level services such as security services.

References

[1] 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

[2] 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.

[3] 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

[4] Foster, I. and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, *11* (2). 115-128. 1997.

[5] pyGridWare: Python Web Services Resource Framework. http://dsd.lbl.gov/gtg/projects/pyGridWare/

[6] WSRF::Lite -- Perl Grid Services. http://www.sve.man.ac.uk/Research/AtoZ/ILCT

[7] 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.

[8] 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.

[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] S. Graham and J. Treadwell eds. Web Services Resource Properties (WS-ResourceProperties). Version 1.2. 04/05/2005. http://docs.oasis-open.org/wsrf/2005/03/wsrf-WS-ResourceProperties-1.2-draft-06.pdf

[12] J. Frey and S. Graham, eds. Web Services Resource Lifetime (WS-ResourceLifetime) Version 1.2. 03/23/2005. http://docs.oasis-open.org/wsrf/2005/03/wsrf-WS-ResourceLifetime-1.2-draft-05.pdf

[13] T. Maguire and D. Snelling, eds. Web Services Service Group (WS-ServiceGroup). Version 1.2. 02/18/2005. http://docs.oasis-open.org/wsrf/2005/03/wsrf-WS-ServiceGroup-1.2-draft-04.pdf

[14] S. Tuecke, L. Liu, S. Meder, eds. Web Services Base Faults (WS-BaseFaults). Version 1.2 03/24/2005. http://docs.oasis-open.org/wsrf/2005/03/wsrf-WS-BaseFaults-1.2-draft-04.pdf

[15] 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

[16] S. Graham and B. Murray, eds. Web Services Base Notification (WS-Base Notification). Version 1.2. 06/21/2004. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf

[17] D. Chappell, L. Liu, eds. Web Services Brokered Notification (WS-BrokeredNotification). Version 1.0. 07/21/2004. http://docs.oasis- open.org/wsn/2004/06/wsn-WS-BrokeredNotification-1.2-draft-01.pdf

[18] W. Vanbenepe, ed. Web Services Topics (WS-Topics). Version 1.2. 07/22/2004. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf

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

[20] A. Barbir, M. Gudgin, and M. McIntosh, eds. Web Services Interoperability Organization (WS-I) Basic Security Profile Version 1.0. Working Group Draft. 2004/05/12. http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html

[21] A. Nadalin, ed. Web Services Secure Conversation Language (WS-Secure Conversation). Version 1.1. May 2004. ftp://www6.software.ibm.com/software/developer/library/ws-secureconversation.pdf

[22] OASIS. Web Services Security: SOAP Message Security 1.0 (WS-Security 2004). OASIS Standard 200401. March 2004. http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[23] Allcock, W., Bresnahan, J., Kettimuthu, R. and Link, J., The Globus eXtensible Input/Output System (XIO): A Protocol-Independent I/O System for the Grid. Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models, IPDPS 2005.