

Exploiting WSRF and WSRF.NET for Remote Job Execution in Grid Environments

Glenn Wasson and Marty Humphrey

Department of Computer Science, University of Virginia, Charlottesville, VA 22904

{ wasson | humphrey } @cs.virginia.edu

Abstract: The Web Service Resource Framework (WSRF) was announced in January 2004 as a new way for manipulating "stateful resources" to perform grid computing tasks using Web Services. This paper discusses, to our knowledge, the most complex application built to date utilizing WSRF concepts. We describe a remote job execution system that handles scheduling, data movement, security and inter-job dependencies. The application is built on top of WSRF.NET, the first publicly-available toolkit for constructing WSRF-compliant Web Services. WSRF.NET is an open-source project that implements both the WSRF and WS-Notification family of specifications and runs on the Microsoft .NET platform. Our explicit goal in the effort reported in this paper is not to replace the existing tools such as GRAM/GlobusRun, MDS and Condor/Condor-G but rather to evaluate--via a concrete implementation--the new capabilities provided for such Grid operations via WSRF. We find that, in general, WSRF and WS-Notification facilitate far richer client-side and server-side interactions than previously accomplished in the state of the art in Grid computing.

1. Introduction

The Web Service Resource Framework (WSRF) [1] is a new set of specifications for achieving the Open Grid Services Architecture (OGSA) vision of grid and web services [2]. WSRF defines the concepts of "stateful resources" and how they can be discovered, queried and manipulated via web services. WSRF addresses many of the issues that arose in the development of the Open Grid Services Infrastructure (OGSI) specification [14], the first to implement OGSA ideas. However, since WSRF is new, few WSRF-based applications have been built and so there has been little to no evaluation of its utility in practice.

This paper presents, to our knowledge, the most complex application built to date using WSRF concepts. This application is a remote job execution testbed that runs "job sets" on behalf of users. A job set is a collections of jobs in which the output of one is used as input to the next. The testbed consists of web services utilizing WSRF and WS-Notification [6] to handle scheduling, data movement, security and asynchronous messaging. The testbed is built on top of WSRF.NET [9][17], the first publicly available toolkit to implement WSRF (and WS-Notification). WSRF.NET is, itself, built on the .NET framework from Microsoft. This paper describes the application and its components as well as a brief overview of WSRF.NET.

The manipulation and management of state is a fundamental and complex issue in distributed systems, and WSRF is a set of specifications proposed to systematically accomplish this task via canonical patterns, interfaces and behaviors. In creating the first publicly-available implementation of WSRF in WSRF.NET, we have recently identified a number of issues with WSRF -- most

notably the implied programming model derived from the specifications, particularly the complexity of service-side and client-code and the complexity of WS-Notification [9]. However, prior to the work described in this paper, we had identified these issues in very general terms without a concrete use-case representative of Grid computing. By implementing a canonical use-case of Grid computing -- the remote execution scenario -- via WSRF and WSRF.NET, we more precisely evaluate the WSRF abstractions. It is important to note that we are not attempting to create new tools specifically as replacements for existing tools that provide this functionality such as GRAM/GlobusRun, MDS and Condor/Condor-G, but rather we see the value of this work to the community as an evaluation of the core WSRF abstractions themselves. We find that, in general, WSRF and WS-Notification facilitate far richer client-side and server-side interactions than previously accomplished in the state of the art in Grid computing. As WSRF is the foundation of WSRF.NET and the impending Globus Toolkit v4, we believe this evaluation shows that WSRF is an important step forward for the Grid community.

The remainder of this paper is organized as follows. Sections 2 and 3 briefly describe WSRF and WSRF.NET respectively. Section 4 describes the testbed architecture, component services and system operation. Section 5 provides a discussion of interesting features of the application and uses these features to evaluate the utility of WSRF concepts in the application. Section 6 concludes.

2. The Web Service Resource Framework

The Web Services Resource Framework defines the notion of "stateful resources" [1], an abstraction for the state with which a web service interacts. WSRF employs a convention on the use of web service technologies (e.g. WS-Addressing [16]) to define the relationship between the service and the resource. The combination of a web service and a stateful resource is referred to as a WS-Resource. Each WS-Resource is described by an XML document called the Resource Properties document. The schema for this document is part of the web service's WSDL. WSRF consists of a set of specifications -- WS-ResourceProperties [4], WS-ResourceLifetime [3], WS-BaseFaults [13] and WS-ServiceGroup [5] -- that define how WS-Resources are named, discovered, queried, indexed, altered, and how their lifetimes are managed.

A separate but related family of specifications, WS-Notification, defines a system for delivering asynchronous messages between web services or between web services and clients. This set of specifications includes WS-BaseNotification [6], WS-Topics [7] and WS-BrokeredNotification [8]. These specifications define formats for notification messages, expressions for describing the messages of interest to clients, and how intermediaries (brokers) can be used to architect various message delivery scenarios.

3. WSRF.NET

WSRF.NET [9][15][17] is a toolkit for implementing WSRF-compliant web services that is implemented on top of the Microsoft .NET platform. Service authors annotate their web service code with metadata via .NET attributes. WSRF.NET tools process this information to convert the original web service into a WSRF-compliant service. Figure 1 shows the WSRF.NET system architecture.

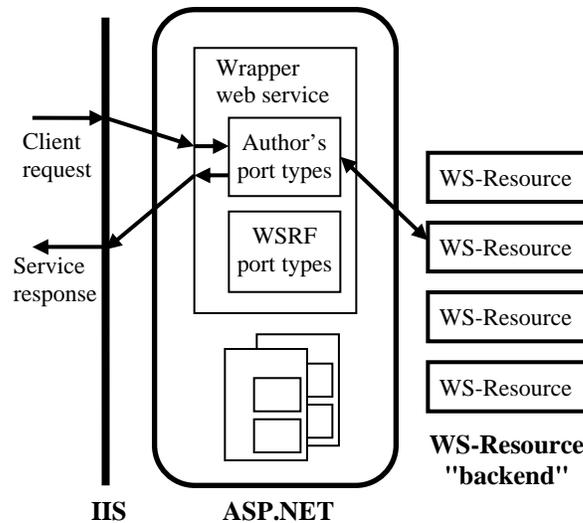


Figure 1. WSRF.NET System Architecture

Notice that WSRF.NET services *are* ASP.NET web services (which are the "normal" Web Services exposed via IIS on a Windows machine). WSRF.NET tooling generates the "wrapper" web service from the web service written by the service author and certain WSRF specification-defined port types that the author chooses to include. This service then runs as a normal web service in the ASP.NET worker process. IIS dispatches HTTP requests to the service, which internally invokes either a method on a port type written by the service author or a port type defined by WSRF (an implementation of which is included with WSRF.NET). Before the correct method is invoked, the wrapper service uses the value of the EndpointReference (EPR) [16] in the <To> header of the invocation SOAP message to interact with a particular WS-Resource and retrieve state values. This typically means querying a database to get the value(s) attached to the unique name given in the <ReferenceProperties> element of the EPR. These values are then made available to the invoked method and when the method completes, any changes to those values will be saved back to the database. Finally, the method result will be serialized and returned to the client by ASP.NET/IIS.

WSRF.NET automatically resolves the execution context presented in the EndpointReference and provides a programmatic interface to access the specified WS-Resource. Although there are many different resolution mechanisms and types of WS-Resources, WSRF.NET implements WS-Resources using any ODBC compliant database (e.g. MS SQL, MSDE, MySQL). State values associated with a given EPR are loaded into the service from the database during method invocation and saved back to the database when the method is done. We refer to this as the "WS-Resource as state" abstraction, i.e. a collection of state values is represented as a WS-Resource. WSRF.NET also supports the "WS-Resource as process" abstraction in which Windows processes are represented as WS-Resources. The remote execution application in section 4 makes use of this support. When a WS-Resource involves a process, the act of creating a new WS-Resource includes using WSRF.NET's process launcher Windows Service¹ to start a new process as a particular user. Authorized clients can then interact with that WS-Resource to query or change the state of the process (e.g. see if it has exited, or kill it).

¹ Note that a Windows Service and a Web Service are different. Windows Services are operating system services that deal only with the local machine and they are not typically accessible via the web.

Internally, WSRF.NET defines an interface for WS-Resource manipulation that service authors can use to define their own types of resource (beyond those already supported by WSRF.NET). This interface defines functions for creating, destroying, loading and saving. While the current version of WSRF.NET (1.1.2) only uses this internally, the next version (2.0) will expose this interface to programmers, thereby allowing a larger set of WS-Resource abstractions (e.g., modeling legacy systems as WS-Resources).

3.1. Programming Services and Clients in WSRF.NET

WSRF.NET uses an attribute-based programming model in which service authors annotate their code with meta-data that describes how the WSRF.NET system should transform the author's service into a WSRF-compliant service. This meta-data takes the form of .NET attributes. Attributes can be added to virtually any construct in any language that compiles onto the .NET CLR [11], e.g. classes, functions, data members, etc.

Although there are a number of attributes used by WSRF.NET, three that are of particular interest are the [Resource] attribute, the [ResourceProperty] attribute and the [WSRFPortType] attribute. These attributes make it easy for programmers to express certain WSRF concepts in their service code. Figure 2 shows an example C# code fragment utilizing these attributes.

```
[WSRFPortType(
    typeof(GetResourcePropertyPortType)]
public class MyServ : ServiceSkeleton
{
    [Resource]
    public string some_data;

    [ResourceProperty]
    public string MyData
    {
        get { return "At " +
            DateTime.Now +
            " the string is " +
            some_data;
        }
    }

    [WebMethod]
    public int MyMethod() { ... }
}
```

Figure 2. Programming WSRF.NET

This code shows a portion of a web service (MyServ) which derives off the WSRF.NET base class (ServiceSkeleton). MyServ contains a public data member (some_data) annotated with the [Resource] attribute. This means that this data member is a part of the state of this service's WS-Resources. Recall that a WS-Resource is a combination of a web service and a "stateful resource". For this service, the "stateful resource" consists of a string. Before any web methods on MyServ are executed, WSRF.NET will use the information in the invoking SOAP headers to load a value for some_data from WSRF.NET's data base. The invoked method may then manipulate some_data as it would any other class data member. If the value of some_data is changed by the invocation,

WSRF.NET will save that new value back to the database. When another invocation message references the same WS-Resource in its SOAP headers, this new value of some_data will be reloaded.

The [ResourceProperty] attribute denotes that the C# Property it annotates should be exposed as one of the service's WS-ResourceProperties [4]. Any client can retrieve the value of the MyData property by calling one of the WS-ResourceProperties functions that retrieve Resource Property values (GetResourceProperty, GetMultipleResourceProperties or QueryResourceProperties). When one of these requests comes into the web service, WSRF.NET will execute the C# Property's "getter" function and return the value to the querying client. If a similarly annotated C# Property contains a "setter" function, it will be used by WSRF.NET when a client attempts to modify the Resource Property (using, for example, SetResourceProperties).

The [WSRFPortType] attribute can be seen annotating the MyServ class. This attribute tells WSRF.NET to allow this service to respond to methods in the GetResourceProperty port type defined in the WS-ResourceProperties [4] specification. In other words, this attribute allows service authors to easily import the functionality of particular port types (either included with WSRF.NET or defined by the service author) into their service. In this case, the MyServ service will implement two methods, MyMethod defined in Figure 2 and GetResourceProperty implemented by WSRF.NET.

This brief introduction to WSRF.NET and WSRF.NET programming necessarily leaves out many important aspects of the toolkit. Interested readers should see the WSRF.NET Programmer's Reference [15] and the WSRF.NET Tutorial [12] for complete documentation. The latest version of WSRF.NET, 1.1.2, is available for download at <http://www.ws-rf.net>.

4. Remote Job Execution Application

As part of our development of our University of Virginia Campus Grid (UVaCG), we have created a set of WSRF.NET services that allows remote execution on, and data movement between, Windows machines across our campus. The overall goal of the UVaCG will be to seamlessly integrate Windows machines (via WSRF.NET) and Linux/UNIX machines (via Globus Toolkit v4) for the campus.

Clients execute sequences of jobs (which we call job sets) in which the inputs to a job include the outputs from previous jobs. The client user describes the job set to a Scheduler service as tuples of { executable, input files, output files }. Input files (as well as the executable) can come from any of the file systems of machines in the grid or the client's file system (as the client's machine may not be in the grid). They can also come from the yet-to-be-generated output of other jobs. When a job uses the output of another job, this represents a dependency between jobs. The Scheduler will use this information when deciding on the order in which to run the jobs.

Figure 3 shows the system architecture of the UVaCG prototype testbed, which consist of a number of web services of five basic types. These are shown as ovals. Each machine in the testbed grid runs a File System service and an Execution service. They also run two Windows services, the ProcSpawn service and the Processor Utilization service. Our testbed also has a single

NotificationBroker service, a Scheduler service and a Node Info service. Each of these services is described in the remainder of this section (Sections 4.1-4.5), and then we describe how everything fits together in Section 4.6. Section 5 contains the discussion of how WSRF benefited (and negatively impacted) the design and implementation of this remote execution scenario.

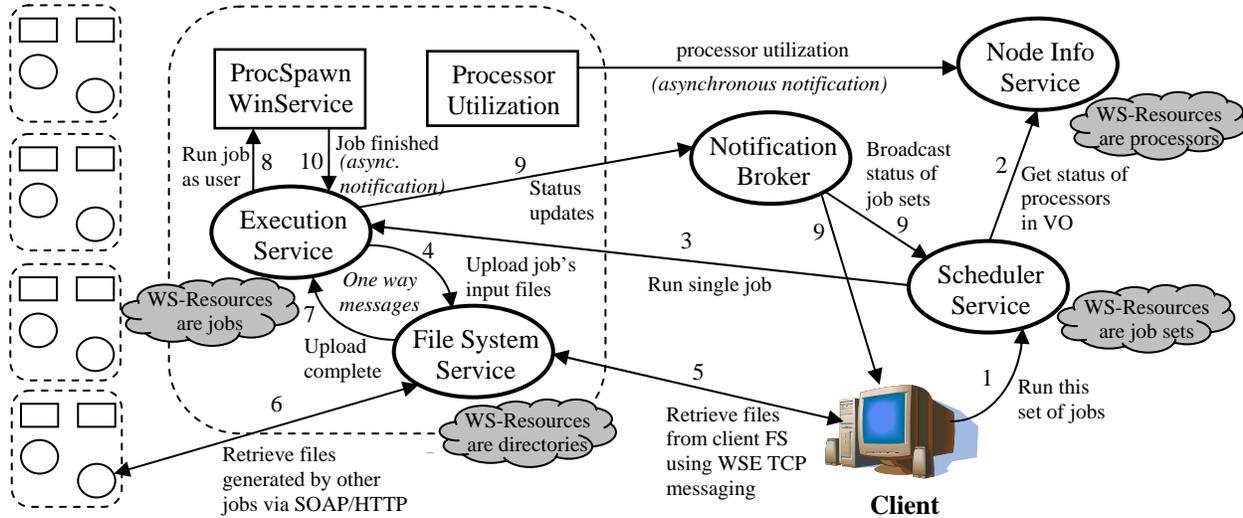


Figure 3. WSRF.NET Remote Job Execution Grid

4.1. The File System Service

The File System Service (FSS) controls access to the portion of the file system usable by the Campus Grid on the machine on which the FSS resides. The WS-Resources used by the File System service represent directories. That is, the EPR in the SOAP headers of each method invocation on the FSS refers to a particular directory and the invocation of the method is done in the context of this directory. These WS-Resources have a single WS-ResourceProperty [4] that provides the actual path to the directory they represent. This is used by the Execution service to specify the working directory for the processes it launches.

The FSS supports 3 methods: Read, Write and List. Read takes a filename parameter and returns to the client the contents of that file from the directory specified in the invocation EPR. Write takes a filename and an array of bytes and creates a new file with the given name in the specified directory. List returns the contents of the directory represented by the invocation's EPR.

The File System service was developed with the understanding that files are large and it is therefore inappropriate to have blocking method calls when uploading to a remote machine. However, this means some extra synchronization is necessary to ensure that a job doesn't start executing until its input files are available. An FSS typically receives a one-way message from the Execution service that contains a list of files to upload. Recall that a one-way message is not the same as a web service method with a void return value. A one-way message closes the connection immediately after sending the message while a void function will actually send a reply message with an empty message body. When the upload is complete, the FSS will send another one-way message (which we call a notification) back to the Execution service indicating that the job may start.

The list of files that the FSS should upload is a set of tuples of the form {EPR, filename, jobname}. The filename is the name of the file on the remote file system and jobname is the name that the FSS should give to the file once it has been uploaded (i.e., it is the name that the job will expect). The EPR names the WS-Resource representing the directory that contains the file. This could be a directory controlled by another FSS (i.e., on another machine) or a directory on the client's file system (which may not be part of the grid). Files can be transferred via HTTP, but this is not the preferred way to move large files. Instead, the FSS uses the Web Service Enhancements (WSE) [10] support for SOAP over TCP. When a client initiates an interaction with the testbed, a WSE TCP server thread is started on the client's machine and the FSS will contact it to retrieve files from the client's local system.

4.2. The Execution Service

The Execution service (ES) is in charge of managing all activities related to the execution of jobs on the machine on which it resides. The ES receives requests to run jobs from the Scheduler and then interacts with both the FSS and the ProcSpawn service to start the job's execution.

The ES's WS-Resources are jobs, meaning that clients can interact with their job by calling methods on the ES. Currently, these methods allow the client to kill the job or to inquire about its exit code (if it has exited). The jobs WS-Resources have two Resource Properties that allow clients to retrieve the job's status (running, exited, etc.) and the job's CPU time used so far.

The Execution service uses the ProcSpawn Windows service that is part of WSRF.NET to start Windows processes as particular users. Although we anticipate having either the ES or the ProcSpawn service be able to map "grid credentials" to local user accounts in the future, currently, the request to the ES must contain the username/password of the account in which the job should be executed. This information is conveyed using a WS-Security password profile SOAP header, which is then encrypted using the X509 certificate of the client.

When a request to run a job is received, the ES creates a new WS-Resource via the FSS. This causes a new directory to be created. The EPR for this directory WS-Resource is returned to the ES and is used as the working directory for the job. The ES then directs the FSS to upload the required input files for the job (including the executable). When the ES receives the message that the uploads are complete, it calls the ProcSpawn Windows service asking it to start the given binary, in the given directory, as the given user. When the job exits, the ProcSpawn service sends a notification message to the ES with the job's exit code. The ES then sends this information to other interested parties (e.g., the Scheduler service, the client user) using the Notification Broker service.

4.3. The Notification Broker Service

The Notification Broker service distributes messages about the status of various jobs in a user's job set to interested components throughout the testbed.

Notification Brokers are described in WS-BrokeredNotification [8] and are used when notification producers and consumers can not or do not care to have direct knowledge of each other. In our testbed, it is often useful to notify more than one party about a single event. While the web service

generating the event could maintain its own list of parties interested in receiving that event, it is more convenient to use the Notification Broker service as a multicast mechanism.

When a client initiates the running of a job set, the Scheduler service will subscribe both itself and the client to receive notifications about various events related to the running of those jobs. The Scheduler will use the notifications to coordinate the execution of jobs based on dependencies. The client application displays the messages to keep the user informed of the job set's progress.

4.4. The Node Info Service

The Node Info service (NIS) is a service group (as defined by WS-ServiceGroups [5]) whose members represent the processors available for scheduling. Each machine in the system runs the Processor Utilization Windows service. This service asynchronously notifies the NIS whenever the utilization of the machine's processors changes by more than a configurable amount. The NIS catalogs this information and delivers it to the Scheduler service upon request.

4.5. The Scheduler Service

The Scheduler Service (SS) is the heart of the remote job execution testbed because it coordinates the activities of the other grid components. The SS receives a description of a job set to run from the client and directs appropriate ESs to run the individual jobs. Each time a job is to be run, the SS polls the Node Info service to get the latest information about the grid's processors. The Scheduler selects an available machine and directs the corresponding ES to run a job. When the SS receives the notification (from the Notification Broker) that the job has completed, it will schedule the next job in the job set.

Since all jobs are scheduled based on dynamic conditions in the grid, the client's will not know where any given job will execute. This means that the job set description given by the client will not specify which of the various FSSs will have the output files from a job. The Scheduler service handles the "filling in" of this information, as it makes scheduling decisions. When the Scheduler asks an ES to run a job, the ES asks its corresponding FSS to create a new directory WS-Resource. The FSS sends the EPR for this new Resource to the ES which sends it to the Scheduler. The Scheduler then makes sure that any further jobs that reference the output of this job will use this EPR in their file upload requests.

4.6. Using the Remote Job Execution Testbed

Having briefly described all the services in the testbed, it is useful to run through an example of how a client executes a job set and what interactions take place between the services in doing so. We will also point out where WSRF mechanisms are used. The sequence of events that occurs in running a job set is shown by the numbered arrows in Figure 3.

First, the scientist uses a GUI tool to assemble the description of their job set. The tool starts a TCP-based server thread that will respond to requests for any input files that need to come from the scientist's local file system. The scientist specifies dependencies between jobs through the input file descriptions. For example, the input file "local://c:\file1" is a file the should come from the

local file system, while the file "job1://output2" means that the job designated as "job1" will produce an output file called "output2" and that file should be retrieved as input to the current job (from wherever "job1" ends up executing). The GUI adds the appropriate URI of the client's TCP file server to any file descriptions using the "local" scheme. Finally, the client program starts one of WSRF.NET's light-weight notification receivers to receive asynchronous, WS-Notification compliant, notifications via HTTP.

Once the job set description is created, it is sent to the Scheduler service (step 1). The Scheduler service then generates a unique topic name [7] for events related to this job set. The SS then invokes the Subscribe() method on the Notification Broker to subscribe both itself and the client's notification listener to receive notifications about the new topic. Next the Scheduler polls the NIS to get the latest processor utilization for the machines in the testbed, as well as their hardware characteristics, such as CPU speed and total RAM (step 2). A straightforward algorithm chooses the fastest, most available machine and then the ES on that machine is sent a request to run a job (step 3).

To run the job, the ES first creates a new directory WS-Resource by contacting the FSS that lives on its machine. The EPR for this directory is then included in the SOAP headers of a file upload request to the FSS causing the FSS to upload the job's files into that directory (step 4). The EPR is also broadcast to the Scheduler and the client (step 9) allowing the Scheduler to fill-in the location of any files generated by this job that will be used as input to future jobs. The client can use this EPR to retrieve files generated by the job or monitor progress by watching for changes in that directory.

The file descriptions given to the FSS to upload consist of a filename and EPR from which to retrieve the file. If the EPR's address component uses the HTTP scheme, the FSS will make a Read() request on the appropriate FSS (step 6). If the file happens to already be on the FSS's machine, the FSS simply moves the file within the portion of the file system it controls (rather than making an HTTP request on itself). If the EPR's address component uses the soap.tcp scheme, the FSS will use WSE TCP messaging to retrieve the file from the client's file system (step 5). When all the files have been uploaded, the client sends a notification message to the ES (step 7).

Now the ES starts the job using the WSRF.NET ProcSpawn Windows service. The ProcSpawner is given the name of the binary, the username/password under which to run it, and working directory (created by the FSS) and any arguments specified by the scientist. Since the request message from the Scheduler contains the topic name generated for this job set, the ES can send out a notification containing the job's EPR (step 9). These will be received by both the Scheduler and the client application and allow either to poll the job for its status (with GetResourceProperty calls on the Resource's status Property). When the job completes, the ProcSpawner sends a notification message to the ES with the process's exit code (step 10) and this is also broadcast via the Notification Broker.

When the Scheduler gets the message that a job has completed, it schedules the next job that no longer has any uncompleted dependencies.

5. Discussion

The collection of services described in Section 4 to accomplish the remote execution scenario that is typical of Grid environments provides one of the best opportunities to date to assess the potential contributions of the WSRF and WS-Notification specifications and abstractions. In our earlier paper, "An Early Evaluation of WSRF and WS-Notification via WSRF.NET" [9], we identified several key issues we found as we were implementing WSRF and WS-Notification. In this section, we review those issues having now built a much larger and more complete system using WSRF.NET.

Perhaps the most important issue in any WSRF-based system is that of state. One of the fundamental questions surrounding WSRF is what view of a service's state should a client possess and how should clients manipulate that state. WSRF proposes WS-ResourceProperties [4] as the mechanism for describing the portion of a service's state that is exposed to clients, as well as a standard set of functions for retrieving/manipulating that state. Additionally, the schema for this exposed state is carried in a service's WSDL. We are now in a position to start asking questions like "how does this canonical view of state (and access to state) aid clients?" While it will take time for the grid community to arrive at a definitive answer, one important argument has become clear. Standardizing the mechanism for describing/manipulating state aided us, as designers of WSRF.NET, in creating the "plumbing" that clients can use. Because WS-ResourceProperties [4] defines a small set of interfaces with standard behavior, it is possible to implement tooling to easily use them. While custom interfaces for manipulating state could be designed, and consumed by clients using standard WSDL tooling to create proxy classes, we believe -- and have seen in this remote execution scenario -- that WSRF.NET's support for WS-ResourceProperties allows for simpler interfaces. Not only do clients *not* have to create these interfaces themselves (i.e., generate proxies), but there is potential to develop higher-level interfaces to standard Resource Properties as part of WSRF.NET. This functionality could then be provided to all clients and work on all services, not just service/client pairs that had agreed upon their own specific interfaces.

Another issue brought out in [9] was the tightness or looseness of the coupling between clients and services. Loose coupling is a fundamental tenant of web services and the various EPRs kept by the client in our testbed could be seen as a tightening of that connection. This issue is still important and while notification may help in keeping the client's and service's view of the resources represented by those EPRs consistent, it does not alleviate it completely. Further exploration is needed to address issues such as the amount of state (in the form of EPRs) that the client is (or can be) expected to maintain. How durable does that client-side information need to be (e.g., should it survive client shutdown?) and how a client might possibly re-discover their resources should their EPRs be lost.

In implementing the services for the remote execution scenario, we have found that the complexity of service code is an issue that is more fundamentally tied to WSRF.NET than WSRF. One of the challenges in developing WSRF.NET was to provide an intuitive and easy to use interface to state on the server-side, i.e. to the WS-Resources themselves. In the remote execution testbed, we can see multiple kinds of WS-Resources -- the Execution service uses jobs (processes) as WS-Resources, the FileService uses directories as WS-Resources, the Node Info service uses processors as WS-Resources, and the Scheduler service uses job sets as WS-Resources. Each of these

Resources however, consists of some state (and in certain cases a running process) and WSRF.NET provides a database-back system for accessing it in service code. While we feel that modeling WS-Resource state as class member variables is intuitive for service authors and provides a valuable set of abstractions, it is not strictly necessary in the WSRF model and it does present challenges in load/saving/querying the data in the database. For example, a service can have an arbitrary structure to its Resource state, and yet WSRF.NET must be able to operate on it effectively. This can cause problems with relational databases which traditionally have fixed columns of fixed types. Saving a service's Resources as binary, unstructured data is effective for loading and storing, but makes it very difficult to query them in the database. For future version of WSRF.NET, we are currently experimenting with XML databases, such as Yukon, because they provide the ability to store and run queries over unstructured data.

A final issue is the complexity of the notification system proposed by WS-BaseNotification [6] and implemented in WSRF.NET. Although the WS-Notification family of specification provides a powerful system of asynchronous messaging, the benefits of this may not be fully realized if it is not easy for service authors and clients to use. One of the primary benefits of standardization of the notification system is that it can be made easy to use as part of the standard WSRF.NET plumbing. Custom mechanisms for asynchronous messaging are permitted by WSRF.NET (and WSRF), but they rely on the service and client agreeing on a "private" interface. To make notification easy to use in WSRF.NET, we have provided a topics-based interface in which notification consumers (sinks) register interest in various notification types (the topics) and provide functions to be called when those notifications are received. Essentially, the topic system acts as a filter allowing notification consumers to simply state (in various notification dialects), which messages they are interested in receiving. Sending notifications from a service is also made easy WSRF.NET provides a single function that services may invoke. Service authors can provide an XML message body or an object which will be serialized into a message body. As we saw in constructing certain services in the remote execution scenario, service authors need not know about the details of WS-Notification messages because WSRF.NET handles the generation of the actual message on the wire. This allows notification messages to be created in a format that can be consumed by receivers on many different platforms without the service author needing to deal with message format.

Overall, we found the abstractions of WSRF and WSRF.NET to be very valuable, particularly in the ability to model multiple types of "resources" and the ease at which notifications were generated and consumed. While many of the issues we identified in [9] remain unresolved, we believe that in building such an extensive system we were able to revisit and refine issues regarding the value and usability of WSRF and WSRF.NET.

6. Conclusion

To our knowledge, our remote job execution testbed is the largest WSRF-based (and WSRF.NET-based) application built to date. As such, it provides a concrete case study in the use and utility of both WSRF.NET and WSRF. In studying WSRF and WSRF.NET as used for the remote execution scenario, in the context of the issues and concerns that we raised as part of an early evaluation of WSRF and WSRF.NET [9], we have found that many of the concerns are manageable via new programming abstractions and our improved tooling. We find that, in general, WSRF and WS-

Notification facilitate far richer client-side and server-side interactions than previously accomplished in the state of the art in Grid computing.

We are currently building a campus-wide grid at the University of Virginia (UVaCG) based on the technology developed for this application. The UVaCG and the applications developed to run on it will harness the campus's Windows machines and provide an even larger environment to evaluate the WSRF.NET toolkit and the WSRF specifications. We have recently begun testing interoperability between WSRF.NET and the Globus Toolkit v4 (actually, GT 3.9.2) in order to more fully develop this scenario into a working, stable Campus Grid. The latest version of our toolkit can be downloaded at <http://www.ws-rf.net>.

7. References

- [1] Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W. 2004. The WS-Resource Framework. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf>
- [2] Foster, I., Kesselman, C., Nick, J. and Tuecke, S. 2002. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Draft of 6/22/02. http://www.gridforum.org/ogsiwg/drafts/ogsa_draft2.9_2002-06-22.pdf
- [3] Frey, J., Graham, S., Czajkowski, C., Ferguson, D., Foster, I., Leymann, F., Maguire, T., Nagaratnam, N., Nally, M., Storey, T., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., and Weerawarana, S. 2004. WS-ResourceLifetime. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-resourcelifetime.pdf>.
- [4] Graham, S., Czajkowski, C., Ferguson, D., Foster, I. Frey, J., Leymann, F., Maguire, T., Nagaratnam, N., Nally, M., Storey, T., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., and Weerawarana, S. 2004. WS-ResourceProperties. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-resourceproperties.pdf>.
- [5] Graham, S., Maguire, T., Frey, J., Nagaratnam, N., Sedukhin, I., Snelling, D., Czajkowski, K., Tuecke, S., and Vambenepe, W. 2004. WS-ServiceGroups. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-servicegroup.pdf>.
- [6] Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., Patil, S., Samdarshi, S., Sedukhin, I., Snelling, D., Teucke, S., Vambenepe, W. and Weihl, B. 2004. Web Services Based Notification (WS-Base Notification). <ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf>.
- [7] Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., Patil, S., Samdarshi, S., Sedukhin, I., Snelling, D., Teucke, S., Vambenepe, W. and Weihl, B. 2004. Web Services Brokered Notification (WS-BrokeredNotification). <ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BrokeredN.pdf>.
- [8] Graham, S., Niblett, P., Chappell, D., Lewis, A., Nagaratnam, N., Parikh, J., Patil, S., Samdarshi, S., Sedukhin, I., Snelling, D., Teucke, S., Vambenepe, W. and Weihl, B. 2004. Web Services Topics (WS-Topics). <ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-Topics.pdf>.
- [9] Humphrey, M., Wasson G., Morgan, M., and Beekwilder, N. An Early Evaluation of WSRF and WS-Notification via WSRF.NET. 2004 Grid Computing Workshop (associated with Supercomputing 2004). Nov 8 2004, Pittsburgh, PA.
- [10] Microsoft. Web Services Enhancements (WSE). <http://msdn.microsoft.com/webservices/building/wse/default.aspx>
- [11] Microsoft .NET Framework Developer Center. 2004. <http://msdn.microsoft.com/netframework/>
- [12] Morgan, M. and Wasson, G. 2004. WSRF.NET Developer Tutorial. http://www.ws-rf.net/WSRFdotNet/WSRF.NET_Developer_Tutorial.pdf

- [13] Teucke, S., Czajkowski, K., Frey, J., Foster, I., Graham, S., Maguire, T., Sedukhin, I., Snelling, D., Vambenepe, W. 2004. WS-BaseFaults. <http://www-106.ibm.com/developerworks/library/ws-resource/ws-basefaults.pdf>.
- [14] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maquire, T., Sandholm, T., Snelling, D. and Vanderbilt, P. 2003. Open Grid Services Infrastructure (OGSI) version 1.0. https://forge.gridforum.org/docman2/ViewProperties.php?group_id=43&category_id=392&document_content_id=347
- [15] Wasson, G. 2004. WSRF.NET Programmer's Reference Manual. http://www.ws-rf.net/WSRFdotNet/WSRFdotNet_programmers_reference.pdf
- [16] WS-Addressing. 2004. IBM/BEA/Microsoft Corporation. <http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-addressing.asp>
- [17] WSRF.NET. 2004. <http://www.ws-rf.net>