J Grid Computing (2008) 6:177–194
DOI 10.1007/s10723-007-9064-x

Resource-oriented Computing: Design, Implementation, and Evaluation of WSRF.NET

Glenn Wasson · Norm Beekwilder · David Del Vecchio · Mark Morgan · Marty Humphrey

Received: 11 July 2006 / Accepted: 11 January 2007 / Published online: 21 February 2007

Abstract Web services have changed the nature of distributed systems development and operation. The Grid community has begun architecting Grid systems to leverage available commercial and open-source Web services technology through the definition of "stateful resources." These resource-oriented systems are an extension of service-oriented systems typically built using Web services in that they treat state (and the management of state) as an architectural concern rather than an application-level concern. We present the design, implementation, and evaluation of WSRF.NET, a toolkit for building resource-oriented services using the Microsoft .NET platform. We describe the benefits WSRF.NET provides over "pure" .NET Web services for resource-oriented systems development, in terms of programmability (both programming-language abstractions and compile-time tooling) and improved run-time persistence/management of state. The run-time overhead of WSRF.NET is quantitatively evaluated against other technologies that can be used to add state management to Web services. We argue the core WSRF.NET primitives incur negligible overhead compared to typical domain-specific resource manipulation operations. For example, a computational simulation that lasts 10 min and reads/writes 100 medium-sized resources over those 10 min incurs only 0.46% overhead in WSRF.NET operations.

Key words Grid computing · middleware · stateful services · web services · WSRF

G. Wasson (✉) · N. Beekwilder · D. D. Vecchio · M. Morgan · M. Humphrey
Computer Science Department, University of Virginia, 151 Engineer's Way, Charlottesville, VA 22904, USA
e-mail: wasson@virginia.edu

N. Beekwilder
e-mail: nfb5z@virginia.edu

D. D. Vecchio
e-mail: dad3e@virginia.edu

M. Morgan
e-mail: mmm2a@virginia.edu

M. Humphrey
e-mail: mah2a@virginia.edu

1 Introduction

Web services have fundamentally changed Grid computing [16]. While Grid computing broadly addresses both the *mechanism* and *policy* by which to collaborate across multiple administrative domains (and thereby form "Virtual Organizations"), the central aspect of Grid computing is middleware. By providing application-independent, transport-neutral mechanisms for interface description (WSDL) and message encoding (SOAP), Web services provide potential solutions to many problems that have persisted in Grid computing, particularly discovery and interoperability. In addition, the availability of Web service tooling has facilitated rapid adoption and made the ability to deploy Web services nearly ubiquitous. Commercial Web service infrastructures have been introduced by several companies, including IBM [30] and Microsoft [34]; these

"Web service hosting environments" consist of execution environments and tooling designed to make Web service programming, deployment, and management easier. Projects such as Tomcat [4] and Mono [38] provide Web service capabilities to the large open source community.

The introduction of Web services caused the Grid community, led by the Open Grid Forum (OGF) [18], to consider how Web service technology could be leveraged to "virtualize" the components of Grid systems [16]. From the work of the OGF and others, the concept of Grids as collections of interacting "resources" emerged. Resources in this context are stateful entities that receive (and typically send) messages. In Grid systems, these could be physical resources such as CPU, disk or network, or they could be software components such as applications or system services (e.g. schedulers, information services, brokers, etc.). These ideas began to take shape in 2002 with the release of the Open Grid Services Infrastructure (OGSI) standard [42] under the broader umbrella of the Open Grid Services Architecture [39]. In OGSI, the notion of resources had not yet been formalized. Instead, an architecture based on stateful services was proposed in which each service was meant to virtualize a separate resource (e.g. a file). While OGSI served as the inspiration for what followed, the standard itself met with limited success because of incompatibilities with available Web services infrastructure. In January 2004, the Web Services Resource Framework (WSRF) [46] and Web Services Notification (WSN) [47] families of specifications were introduced to address these problems. WSRF views applications as communicating collections of WS-Resources, which are combinations of Web services and stateful resources. In other words, a Web service provides the web-addressable "front-end" for the resource's functionality on the back-end. Each WS-Resource can be individually addressed and is described by an XML document with known schema. The WSRF specifications standardize interactions to discover, group, query and manipulate resources (via their corresponding WS-Resource). The WSN specifications provide a messaging architecture that, while independent of WSRF, is often referred to by WSRF as a means of sending asynchronous messages between resources.

This new resource-oriented architecture extends the notion of Service Oriented Architecture (SOA) [10]. The resource-oriented architecture focuses on state and the management of state as an architectural concern, while in an SOA state is typically treated as an application-level concern. As such, today's Web service programming environments do not explicitly support the resource-oriented model for building systems. This paper discusses WSRF.NET, a toolkit for building resource-oriented systems using the Microsoft .NET platform [36]. WSRF.NET provides an attribute-based programming model that we feel is significantly easier to use than "vanilla" Web services on top of which one must implement one's own mechanisms for manipulating stateful resources. WSRF.NET allows Web service programmers to simply expose and manipulate associated stateful resources via the WSRF/WSN defined messages. Also, WSRF.NET provides service and client-side libraries that automate many common WSRF/WSN tasks. In earlier work [25, 29], we presented the design and many open issues involving our first version of WSRF.NET; this paper presents the lessons we have learned since and offers the first quantitative evaluation of the run-time overhead of WSRF.NET as compared to other technologies that can be used to add state management to Web services. We argue that the core WSRF.NET primitives (Create, Query, Read, Write, Delete) incur negligible overhead compared to typical domain-specific resource manipulation operations. For example, a computational simulation that lasts 10 min and reads/writes 100 medium-sized resources over those 10 min incurs only 0.46% overhead in WSRF.NET operations.

The remainder of this paper is organized as follows. Section 2 discusses the WSRF and WSN specifications in more detail. Section 3 describes WSRF.NET, its architecture and programming model. Section 4 argues for the value that WSRF.NET provides to designers of resource-oriented services over standard .NET Web services. Section 5 contains an evaluation of WSRF.NET. Section 6 concludes.

2 The Web Services Resource Framework and Web Services Notification Specifications

WSRF [46] was introduced in January 2004 with the goal of defining conventions for representing, abstracting, and manipulating stateful resources via Web services. 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 [22]. The most common pattern is to address the WS-Resource using a WS-Addressing Endpoint Reference (EPR) [24]. WSRF defines various operations on resources which are achieved by sending standard messages to a resource's Web service (the one which is part of its WS-Resource). The four WSRF specifications, currently standardized in OASIS, are:

- *WS-ResourceProperties* [20] defines how a WS-Resource is described by an XML document 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. Elements of this document are referred to as Resource Properties. WS-ResourceProperties defines various methods for retrieving Resource Properties, from simple retrieval "by name" to full XPath queries of the document. Resource Properties can also be "set," meaning values for properties can be sent from client to service. It is up to the service to decide how these values should be incorporated into the stateful resource.
- *WS-ResourceLifetime* [41] defines mechanisms for destroying WS-Resources. Destruction can either be immediate or via a lease-based mechanism (resources terminate at some specified time in the future). Note that there is no "create" mechanism defined, as WSRF views this as application specific.
- *WS-ServiceGroups* [33] describes how collections of Web services and/or WS-Resources can be represented and managed. A grouping mechanism, based on common characteristics expressed through Resource Properties, is defined to provide access to a set of WS-Resources with those characteristics.
- *WS-BaseFaults* [32] defines a standard exception reporting format. Other WSRF and WSN specifications define fault conditions in terms of WS-BaseFaults.

The WSRF specifications are compliant with the WS-Interoperability Basic Profile [7], meaning that any WS-I-compliant Web services client can interact with any service that supports the WSRF specifications.

The WS-Notification (WSN) family of specifications is also separately standardized in OASIS. These three specifications are: WS-BaseNotification [21], WS-BrokeredNotification [13], and WS-Topics [43]. 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 service 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 cannot or do not care to know who is subscribed.

3 WSRF.NET

WSRF.NET is a toolkit for building resource-oriented, WSRF-compliant Web services and clients. It consists of a programming model (and associated tooling) as well as a set of service and client-side libraries that automate common tasks. The general architecture of a WSRF.NET service is shown in Fig. 1.

WSRF.NET services are normal Microsoft Web services. That is, they are indistinguishable from any other Web service to Microsoft's Internet Information Services web server (IIS) and web service container (ASP.NET). From the Microsoft web infrastructure's

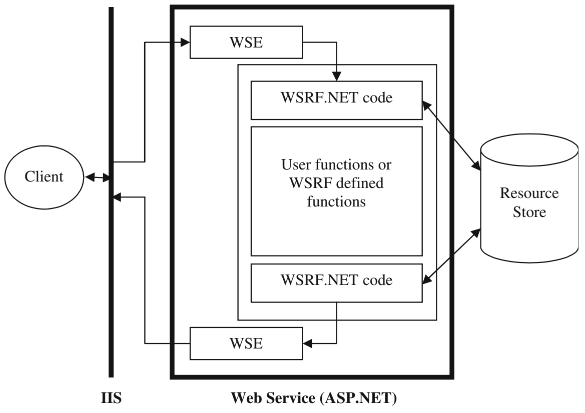


Fig. 1 WSRF.NET Service Architecture

perspective, WSRF.NET service code may be executed as any other service code is executed. A client invocation proceeds as follows. The client's request message (typically encoded in SOAP and sent over HTTP) is received by IIS and forwarded to ASP.NET (based the request URL suffix, .asmx for Web services). ASP.NET creates an Application Domain (a protected memory segment) for the service to execute in, instantiates a new instance of the Web service class and sends the client message through the Web Service Enhancements (WSE) filters [37] to process the SOAP message headers (for security processing for example). Next the WSRF.NET service code executes. A WSRF.NET service consists of a service author's Web service "wrapped" by WSRF.NET code. The initial block of WSRF.NET code, which runs before the service author's code, interacts with one of WSRF.NET's compatible back-end resource stores to "load" the resource state into memory. Next the Web service author's code runs, presumably accessing and/or manipulating the resource state and finally returning results for the client. Before those results are actually placed on the wire to the client, WSRF.NET code stores any changes to the resource state back to the resource store. Finally, results are returned to the client via WSE output filters and then IIS. ASP.NET now destroys the Web service object it created for this call. Recall the standard Microsoft Web services as stateless and hence the WSRF.NET "storage" step is necessary to preserve resource state. In this way, resource-oriented systems can be built on top of stateless Web services.

The exact activities that occur when "loading" or "storing" a stateful resource depends on the nature of the resource itself. WSRF.NET natively supports three types of resources. First, it supports data resources (e.g. a healthcare record). These resources are stored in either an XML database (WSRF.NET supports Microsoft SQLServer [35] and Xindice [5]) or an in-memory XML document. Loading and storing such resources is done through XPath queries and SQL commands respectively. The second type of resource supported by WSRF.NET is the Windows process. WSRF.NET can create (launch) processes and provide information about the state of the process to web clients. WSRF.NET uses a Windows service to hold the process handle across Web service invocations, thus keeping the process information available in the OS. Loading this resource involves binding a process handle usable by the Web service to one held by the Windows service. Storage is simplified for this resource because of the limited ability of client to affect processes through these handles. Any changes that can be made (including killing the process) are immediately sent to the operating system through the process handle and therefore all that must be maintained by WSRF.NET is the process handle. The last type of resource supported by WSRF.NET is the "custom" resource. Any other type of resource, such as scientific instruments or legacy data, can be used by WSRF.NET simply by creating wrapper objects that implement the IResource interface. The IResource interface consists of loading and storing methods which can be custom tailored to any "back-end" resource, allowing WSRF.NET to expose it as a WS-Resource.

3.1 Programming WSRF.NET

The guiding principle of WSRF.NET's programming model is to make programming a WSRF.NET service as easy as programming any other Web service. WSRF.NET's programming model provides an abstraction for constructing, representing and manipulating stateful resources, as well as a means of easily generating messages that comply with the WSRF protocols. These capabilities are accessed through a set of metadata tags, called attributes in .NET, which can be placed on a Web service class (or members of the class). An example of a WSRF.NET service using these Attributes is shown in Fig. 2. The text enclosed in square brackets ([]) are the attributes.

The three WSRF.NET attributes shown in Fig. 2 are [Resource], [ResourceProperty] and [WSRFPortType]

Fig. 2 WSRF.NET service code with attributes

```
[WSRFPortType(typeof(GetResourcePropertyPortType))]
[WSRFPortType(typeof(ScheduledResourceTerminationPortType))]
[WSRFPortType(typeof(NotificationProducerPortType))]
public class MyService : ServiceSkeleton
{
     [Resource]
     private string[] stringData;

     [Resource]
     [ResourceProperty]
     public int v;

     [ResourceProperty]
     public string Property1
     {
        get { return stringData[0]; }
     }

     [ResourceProperty]
     public string Property2
     {
         get { return stringData[1]; }
         set { stringData[1] = value; }
     }

     public MyService() { // constructor }

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

(note that the [WebMethod] attribute is used by, but not specific to, WSRF.NET). These attributes form the core of the WSRF.NET programming model (see [44] for a complete list of available attributes). [Resource] is used to mark data members that are part of a stateful resource manipulated by this service. In other words, the resources of a service are defined by the set of data members labeled with the [Resource] attribute. For example, the resource defined for the service in Fig. 2 consists of a string array and an integer.[1] Accordingly, the actual state of a resource is a set of values for these members. Since multiple WS-Resources can share the same Web service, i.e. multiple stateful resources can be exposed as WS-Resources by the same service, there can be multiple instances of each resource "type." In other words, the data types of all members labeled with [Resource] define a sort of resource schema for the service. All resources associated with the service will consist of a set of values corresponding to the types in the schema. Creating a new resource involves initializing a new set of values and placing them in a persistent store indexed under a new resource name. In WSRF.NET, the resource name is a WS-Addressing EndpointReference containing both the service's URL and a unique GUID.

WSRF defines the idea that the actual state of the resource (stored on the service side) is potentially different than the "view" of the resource state that is exposed to clients. The client's view of the state is therefore a projection or transform of the actual state into a format which is valid with respect to the service's resource schema. In WSRF, this transformed state is called the Resource Property document (and individual elements of that document are called Resource Properties). The schema for the Resource Property document is part of the service's WSDL document. In WSRF.NET, service authors define this schema using the [ResourceProperty] attribute. This attribute can be placed on either .NET property functions or on data members themselves. The effect

[1] A single data member may have multiple associated attributes, as shown by the integer "v" having both [Resource] and [ResourceProperty] attributes

of placing [ResourceProperty] on a .NET property depends on the definition of that .NET property. If the .NET property defines a "getter" function, that property will be exposed to clients as a ResourceProperty that can be retrieved by any of the WS-ResourceProperty specification's functions (GetResourceProperty, GetMultipleResourceProperty or QueryResourceProperty). Similarly, if the .NET property defines a "setter," the ResourceProperty exposed to the client will be modifiable using the SetResourceProperty function. In Fig. 2, Property1 has a "getter," while Property2 has both a "getter" and a "setter." .NET properties provide a convenient way of transforming the internal state of a programmatic object into a form for consumption outside the object and the [ResourceProperty] attribute leverages this concept, familiar to .NET programmers. Presumably, each [ResourceProperty]-annotated .NET property will base its return value on some data member(s) labeled as [Resource]. However, [ResourceProperty] can be used to compute dynamic values that are not based on stored state (e.g. the current time). While [ResourceProperty] attributes most commonly are used on .NET properties, they can also be placed directly on data members (as shown on the data member "v" in Fig. 2). This is a shorthand way of specifying that no transform should be used when exposing this value as a Resource Property. In other words, the value stored for the resource "v" can be directly accessed by clients.

In addition to defining the concept of stateful resources, and the exposed view of those resources based on ResourceProperties, WSRF defines a collection of messages for querying, manipulating, and discovering resources. Additionally, WSN defines a number of messages for publish/subscribe messaging. Most messages are defined in pairs, a request message made to a WS-Resource, and the response message it will generate. WSRF places each messages pair definition within a separate WSDL port type definition (that also contains a Resource Property document definition). WSRF.NET makes it easy for service programmers to develop services that respond to these messages via the [WSRFPortType] attribute.

In ASP.NET, programmers create Web services that receive and respond to messages by writing functions. ASP.NET deserializes received messages and turns them into function invocations. Function results are then serialized into response messages. WSRF.NET libraries provide functions that follow the request/response message patterns for all WSRF and WSN defined messages. These functions can be included in a Web service (and hence the ability to respond to these messages) by placing the [WSRFPortType] attribute on the Web service class and parameterizing it with the type of WSRF.NET object that implements the desired functions. For example, the service in Fig. 2 will contain the GetResourceProperty port type (meaning it will respond to the GetResourceProperty message), the ScheduledTerminationPortType (meaning it will respond to the SetTerminationTime message) and the NotificationProducer port type (meaning it will respond to the Subscribe message). In addition, the service will also expose any ResourceProperties defined by those port types.

At this point WSRF's "aggregation model" becomes important. In WSRF, a WS-Resource may respond to messages defined in many different port types. However, the WS-I Basic Profile does not define how multiple port types on a single Web service are to be interpreted. WSRF therefore mandates a "cut and paste" model of aggregation so that WS-Resources end up exposing only a single port type. This means that the WSDL document for a service must contain a single port type that defines all messages that the service supports (each message having been "cut and pasted" from its original port type definition). In addition, any Resource Properties defined in the original set of port types must be combined into a single Resource Property document that is to be included in the final WSDL document. For example, the WS-ResourceLifetime specification defines a ScheduledTermination port type. This port type defines the SetTermination request/response messages and two Resource Properties called TerminationTime and CurrentTime. The WS-ResourceProperties specification defines a QueryResourceProperties port type that includes definitions of both the QueryResourceProperties messages and the QueryExpressionDialect Resource Property. A WS-Resource that supports the ScheduledTermination and QueryResourceProperties port types will actually expose a single port type that responds to SetTerminationTime and QueryResource Properties messages. In addition, that WS-Resource's Resource Property document will contain TerminationTime, CurrentTime and QueryExpressionDialect elements. In WSRF.NET, this "cut and paste" aggregation is handled automatically by tooling discussed below.

3.2 WSRF.NET Tooling

Integration with Visual Studio is an important part of making the WSRF.NET programming experience easy. WSRF.NET includes tools to allow the Visual Studio programmer to automatically build and deploy their resource-oriented service just as they would any other Web service. These tools, which run after the Web service code is compiled, rewrite the Web service to (a) include functions that implement the methods defined in the port types imported with [WSRFPortType] and (b) add code to load/store appropriate [Resource] annotated data members in a configured database. In addition, a WSDL document is generated for the service that aggregates the messages and Resource Properties defined in the Web service class (and included port types) into a single document.

The main tool that performs these operations is called the Port Type Aggregator (PTA). In Visual Studio, metadata can be associated with any file. WSRF.NET defines a Boolean metadata tag for Web service (.asmx) files called "WSRF." If this tag is set to "true," the PTA is automatically run as a post-build step for that Web service. The PTA works as follows. Recall that the .NET attributes of any object can be read through .NET's reflection API. The PTA uses reflection on the compiled DLL file for the Web service to find any attributes used by WSRF.NET. It then composes a new Web service class based on what was found. First, this class must include any methods defined in any port types included by the [WSRFPortType] attribute. This is done by adding well-known functions to the new class (with appropriate attributes, e.g. [WebMethod]) so that the ASP. NET infrastructure will expose these methods from the service. If the port type being imported is one defined by WSRF or WSN, the function is essentially a wrapper for code in the WSRF.NET Service Library. If the port type is one written by the service author (any Web service can be treated as a port type and imported into another service – WSRF.NET simply aggregates their functions together), the function will wrap service author libraries. This function wrapping is necessary because each ASP. NET Web service is a single class whose methods define the messages the service can process. In order to expose methods in other libraries, a wrapper must exist in the exposing Web service. Next, code is added to each function, whether "imported" by [WSRFPortType] or not, to retrieve resource state at the beginning of the method and save resource state at the end of the method. These loading and saving calls, implemented by WSRF.NET libraries modify each function's control flow so that it is "load resource state, perform function computation, save resource state." The new, modified Web service class is then written to disk, compiled, and deployed by modifying the original Web service .asmx file to load the new service DLL when the service is accessed.

Service WSDL generation leverages the WSDL generation code of the .NET framework. However, recall that WSRF defines a <types> section for a service's WSDL that contains the schema for the service's Resource Property document. Since this section is not generated by the .NET framework, the PTA must generate it. The PTA assembles the schema from the types of the data members annotated as [ResourceProperty]s, found by reflecting on the Web service DLL for the [ResourceProperty] attribute. Note that the actual mapping of Resource Property names (which appear in the Resource Property document and are used by clients in, for example, a GetResourceProperty call) to entities in the Web service code (either of data members or .NET properties) is done when the Web service class is first loaded into memory and this information is then cached for subsequent accesses.

4 Building Resource-Oriented Systems: How WSRF.NET Adds Value over .NET

Resource-oriented systems are different than Service-oriented systems because resource-oriented systems treat state (or stateful resources) as an architectural concern rather than an application-specific one. As such, current Web service programming infrastructures, such as Microsoft's .NET Web services, do not fully address the needs of resource-oriented system designers. WSRF.NET provides value over standard Web services by providing (1) standardized interactions for managing stateful resources (as defined by WSRF), (2) a programming model for managing state in Web services that insulates the programmer from the details of the backend store and (3) libraries that automate common tasks in resource-oriented systems. While this section argues how WSRF.NET provides value over standard Web services, this is not a debate over whether Web services should be stateless or stateful at the implementation level. We believe that almost all services will contain internal state, and Microsoft provides several mechanisms for Web service programmers to save and recall state within services. In this section, we compare state management in WSRF.NET with state management using "vanilla" .NET Web services augmented with either in-web-service-process-memory, external in-memory session storage (e.g. IIS Sessions or the ASP.NET State Service) or application specific database code. In Section 5, we discuss the performance of WSRF.NET against these alternatives.

4.1 WSRF.NET's Compliance with Emerging WSRF / WSN-based Systems

One of the most important advantages that WSRF.NET provides is an easy way for service authors to develop services that comply with the WSRF and WSN specification families. This allows service authors to interact with the emerging WSRF-compliant services that others are building. Fourteen companies, including IBM, Intel, HP and Oracle, participated in the Technical Committee (TC). The WSN specifications have a similarly strong commercial backing, such that it is easy to imagine products supporting WSRF and WSN being released. An implementation of the Web Services Distributed Management [45] standard, which relies on WSRF, will likely be the first such product to appear. The most widely deployed Grid software, Globus Toolkit [15] version 4 (GT4), also uses WSRF and WSN and a number of open source implementations of the WSRF/WSN specifications have emerged [28]. Services based on WSRF.NET can interoperate with any of these other commercial or academic services that speak WSRF and/or WSN. Note that WSRF is not the only specification for resource-oriented services. WS-Management [6] has recently been proposed as an alternative solution to many of the same issues. While we defer an in-depth discussion of the relationship between WSRF.NET and WS-Management until Section 6, we note that, as of today, there exists a single implementation [31] of WS-Management, while at least five independent implementations of WSRF/WSN exist for various platforms [28].

4.2 WSRF.NET's Resource-Oriented Programming Environment

Another important advantage of WSRF.NET derives from its attribute-based programming model. The interface between Web service and resource state is extremely important because almost all interactions between a client and a resource-oriented service result in interactions between that Web service and some resource state. Note that in this context, the word "interface" refers to the way in which the programmer of a Web service can access/manipulate a stateful resource, and not necessarily to a literal API. WSRF.NET's attribute-based interface provides not only easy access to resource state, but also masks the complexity of many of the state management tasks (e.g. loading, saving, querying). We argue that to effectively program any resource-oriented system, the programming environment (the combination of programming interface and underlying resource management system) should provide: easy access to resource state from Web service code; the ability to load/store multiple resources of a given "type" (i.e. with a given schema) ; and easy mechanism to define new resources when programming new Web services. In the remainder of this section we discuss how these properties are exhibited by WSRF.NET and how they are *not* by any of the "pure Web services" alternatives.

First, in WSRF.NET, resource state is accessed through typed objects that are member variables of the Web service class. State is automatically loaded into these data members and so access to resource state is as easy as accessing the value of a variable. This stands in contrast to any of the in-memory storage mechanisms. These include static variables used in Web service code (allowing resource state to be stored in the Web service process itself), IIS Sessions (which stores data in the IIS process) and the ASP.NET State Service (which stores data in a separate "state server" process). Each of these provides (or can provide) a hashtable that a running Web service can access and use to store resource state. This "hashtable interface" to a resource is typeless, i.e. the hashtable stores generic "objects" (instances of System.Object in C#, for example). Although the "object" retrieved from the hashtable may be immediately cast to a specific type (and from then on provide the benefits of typed programming), the ASP.NET state service places the burden on the Web service programmer to perform this cast and to ensure that only objects of that type are actually stored in the hashtable.

A second desirable property for resource-oriented systems is the ability to have multiple resources (each defined by the same resource schema) accessed via the same Web service. Recall that a WS-Resource is the combination of a stateful resource and a Web service and it is common practice for multiple WS-Resources to use the same Web service. In fact, one of the issues with OGSI was the mandate that each resource be accessed through a *separate* Web service. In WSRF.NET, the collection of types of the variables labeled with the [Resource] attribute define the resource schema. Independent copies of the values of these variables therefore represent independent resources. WSRF.NET automatically creates new resource instances, and loads/stores previously created instanced based on the <ReferenceProperties> element of the WS-Resource's EPR. This allows the <Address> element of the EPR, i.e. the URL of the Web service, to remain the same for different WS-Resources and hence the same Web service to serve multiple resources. The fact that this automatic management of resource state occurs is masked from the Web service programmer – they need only place the [Resource] attribute on appropriate variables and WSRF.NET handles the rest. Compare this with using static member variables of the Web service class to store state. The "static variable" approach is less appealing because although static variables can provide a typed interface to resource state (as does WSRF.NET), simple static types do not allow multiple instances of the resource to be stored for the same Web service. In other words, a static integer is single valued, where a [Resource]-annotated integer will contain different values based on the value of the EPR used to access the WS-Resource. Although static arrays could be used (with each array element holding the value of a different resource), in the .NET Framework version 1.1, only fixed length arrays are typed. In other words, the maximum number of possible resources must be defined at compile time in order to have the benefit of a typed interface. Static hashtables or ArrayLists (dynamically sized arrays) could be used to store multiple resource instances, but these too provide an untyped interface.

Managing multiple resource instances per Web service implies the ability to find the one (or ones) relevant to a service invocation and make them accessible from the Web service code. WSRF.NET uses automatically generated XPath [14] and XQuery [9] queries, where as IIS Sessions and the ASP.NET State Service find resources based on a session ID included in the invocation message. Other methods require the web service programmer to manually find the appropriate resource. In addition, in practice, programmers create some operations that do not reference a single resource, but are meant to effect multiple resources. WSRF.NET can use the query capability of the underlying database to find sets of resources matching many different criteria. When using IIS Session or the ASP.NET State Service, a web service only has access to one session at a time and therefore cannot effect multiple resources with a single invocation.

A final property of a resource-oriented system is the ease with which new resources can be coded when programming a new Web service. In other words, programming a new WS-Resource should involve as little as possible beyond programming a new Web service. In WSRF.NET, creating a new "type" of stateful resource (i.e. a new resource schema) is as easy as applying the [Resource] attribute to appropriate member variables of the new service. Compare this with managing resource state with service-specific database code. Custom database code could be written for each Web service to save/load resource state. For a database based on tables (e.g. an SQL database), although many layouts are possible, it is easy to imagine using a separate table for each service. The types of all the columns of this table define the resource schema and provide the service programmer with typed access to resource state. Creating a new instance of the resource involves creating a new row in the table. However, each new service will require creating a new table to match the new resource schema. This, in turn, requires the Web service author to have access to the underlying database (linking the roles of Web service programmer and database administrator) as well as an understanding of database-specific table creation mechanisms. Of course, resources with many different schemas could be stored in the same table if they were stored as binary blobs, but this impedes the service author's ability to query the table for resources whose state matches particular parameters. XML databases can overcome the need to create a new table for each new type of resource because XML documents, by their nature, need not assume a structure for the data they contain. This allows them to contain the serialized state of any resource and, in turn, multiple services to use the same XML database as a backend store. However, the result of querying an XML database is typically an XML document. This means the service author must write additional code to deserialize the document and make its values available. This is similar to (in fact more complex than) the problem with the hashtable interface, it works, but it requires extra effort to get at resource state as typed quantities and to ensure that only proper entities (those matching the resource schema) are stored. WSRF.NET automatically deserializes the XML documents it retrieves from its backend database and makes the values available through data members of the Web service. No additional code must be written by the Web service author.

In addition to not meeting the properties mentioned above, none of the alternative stateful resource mechanisms discussed consider resources that are not pure data resources. While it is common for a stateful resource to be data, WSRF.NET easily allows other notions of resource. For example, WSRF.NET contains a process forking system that interfaces with running Windows processes. Using this system, Web service authors can create resources that are processes (i.e. WS-Resources which are web addressable representations of executing processes) simply by placing the [Resource] attribute on a member variable of WSRF. NET's Process type. In addition, WSRF.NET defines the IResource interface which provides an extensible system for accessing resources. Any Web service member variable that implements the IResource interface (and is annotated with the [Resource] attribute) will automatically be loaded, stored, created and destroy through function calls on the IResource interface by WSRF.NET. By providing custom implementations of these functions, almost any resource can be addressed by WSRF.NET – including legacy resources (e.g. pre-existing databases).

4.3 WSRF.NET's Automation of Common Tasks

In addition to providing a programming model for resource-oriented services, WSRF.NET provides class libraries that automate certain common tasks for services and clients. The first of these is notification. Notification, via WS-Notification, is well supported in WSRF.NET. On the service side, all a programmer needs to do to send a notification message is to declare a ProducibleTopic object, using an XML qualified name that will be the topic name [43], and then call the ProducibleTopic's notify() method, passing an object that will be serialized as the body of the notification message. WSRF.NET will automatically communicate with a subscription manager service to determine the Endpoint information for all clients that have subscribed to the given topic and send the message to each. WSRF.NET supports topics that follow both the "simple" and "concrete" topic naming conventions defined in WS-Topics. Receiving a notification message on the client side is similarly easy. WSRF.NET provides a NotificationListener object which can receive notifications over HTTP without the use of IIS, Microsoft's Web server. In other words, arbitrary client binaries (and not just Web services sitting behind the web server) can receive notifications. In order to have a client receive notifications, the first step is often to subscribe the client to the desired topic (s) on the service(s) that will generate the notifications. This is done using the standard WS-Notification Subscribe message [21] and can be sent using WSRF. NET's NotificationProducerProxy. Now the client must create a NotificationListener and describes the topics it should listen for (note that even if subscriptions are not being used, the client can use the NotificationListener to filter notification message). Topics of interest are described by topic expressions. The meaning of a topic expression depends on the topic space of the notifying service. For example, the topic expression "A" could mean the listener is interested just in messages about topic "A," or, if the topic space is hierarchical, "A" could mean that the client wishes to receive messages about topics "A/B" and "A/C." Creating a topic expression for a WSRF.NET client is simply a matter of creating a TopicExpression object with the appropriate XML qualified name. Then an event handler is registered with the TopicExpression that will be raised whenever a notification on the specified topic occurs.

A second common operation for a service is to interact with a running process. These could either be processes that are being presented to clients as WS-Resources or they could be worker processes used by the Web service to perform its task. Typical operations include starting a process, possibly terminating a process before it exits normally, knowing when the process completes and determining the state of the process (either when running, e.g. PID, or after termination, e.g. exit code). While the .NET Process object can interface a Web service to a currently running process, WSRF.NET ProcSpawn service can provide these capabilities for processes which have also exited. In other words, the .NET Process object cannot bind to any process which has completed because the operating system automatically cleans up that process' information. The WSRF.NET ProcSpawn service, on the other hand, is a Windows service (not a Web service) that installs along with WSRF.NET. This service is used by WSRF.NET Web services to launch and monitor processes on a single machine. Because the ProcSpawn service remains in memory (as opposed to the Web service instance which is dynamically created and then destroyed for every invocation), it can continue to hold Windows handles to processes and therefore access information about them even after they terminate. In addition, the ProcSpawn service can notify (using WS-Notification) a Web service when a given process completes. The interface to the ProcSpawn service is via the WSRF.NET ProcessHandle object. By placing the [Resource] attribute on an object of this type, a WSRF.NET service can use the ProcSpawn service to automatically create a Web service accessible binding to the running (or exited) process whenever a method invocation occurs. Therefore, it is easy to use this capability to expose a process as (or as part of) a WS-Resource.

Another useful mechanism provided by WSRF.NET is the ability for services to easily initialize new WS-Resources. While the WSRF specifications do not define any particular message by which a *client* creates a new WS-Resource (this was thought to be too application specific to be standardized), WSRF.NET does provide methods that the *service* author can use to interact with the backend resource store. Each port type in a WSRF.NET service defines an InitResource() method which is used to initialize all [Resource] annotated member variables of that port type. The base type for all WSRF.NET services defines a create() function which will invoke the InitResource() method on all port types included in a service using the [WSRFPortType] attribute. The effect of calling the create() method from within service code is that initial values for all [Resource] annotated members are set. These values will then be stored to the backend data store when the current client invocation completes. So, while the service author is free to choose the exact signature for the "create" function that is exposed to clients, internally, all that function needs to do is to call the create() method. In addition, WSRF.NET provides an implementation of a port type called the GCGResourceFactoryPortType. Placing the attribute [WSRFPortType(typeof(GCGResourceFactoryPortType))] on a service class automatically provides a web accessible Create() method that clients may call. When using this port type, each other port type in the WSRF.NET service can be annotated with a [Resource InitializerType] attribute. This attribute defines the data type of an object used to initialize that port type. In other words, the data type passed as a parameter to that port type's InitResource() method. When a client calls the GCGResourceFactoryPortType's Create() method, they pass in a document containing serialized versions of all the types specified in a service's [ResourceInitializerType] attributes. WSRF.NET will automatically deserialize these and pass them to the correct InitResource() method. This resource creation mechanism is both simple and powerful. In general, it can be used to pass arbitrary data to a service that it can use in creating a new resource. In its simplest form (where no client data is needed), all that is required is for a single [WSRFPortType] attribute to be placed on the Web service class. The benefit of providing this method (even though it is not defined by WSRF) is that it allows the service author to deal with programming language types instead of XML messages. We are currently developing tools to allow clients to automatically generate the document that is passed to the GCGResourceFactoryPortType based on the attributes of the service.

5 Evaluation

To evaluate WSRF.NET, we consider the fundamental issue to be: to what extent does the programmability (both programming-language abstractions and compile-time tooling) and improved run-time persistence/management of state of WSRF.NET outweigh the run-time overhead incurred to support such abstractions? Unfortunately, concepts such as programmability are difficult to quantitatively evaluate, so this equation does not reduce to a simple comparison of two similar metrics. Instead, we have used Section 4 to (in part) qualitatively argue that WSRF.NET makes it easy to implement resource-oriented services as compared to alternative resource storage/maintenance mechanisms.

In this section, we quantitatively evaluate the run-time performance of using WSRF.NET. While this information cannot be used to answer the question posed in the first sentence of this section, we argue that the core WSRF.NET primitives incur negligible overhead compared to typical domain-specific resource manipulation operations.

Additionally, an XML database is typically configured as WSRF.NET's back-end resource store while several of the alternate storage mechanisms, static hashtables, IIS session state and the ASP.NET State Service, store data in RAM. Fundamentally, this means WSRF.NET provides a level of data persistence that in-memory techniques do not, but it also means that a performance comparison between WSRF.NET and these alternatives involves a comparison between RAM and disk I/O. While there is no quantitative measure of persistence, we take it as an axiom that persistence is crucial for any long-lived, widely-distributed system. In other words, hardware failure in a large, long-running system will occur and simple RAM-based storage is insufficient.

To evaluate the run-time overhead of WSRF.NET and the alternatives, a series of tests were run involving a client making an invocation on a Web service. Both client and server ran on the same machine and the times reported are from the time the client sends the invocation message to the time the client receives the response from the service. The tests are:

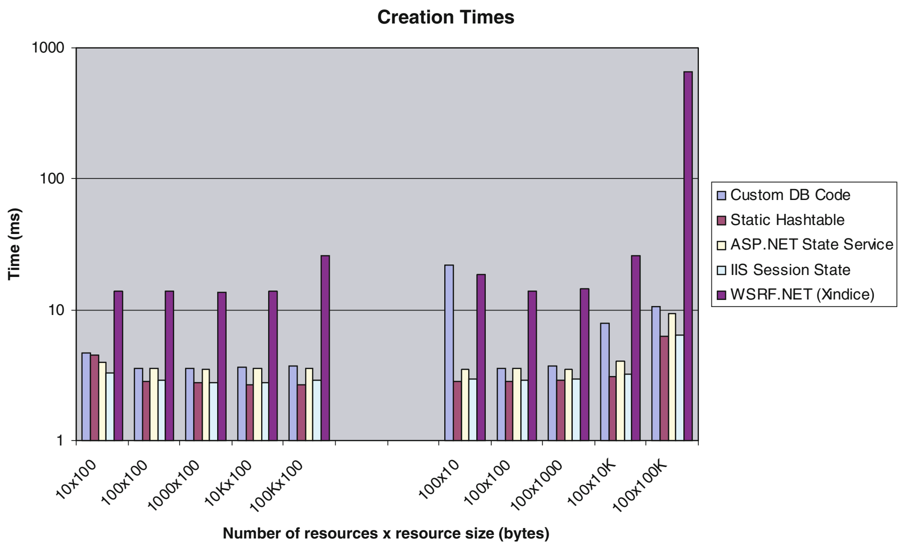
- Create test – the client invokes a method on the service that creates a new instance of a resource and saves it to the back-end store. The EPR for the new WS-Resource is returned to the client.
- Query test – the client accesses a WS-Resource causing the system to perform a query to find the appropriate resource in its store and load the state into memory (if appropriate). The service returns the first byte of the resource state.
- Read test – the client accesses a WS-Resource as in the query test, but this test is run after the query test allowing the systems to show their caching behavior (if any). Return value is the same.
- Write test – the client accesses a WS-Resource and modifies its state (randomly). This causes the system to have to write the new resource value to the backend store. The service sends an ACK to the client.
- Delete test – the client asks the service to destroy a particular WS-Resource. This involves the service finding the appropriate resource (via a query) and then removing it from the backend store.

The five types of tests were run for resources sizes of increasing orders of magnitude from 10 B to 100 kB with 100 resources at each size. Another battery of the five tests was run for increasing orders of magnitude of number of resources from 10 to 100 K resources, each of 100 B. The experiments were run on dual AMD Opteron 240 (1.4 GHz) processor machines, with 2 GB RAM, running Windows 2003 Enterprise Edition (Service Pack 1). Numbers reported for all tests except Create and Delete are averages from 1,000 invocations. Create and Delete times are averages from the creation or deletion of the number of resources uses in the test. For example, in tests with 100 resources, the times are averages over 100 operations; in tests with 100 K resources, the times are averages over 100 K operations. The configuration of the various systems is as follows. WSRF.NET was configured to use the Xindice database. The ASP.NET State Service was configured to store state in the ASP.NET State Service process. The "Custom DB Code" used SQLServer Express Edition 2005 and SQL statements to load/store non-XML data. The "Static Hashtable" configuration involves a Web service which declares a static Hashtable object in the Web service class and therefore, even though ASP.NET destroys the instance of the Web service after each client invocation, the static data remains in memory. The "IIS Session State" configuration uses IIS sessions to make a hashtable (managed by IIS) accessible to the Web service. The contents of this hashtable are based on a session ID included with the client request.

Figures 3, 4, 5, 6, 7 show the results of the Create, Query, Read, Write and Delete tests. Note that all graphs are log scale.

From Fig. 3, it can be seen that all systems retain their "create performance" for small resources as the number of resources increase, but begin to take exponentially more time as the size of the resources increases. The difference between WSRF.NET and all but the "Custom DB Code" case is due to the difference between in-memory hashtables and databases writing to disk. The difference between WSRF.

Fig. 3 Timing for the Create test



NET and the Custom DB Code configuration is due to the difference between serializing and writing XML data and writing data to an SQL table. While the Custom DB Code is faster, it has the limitations discussed in Section 4, namely that new code and new database tables must be created for each new Web service and resource type. The larger than expected bars for WSRF.NET and the Custom DB Code for the 100×10-B case appear to be caused by a small number of anomalously high creation times. In the Query Test of Fig. 4 we can see that WSRF.NET's cost is impacted more by large numbers of resources than by large resource size. The alternatives retain relatively constant performance due to the constant time nature of the hashtable lookup that can be used to find the resource reference by an invocation. The fact that the alternatives perform somewhat worse for large size resources is expected since increased time is needed to transfer the resource state to the Web service from an external process (such as the ASP. NET State Service) or from the SQL database. The Read Test of Fig. 5 shows that WSRF.NET's performance improves dramatically for WS-Resources accesses after the first (due to WSRF.NET's write-through cache). The difference in performance with the alternatives is due to the fact that, after the invocation, the resource must be serialized to check if it has changed with respect to the cached copy. The

Fig. 4 Timing for the Query test

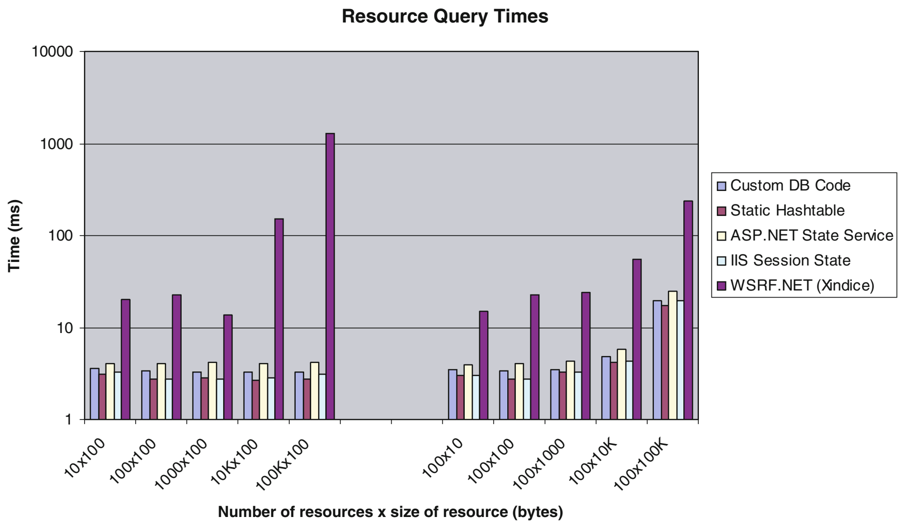
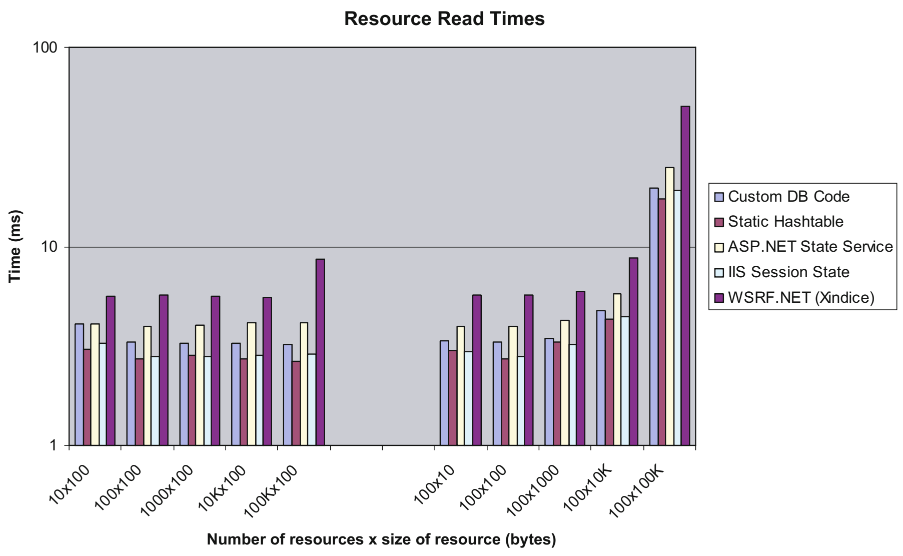


Fig. 5 Timing for the Read test



Write Test of Fig. 6 shows similar results to the Read Test, with both WSRF.NET and the Custom DB Code performing slower as the size of resources increases. This is expected since both of these systems must write data to the file system. Finally, the Delete Test of Fig. 7 shows performance for WSRF.NET that is similar to the Query Test. This is because even if the resource is cached, a query must be done on the database to find the XML document that must be removed.

While WSRF.NET's overhead is greater than the other alternative technologies, we believe that it typically represents only a small portion of the time that would be used by the domain science of a real application. Consider a medical data mining application which looks through patient records to find suitable candidates for a new drug treatment study. If these records are being exposed as WS-Resources, the application might access the resource (read it into memory), compute how well the patient matches the studies criteria and then, after examining all the records, go back and write information about the quartile into which the patient fell into their record. Assume the data mining application runs for 10 min and accesses 100 resources. We can model WSRF. NET's overhead for this as 100 queries (to load

Fig. 6 Timing for Write test

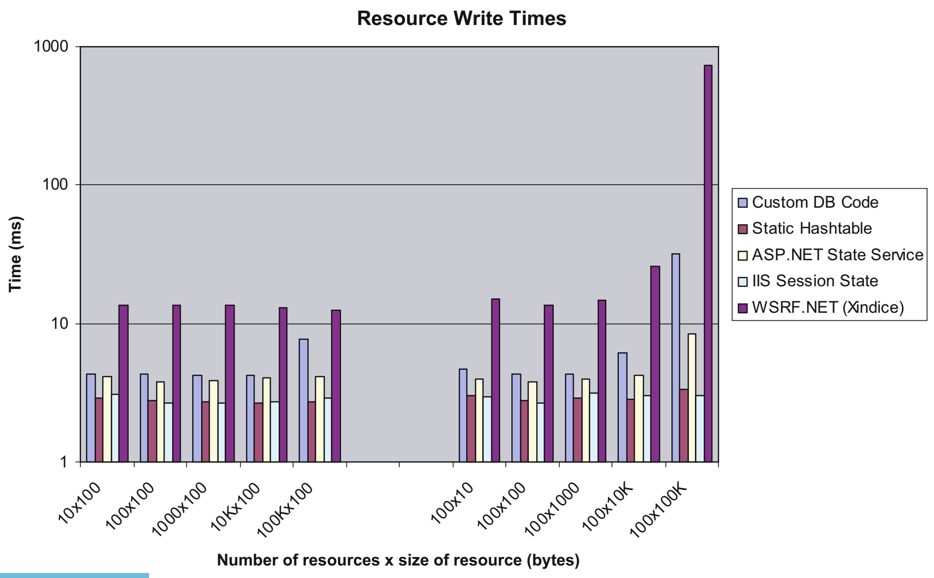
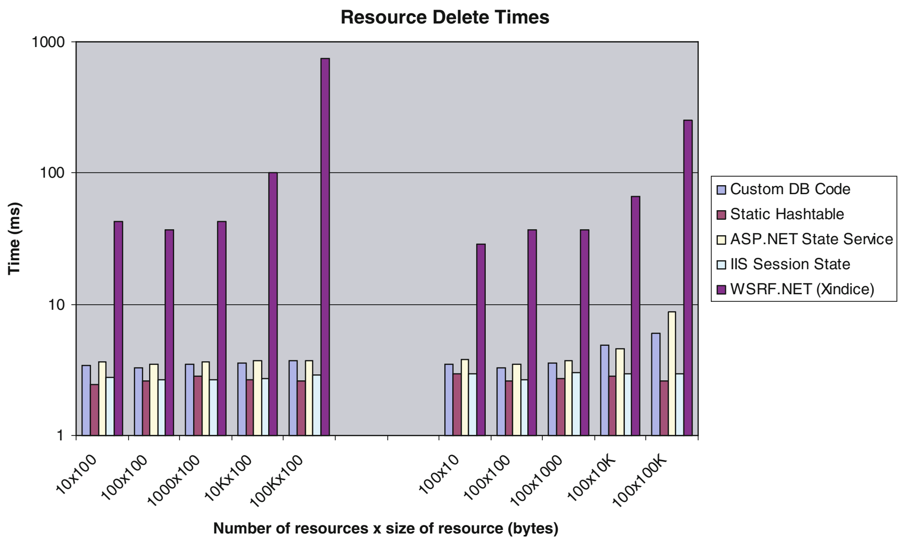


Fig. 7 Timing for Delete test



resources) and 100 writes. For resource size (patient record size) of 100 B, resource queries take 13.7 ms on average and writes take 13.6 ms on average, as shown in Fig. 4 and Fig. 6. Therefore, the total overhead would be (100 * (13.7+13.6 ms)), assuming that no resources are accessed multiple times and therefore not read from the cache. This means WSRF.NET's overhead for a 10-min run would be (100 * (13.7+13.6 ms)) / (60 s/min *10 min), or 0.46%. Table 1 shows the percentage of WSRF.NET overhead for all test scenarios in which there are at least 100 resources. The top row shows the number of resources × the size of resources (in bytes). The bottom row shows the percentage of a 10-min run that is occupied by WSRF.NET.

We can see that WSRF.NET's overhead is very small for all but the cases with large numbers of resources or when the resources are of large size. This is particularly relevant when considering that the other mechanisms for state management do not provide WSRF.NET's persistence of state, nor WSRF.NET's ease of programming. Obviously as application run times grow larger (many jobs run on today's computational Grids consume hours of days of CPU time), WSRF.NET's overhead drops considerably. In fact, it is precisely for such long-running jobs that WSRF.NET's database-backed resource management system is particularly important; days of work should not be lost due to a simple power failure.

6 Conclusion

WSRF.NET provides substantial value in the development of resource-oriented systems. WSRF.NET augments the standard Microsoft .NET Web services platform with WSRF-compliance, a resource-oriented programming model and set of useful class libraries. While WSRF.NET provides the ability to build resource-oriented systems on .NET, it should be noted that there are alternatives systems for other platforms. Most notably, the Globus Toolkit version 4 (GT4) [19] provides both Java and C toolkits for developing WSRF/WSN-compliant services. WSRF::Lite [48] and pyGridWare [40] provide Perl and Python programming environments for WSRF/WSN. While many of these systems can run on Windows (frequently with reduced functionality as compared to their Linux implementations), none leverage the .NET

Table 1 Percent overhead for WSRF.NET mechanisms in 10 minute application

WSRF.NET overhead							
100×100	1,000×100	10K×100	100K×100	100×10	100×1000	100×10 K	100×100 K
0.6%	0.45%	2.7%	21.7%	0.49%	0.64%	1.35%	16.1%

framework. A full comparison (both qualitative and quantitative) of these systems and WSRF.NET can be found in [28].

Interest in resource-oriented systems is growing. In 2004, Microsoft, Intel, Sun and others introduced WS-Management [6]. WS-Management is itself a set of conventions on using several other specifications including WS-Transfer [3], WS-Eventing [11], WS-Enumeration [2] and WS-Addressing [24]. WS-Transfer defines Get, Put, Create and Delete messages for resources (i.e. it defines the CRUD pattern for Web services). WS-Enumeration provides a mechanism for clients to iterate through data sets managed by services. WS-Eventing defines an asynchronous, topic-based, messaging system and WS-Addressing presents a standardized way of naming resources. While on the surface, WSRF and WS-Management have approximately the same functionality, there are differences. For example, WSRF has no analog for WS-Management's WS-Enumeration. WS-Management, on the other hand, has no analog for the lease-based lifetime defined by WSRF. WS-Management defines a standard "Create" message, while WSRF believes such a message to be too application specific to be meaningfully standardized. WSRF defines a standard mechanism for clients to get the exposed schema of a resource, via the Resource Property document. WS-Management contains no standard mechanism for accessing resource schema, believing that that is an application-level concern. While it is difficult to definitively determine that one set of specifications is superior to the other, some prelimary comparison work has been done between WSRF/WSN and WS-Transfer/WS-Eventing [27].

Currently, the relationship between WS-Management (and its constituent specification) and WSRF/WSN is unclear. One possibility is that in the future the two standards will merge, taking the best of both and providing a unified structure for the Web services industry. Another possibility is that both standards will exist simultaneously and different systems will utilize the different technologies. While many in the Grid community are working on the former, the later is not completely outside the realm of possibility. WSRF/WSN has several implementations that use a variety of programming languages/environments and run on a variety of OSs/architectures. While only one implementation of WS-Management exists [31], Intel has said that they will embed WS-Management in their hardware to make it easier to manage their devices. This may provide strong impetus for others to build systems using WS-Management.

Regardless of which set of specifications gains the mostly widespread use, we believe there is a place for the technology of WSRF.NET. Of course, a toolkit targeted as a different set of specifications would provide compliance with those new specifications (not WSRF), but WSRF.NET's programming model and class libraries should be advantageous for programming any resource-oriented system. One of the main future work tasks for the WSRF.NET project is to take the lessons learned in developing WSRF.NET and apply them to either a toolkit that provides the functionality of WS-Management, or one that is agnostic of WSRF/WS-Management (one that, for example, has a compile-time switch that determines whether the resulting service will comply with WSRF or WS-Management). Another future direction is continuing to expose upcoming Microsoft technologies in a way that is convenient for programming resource-oriented systems. The Windows Communication Foundation (WCF), previously code-named "Indigo," and the next generation of the Windows operating system, code-named "Longhorn" represent the evolution of both Windows and the .NET platform. Since both will be part of the Microsoft Web services platform, it is natural that WSRF.NET should embrace both. Another important piece of future work, which has already begun, is to provide more Grid-specific functionality on top of WSRF.NET. We have begun developing services that comply with both OGSA and the Globus protocols (e.g. GRAM, GridFTP).

WSRF.NET has a broad user community. To date, downloads from more than 1,440 unique IP addresses, representing 44 countries and every continent, have been recorded. WSRF.NET has also been used as the basis for a number of other projects, such as the UVA Campus Grid [26], OGSA Byte-IO Service [12], OGSA Basic Execution Service (BES) [8], and the Akogrimo [1] and GRASP projects [23]. WSRF.NET is providing clear, measurable value to the resource-oriented systems community.

Acknowledgements This research was 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), the San Diego Supercomputing Center and Microsoft Corporation.

References

1. Akogrimo Project: Access to Knowledge through the Grid in a Mobile World. http://www.mobilegrids.org/. Cited December (2005)
2. Alexander, J., Box, D., Cabrera, L., Chappell, D., Daniels, G., Geller, A., Kaler, C., Orchard, D., Sedukhin, I., Simek, M., Theimer, M.: Web Services Enumeration (WS-Enumeration). http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-enumeration.pdf. September (2004)
3. Alexander, J., Box, D., Cabrera, L., Chappell, D., Daniels, G., Geller, A., Janecek, R., Kaler, C., Lovering, B., Orchard, D., Schlimmer, J., Sedukhin, I., Shewchuk, J.: Web Services Transfer (WS-Transfer). http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-transfer.pdf. September (2004)
4. Apache Software Foundation: Apache Tomcat. http://tomcat.apache.org/. Cited February (2005)
5. Apache Software Foundation: Apache Xindice. http://xml.apache.org/xindice/. Cited February (2005)
6. Arora, A., Cohen, J., Davis, J., Golovinsky, E., He, J., Hines, D., McCollum, R., Milenkovic, M., Montgomery, P., Schlimmer, J., Suen, E., Tewari, V.: Web Services Management (WS-Management). http://www.intel.com/technology/manage/downloads/ws_management.pdf. Feb. (2005)
7. Ballinger, K., Ehnebuske, D., Gudgin, M., Nottingham, M., Yendluri, P.: WS-I Basic Profile 1.0. *Web Services Interoperability Organization*. http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html. April 16, (2004)
8. Basic Execution Service (BES) Working Group: Global Grid Forum. https://forge.gridforum.org/projects/ogsa-bes-wg/. November (2005)
9. Boag, S., Chamberlin, D., Fernandez, F., Florescu, D., Robie, J., Simeon, J. (eds.): XQuery 1.0: An XML Query Language. W3C. http://www.w3.org/TR/2005/CR-xquery-20051103/. November 3, (2005)
10. Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., Orchard, D.: Web Services Architecture. W3C Working Group Note. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. February (2004)
11. Box, D., Cabrera, L., Critchley, C., Curbera, F., Ferguson, D., Geller, A., Graham, S., Hull, D., Kakivaya, G., Lewis, A., Lovering, B., Mihic, M., Niblett, P., Orchard, D., Saiyed, J., Samdarshi, S., Schlimmer, J., Sedukhin, I., Shewchuk, J., Smith, B., Weerawarana, S., Wortendyke, D.: Web Services Eventing (WS-Eventing). http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf. August (2004).
12. Byte-IO Working Group. Global Grid Forum. https://forge.gridforum.org/projects/byteio-wg/. October (2005)
13. Chappell D., Liu, L. (eds): Web Services Brokered Notification 1.3 (WS-BrokeredNotification). OASIS WSN-TC. http://www.oasis-open.org/committees/download.php/13485/wsn-ws-brokered_notification-1.3-spec-pr-01.pdf. July 7 (2005)
14. Clark, J., DeRose, S. (eds): XML XPath Lanaguage (XPath) Version 1.0. *W3C*. http://www.w3.org/TR/xpath. November 16 (1999)
15. Foster, I., Kesselman, C.: Globus: a metacomputing infrastructure toolkit. Int. J. Supercomput. Appl. **11**(2), 115–128 (1997)
16. Foster, I., Kesselman, C., Nick, J., Teucke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Services Infrastructure WG – Global Grid Forum*. (2002)
17. Foster, I., Kesselman, C., Nick, J., Teucke, S.: Grid services for distributed systems integration. IEEE Computer **35**(6), 37–46 (2002)
18. Global Grid Forum. http://www.ggf.org (2005)
19. Globus Project: Globus Toolkit 4. http://www.globus.org (2005)
20. Graham, S., Treadwell, J. (eds): Web Services Resource Properties 1.2 (WS-ResourceProperties). OASIS WSRF-TC. http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-pr-02.pdf. Oct. 6 (2005)
21. Graham, S., Hull, D., Murray, B. (eds): Web Services Base Notification 1.3 (WS-BaseNotification). OASIS WSN-TC. http://www.oasis-open.org/committees/download.php/13488/wsn-ws-base_notification-1.3-spec-pr-01.pdf. July 7 (2005)
22. Graham, S., Karmarkar, A., Mischkinsky, J., Robinson, I., Sedukhin, I.: Web Services Resource 1.2 (WS-Resource). OASIS-WSRF TC. http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-pr-02.pdf. October 6 (2005)
23. GRASP Project: Grid Application Service Provision. http://eu-grasp.net/english/default.htm. January (2005)
24. Gudgin M., Hadley, M. (eds): Web Service Addressing 1.0 – Core. W3C Working Draft. http://www.w3.org/TR/2005/WD-ws-addr-core-20050331/. March 31 (2005)
25. Humphrey, M., Wasson, G.: Architectural foundations of WSRF.NET. International Journal of Web Services Research **2**(2), 83–97 (2005). April–June (2005)
26. Humphrey, M., Wasson, G.: The University of Virginia campus Grid: integrating grid technologies with the campus information infrastructure. 2005 European Grid Conference (ECG 2005), Amsterdam, The Netherlands, Feb 14–16 (2005)
27. Humphrey, M., Wasson, G., Kiryakov, Y., Park, S., Del Vecchio, D., Beekwilder, N., Gray, J.: Alternate Software Stacks for OGSA-based Grids. Proceedings of Super Computing 2005. Seattle, WA. Nov. 12–18 (2005)
28. Humphrey, M., Wasson, G., Gawor, J., Bester, J., Lang, S., Foster, I., Pickles, S., McKeown, M., Jackson, K., Boverhof, J., Rodriguez, M., Meder, S.: State and Events for Web Services: A Comparison of Five WS-Resource Framework and WS-Notification Implementations. Proceedings 14th International Symposium on High-Performance Distributed Computing (HPDC-14). Research Triangle Park, NC24–27 July (2005)
29. Humphrey, M., Wasson, G., Morgan, M., 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. (2004)
30. IBM. WebSphere: http://www-306.ibm.com/software/ websphere/. (2005)
31. Java.NET: Wiseman: A Java Implementation of WS-Management. https://wiseman.dev.java.net/. January (2005)
32. Liu, L., Meder, S. (eds.): Web Services Base Faults 1.2 (WS-BaseFaults). OASIS WSRF-TC. http://docs.oasis-open.org/wsrf/wsrf-ws_base_faults-1.2-spec-pr-02.pdf. Oct. 7, (2005)

33. Maquire, T., Snelling, D., Banks, T. (eds.): Web Services Service Groups 1.2 (WS-ServiceGroups). OASIS WSRF-TC. http://docs.oasis-open.org/wsrf/wsrf-ws_service_group-1.2-spec-pr-02.pdf. Oct. 7 (2005)
34. Microsoft. ASP.NET: http://asp.net/Default.aspx?tabindex=0&tabid=1. (2005)
35. Microsoft. SQL Server 2005: http://www.microsoft.com/sql/default.mspx. (2005)
36. Microsoft. .NET Framework.: http://www.microsoft.com/net. (2005)
37. Microsoft: Web Services Enhancements (WSE). http://msdn.microsoft.com/webservices/webservices/building/wse/. (2005)
38. Mono Project: Mono, http://www.mono-project.com/Main_Page. (2005)
39. Open Grid Services Architecture Global Grid Forum. https://forge.gridforum.org/projects/ogsa-wg. (2005)
40. pyGridWare: Python Web Services Resource Framework. http://dsd.lbl.gov/gtg/projects/pyGridWare/. (2005)
41. Srinivasan, L., Banks, T. (eds.): Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). OASIS WSRF-TC. http://docs.oasis-open.org/wsrf/wsrf-ws_resource_lifetime-1.2-spec-pr-02.pdf. Oct. 7 (2005)
42. Teucke, S., Czajkowoski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maquire, T., Sandholm, T., Snelling, D., Vanderbilt, P.: Open Grid Services Infrastructure 1.0. OGSI-WG: Global Grid Forum. http://www-unix.globus.org/tool kit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf. (2003)
43. Vanbenepe, W. (ed.): Web Services Topics 1.2 (WS-Topics). OASIS WSN-TC. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf. July 22 (2004)
44. Wasson, G.: WSRF.NET Programmer's Reference Manual. http://www.cs.virginia.edu/~gsw2c/WSRFdotNet/WSRFdotNet_programmers_reference.pdf. August 23 (2005)
45. Web Services Distributed Management (WSDM): OASIS WSDM Technical Committee. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm. (2005)
46. Web Services Resource Framework (WSRF): OASIS WSRF Technical Committee. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf. (2005)
47. Web Services Notification (WSN): OASIS WSN Technical Committee. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn. (2005)
48. WSRF::Lite - Perl Grid Services. http://www.sve.man.ac.uk/Research/AtoZ/ILCT (2005)