

OGSI.NET: OGSI-compliance on the .NET Framework

Glenn Wasson, Norm Beekwilder, Mark Morgan, and Marty Humphrey
Grid Computing Group
Computer Science Department
University of Virginia
Charlottesville, VA 22904

Abstract: *The Open Grid Service Infrastructure (OGSI) has been designed to facilitate the creation of multiple, interoperable Grid Service hosting environments, but to date only one fully OGSI-compliant hosting environment exists, Globus Toolkit version 3 (GT3). This paper describes the design and implementation of OGSI.NET, which is the second, independent hosting environment that is fully compliant with the OGSI specification. While the Microsoft suite of Visual Studio, Internet Information Services (IIS), and the Web Services Enhancements (WSE) provide a robust foundation upon which we implemented OGSI.NET, the challenges included constructing the Grid Service container, parsing OGSI WSDL (“gwsdl”), and supporting Grid Service Handles (GSHs). By describing our approach to these challenges, and by describing the unique Grid attribute programming model we are developing as part of OGSI.NET, we contribute an early evaluation of OGSI and the broader Open Grid Services Architecture (OGSA).*

1. Introduction

The goal of the Open Grid Services Architecture (OGSA) [1] is to significantly broaden the established approach of Grid Computing and tap into the emerging capabilities of Web Services, which are largely being driven by the commercial sector. To date, the core of OGSA has been the Open Grid Services Infrastructure (OGSI) [7], which essentially defines the common capabilities and interfaces of the “hosting environment” for services that adhere to the larger vision of the OGSA. OGSI defines conventions for creating, managing, and exchanging information between Grid services. To date the only hosting environment that fully complies with the OGSI specification is the Globus Toolkit version 3 (GT3) [2].

This paper describes our design and implementation of OGSI.NET, which to our knowledge is only the *second*

hosting environment that has been constructed to be fully OGSI-compliant. OGSI.NET is built upon the .NET Framework [4], which is the Web Services infrastructure of Microsoft. Generally speaking, there are two interrelated reasons why we chose to design and implement an OGSI-compliant hosting environment on .NET. First, Microsoft technologies are among the first implementations of many of the emerging Web Services specifications that are the foundation of OGSA and OGSI; we wanted to perform a critical evaluation of both the expressibility and the performance of these specifications in the context of Grid requirements. Similarly, we wanted to create a platform by which to evaluate OGSI in the context of “pure” Web Services to determine the value of OGSA and OGSI. There is starting to emerge a healthy debate in the community regarding the value of OGSI (e.g., [5]), and we wanted to have a defensible position one way or the other in this debate. Second, many people equate Grid computing and GT3, and GT3 is synonymous with OGSI, so if we as a community are going to start getting Windows-based clients and servers more involved in Grids beyond running Internet Explorer, arguably, it is necessary to have an OGSI implementation on .NET. That said, it is important to understand that we acknowledge and increasingly agree with the belief that “pure” Web Services might be enough to meet the requirements of Grids, especially if the particular Grid consists entirely of Windows machines; however, the OGSI layer and rendering is important if a heterogeneous mix of Windows and UNIX-based machines exist in the particular Grid.

The contributions of this paper are that it presents the design and implementation of the OGSI.NET hosting environment and discusses some early observations on the challenges and issues of implementing the OGSI specification. We describe the programming model we are developing to enable easier construction of Grid applications. We describe how we utilize certain Microsoft technologies to create OGSI.NET and we discuss our initial impressions on the *value* of OGSI itself through our use of OGSI.NET to construct

sample demo applications. We also discuss the future of the OGSINET hosting environment, focusing on issues of interoperability with other hosting environments such as GT3 and on how we believe future products of Microsoft will add to the value of OGSINET, which directly translates into added capabilities for Grid computing in general.

2. Basic Architecture

In many ways, the single design goal of OGSINET is to achieve OGSINET-compliance by exploiting the core technologies of Microsoft as much as possible. Simply, by creating the thinnest layer possible, we could replace current Microsoft technologies with newer versions as they become available. This is particularly important with regard to Web Services security. That is, many of the Web Services security specifications [3] do not yet have publicly-available implementations (either from Microsoft or elsewhere). We want to utilize, as easily as possible, the Microsoft SDKs as they emerge to implement these specifications. An example of such an SDK is the Microsoft Web Services Enhancements (WSE) [10].

The basic design of OGSINET is to have a container entity that “holds” all service instances running on a host. The container process consists of a collection of ApplicationDomains (or AppDomains), Microsoft’s mechanism for intra-process memory protection. Each service instance executes in its own AppDomain and there is one additional domain for the container’s logic (some dispatching and message processing functionality). The object in this final AppDomain is referred to as the dispatcher. While the current version of OGSINET (2.0) keeps each service in its own AppDomain, there is an open issue about other possible mappings for future versions of the container. For example, all instances created by a given factory could live in the same AppDomain. Having more than one service in a given AppDomain would make inter-service communication more efficient, but at a cost of some security since any errant or malicious service has direct access to the memory of all the services in its AppDomain.

A client makes a request on the OGSINET architecture by sending a message to the IIS web server. In order to support arbitrary names for grid services (as required by the OGSINET specification), OGSINET uses an ISAPI filter to intercept requests at an early stage in the IIS request chain. This filter rewrites the request so that IIS will dispatch it to OGSINET’s ASP.NET HttpHandler. This HttpHandler dispatches the request to the OGSINET container. The container process has a thread pool and each IIS request causes one of the container process’

threads to execute the dispatcher. The dispatcher determines which service instance should get the request and transfers execution of that thread to an object in the appropriate AppDomain. A diagram of the system is shown in Figure 1.

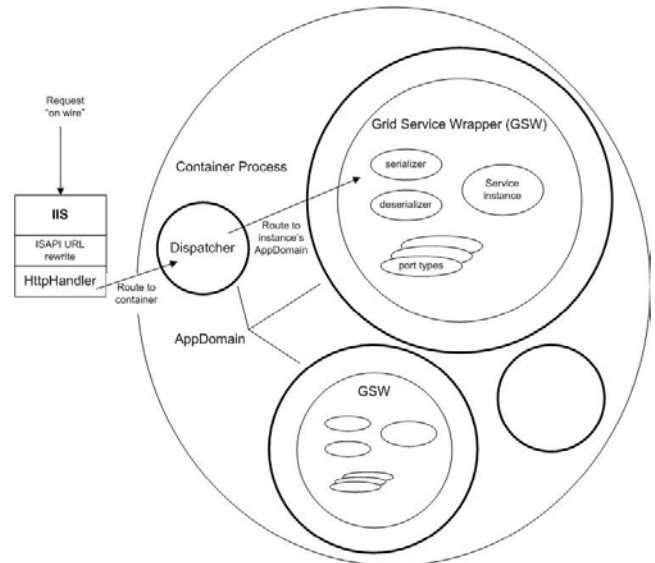


Figure 1. OGSINET Container on the .NET Platform

Inside the AppDomain, control transfers to an object called the Grid Service Wrapper (GSW). A GSW encapsulates a service instance, including the implementations of its methods and its service data. The GSW maintains lists of the port types that the service supports (both custom port types written by the service’s author and OGSINET specification port types provided with the container, e.g. grid service port type, notification source port type, etc.) and the serializers and deserializers needed for the various messaging protocols the service supports (e.g. SOAP or Microsoft Remoting). The GSW performs processing of any message specific information (e.g. SOAP headers) and deserializes the request message to get the name of the method to invoke and any parameters. Then the GSW selects the port type that implements the requested method and uses reflection to get a handle to the method. The method is invoked with any parameters and message specific data (e.g. SOAP header data) sent in the request. The data returned by the invocation is serialized by the GSW and sent back to the dispatcher as a binary array. The dispatcher sends this array back to IIS for return to the client.

3. System Elements

This section discusses the major components of OGSINET: the dispatcher, service wrappers (grid service and light-weight service), factories and message handlers.

3.1. Dispatcher

The dispatcher is the interface between the client request and the service instance that serves that request. The dispatcher's main function is to route request messages to the appropriate service instance and return the results to the client. The dispatcher contains a mapping from request Grid Service References (GSRs) [7] to AppDomains within the container. The raw request is dispatched to the Grid Service Wrapper (see section 3.2) in the requested service's AppDomain. The Grid Service Wrapper will perform the requested work and return a byte stream to the dispatcher (if appropriate) that the dispatcher will then send on to the client.

The simplicity of the dispatcher is deliberate and assists in providing concurrent access to the container because it allows requests to be independent of each other. Security is also enhanced because the dispatcher does not "process" the incoming or outgoing messages. If the dispatcher did marshal/unmarshal types from the messages, it would potentially be running user (service) code which might endanger the container by corrupting the dispatcher. By processing messages in the AppDomain of the requested service instance, only that instance is affected.

3.2. Grid Service Wrapper

The Grid Service Wrapper or GSW encapsulates the various functional units of a grid service instance. Each AppDomain in the container (except the dispatcher's) has a GSW and each GSW wraps a grid service instance. The GSW is a convenient operational unit for grid service authors because it provides:

- Pluggable, service-specific message serializers and deserializers
- Support for plugging-in port types that the service supports (including those not written by the service author, e.g. the grid service port type)
- An SDE management API
- Service policy support

The Grid Service Wrapper also provides the dispatcher with a well-known interface through which any grid service instance can be accessed. The GSW's ProcessRequest method is used to invoke any function supported by the instance (see section 4) while the ProcessQuery method is used to retrieve a service instance's WSDL from the instance itself (allowing the instance to potentially modify its WSDL to, for example, update its communication end-points).

A grid service author decorates their service code with .NET attributes (see section 7) which provide the container (at runtime) with information about the

service. The container's configuration file provides further information about services. When a GSW is instantiated in an AppDomain (see section 3.4), it loads the assembly containing the appropriate grid service. The config file tells the GSW which of the various serializer/deserializer modules registered with the container are usable by this service. The attributes on the service's class declare the port types supported by the service. The GSW instantiates instances of all the service's port types, both those written by the service author and those defined by the OGSINET specification and included with OGSINET.

The service's service data elements (SDEs) are initialized by the GSW and exposed through the interfaces defined in the OGSINET specification. Currently, both XPath and "ByName" queries are supported. Finally, a service's policy is initialized from any policy attributes of the service. Any requests made of this service are checked against this policy by the GSW.

3.3. Light-Weight Service Wrapper

The Grid Service Wrapper depends on the functionality of the container in order to handle requests and dynamically create or destroy service instances. While the container presents a powerful, server-side abstraction to service authors, there are occasions when "services", i.e. entities exposing service-like interfaces, are useful on the client-side (e.g. notification sinks). Also, certain server-side services have limited functionality (such as those that just hold some SDEs, e.g. ServiceGroupEntry). These services are often only used in concert with some number of other services and the protection of having these simple services in their own AppDomain is outweighed by the overhead of constantly communicating across the AppDomain boundary. OGSINET provides a second kind of service called the light-weight service, which allows entities with limited functionality to present OGSINET-compliant interfaces without the full container underneath them (nor factory services to create them). In OGSINET, such services are encapsulated by light-weight service wrappers (LWSW). Light-weight services are similar to grid services, but have the following restrictions:

- Cannot create other services
- Are terminated when the "parent" grid service, or client terminates (or when explicitly destroyed by the parent/client)
- Do not have the configurability of grid services (A grid service has both "service params", parameters specific to this service instance, and "global params", parameters for every instance running in the container. Light-weight services have no

service params, though they may use the container's global params when created on the server side)

3.4. Factories

Factories are services that create instances of other services. In OGSINET, this means creating a new AppDomain and a new Grid Service Wrapper in that domain. The Grid Service Wrapper will then load the assembly for the appropriate service and instantiate a new instance as discussed in section 3.2. A factory service stores a reference to the GSW in the new domain along with the published name of that instance (a GSR). This mapping is also sent to the dispatcher. Object references can be "wrapped" and passed across AppDomains, and this is how the reference to the new service instance (i.e., the reference to its GSW) is passed from the factory's domain to the dispatcher's.

In general, the type of instance created by a factory is specified at container-start time (i.e. in the container's config file). However, a second type of factory in OGSINET, the meta-factory functions differently. A meta-factory is a factory that receives a collection of .NET assemblies that define a grid service class in the creation parameters for its createService method. A meta-factory then creates a new factory that instantiates instances of the service defined in the assemblies (including auto-generating the WSDL for the new services). In this way, a meta-factory can deploy service instances into a container that are not specified at container-start time and which are not initially resident on the container's host.

3.5. Message Handlers

Message handlers perform message format specific processing on a service instance's incoming and outgoing messages (including exceptions the service may throw). OGSINET 2.0 contains two message handlers for the SOAP and remoting¹ message formats. A service instance's GSW will create message handler objects for each message format that the service supports when the instance is created. A message handler deserializes the request message that arrives (as a byte array) from the dispatcher. This includes creating any needed parameter objects and processing any message headers. When the request is completed or an exception is thrown, the handler will serialize the results into a byte stream which is passed to the dispatcher to be returned to the client. In this way, the

¹ OGSINET supports the "binary format over http" flavor of remoting. The remoting message handler deals with the binary format and the Microsoft remoting system handles the transport.

dispatcher handles the transport protocol while the message handlers deal with the messaging protocol.

The message handlers allow service authors to write services independent of messaging issues. Marshalling, unmarshalling and exceptions are all handled transparently by the message handlers.

4. Service Function Invocation

Each grid service instance provides some functions that clients can invoke. This section discusses the process by which client requests are served. We frame the discussion in terms of an HTTP transport, though other mechanisms could, in principle, be supported.

When a request arrives at the IIS web server, the following steps occur.

1. ISAPI filter: An ISAPI filter "rewrites"² the request's destination URI so it is handled by OGSINET's managed-code HttpHandler (if that message is bound for the OGSINET container).
2. HttpHandler routing: The HttpHandler dispatches out of the IIS/ASP.NET system and into the OGSINET container. The request message is sent to the container's dispatcher.
3. Dispatcher finds Grid Service: The dispatcher finds the appropriate GSW (and hence AppDomain) for the service by looking up the request URI in the dispatcher's service table. The dispatcher then gets a handle to that GSW and calls its ProcessRequest method, which takes the raw message as a parameter.
4. Process message headers: The GSW's message handler processes the headers of the raw message. If the message was a SOAP message, this is done by via the Web Services Enhancements (WSE) pipeline [10]. The processed header information is used by the container to check if the invocation message met the service's stated policy and is made available to the code of the invoked function (which may wish to process it further).

² The ISAPI filter dispatches requests to the ASP.NET infrastructure by "rewriting" the request URL. This is necessary because IIS dispatches requests based on extensions in the name of the requested page (e.g., request.asmx will be handed to ASP.NET to process). In order to handle arbitrary service names, an ISAPI filter was used because the request could be intercepted at an early stage in the processing (before IIS begins to dispatch) and rewritten such that any request starting with a container's well-defined prefix would be handled by OGSINET's HttpHandler.

5. Determine function name and parameters: The message handler deserializes the message body, determining the name of the function the client wants to invoke and decoding any invocation parameters.
6. Find method handle for method name: The GSW finds the port type implementing the specific function from its port type array and then uses reflection to get a handle to the desired method.
7. Invoke function: The GSW invokes the requested function and gets the result.
8. Serialize the results: The GSW uses the message handler to serialize the results (or a thrown exception) into a byte array.
9. Send results back to client: The GSW sends the result array to the dispatcher, which sends them to the `HttpHandler`, where they are sent back to IIS. IIS handles forwarding the results to the client.

5. Persistence

There are multiple types of persistence in the OGSi framework:

- Services that are automatically loaded into the container at startup (and therefore “persist” if the container goes down and gets restarted)
- Services that do not require soft-state keep-alive messages (i.e., services that have an infinite timeout)
- Services with state that is saved to permanent media (e.g., disk) and which can load that state back into a running service instance

Certain services must be available to make the container useful. For example, indexing/registry services are needed to discover existing service instances and handle resolution services are needed to determine how to communicate with them. OGSi.NET’s container configuration file allows the specification of services that are to be created and loaded when the container initializes.

The OGSi Specification [7] (and hence OGSi.NET) defines the notion of timeout for services. A service may elect to “die” if it does not receive a message from a client (which could be another service) within this time period. This prevents unused (and forgotten) instances from cluttering the container. However, this timeout can be set such that no keep-alive messages are needed. This, in effect, makes the service permanent as long as the container is running. However, if the container is shutdown or crashes, these services will not be restarted (as opposed to services listed in the container’s config file). Obviously, a service in the container’s config file will have both types of persistence - there seems to be little point in

starting a service automatically, but then letting it time out.

The removal of services when their timeout expires is the job of the dispatcher. Periodically, the dispatcher runs a “garbage collector” to remove old instances. The OGSi Specification does not say that a service must immediately be removed if its timeout has expired, but rather that clients may no longer count on that service’s availability. This means the dispatcher’s garbage collection machinery does not need to be highly synchronized with the clocks of the service instances.

Permanent service state can be supported through service data elements (SDEs). The Grid Service Wrapper provides access to a service’s SDEs through functions defined by the OGSi Specification. In OGSi.NET 2.0, service-specific SDEs are specified via attributes on the service’s (or port type’s) class or on data members. By default, this data does not persist between restarts of a service instance. However, an SDE itself can be annotated with attributes that tell the container to both save the SDE’s value when it changes and load the SDE’s value when it is unknown (presumably when the service is starting up). OGSi.NET refers to code run when an SDE’s value is retrieved as a `GetHandler` and code run when an SDE’s value is changed as a `SetHandler`. Grid service authors can write arbitrary `Get` and `SetHandlers` for each SDE to make them persistent (or perform any number of other functions).

6. Security

There are several security concerns in the creation of a .NET hosting container. Some of these concerns are unique to .NET, while some are issues for any hosting environment. First, OGSi.NET provides standards-based message layer security to the service instances (e.g. WS-Security) via the Web Service Extensions (WSE) pipeline [10] run by the message handler in the Grid Service Wrapper.

In addition, there are security concerns relating to the dispatcher and the container. How are these system components to be protected from badly implemented or malicious services? `AppDomains` provide the memory protection of a process without the heavyweight activity of creating a new process for each service. Allowing each service instance to live in its own `AppDomain` provides a large amount of protection for the other services in the container. Because the Grid Service Wrapper actually invokes functions on the service from within the service’s `AppDomain`, any problems that cause the service to crash or hang will not effect the dispatcher or the other

services in the container. However, if services are allowed to make calls into unmanaged code, they can bypass the protection of AppDomains.

Similarly, factories create Grid Service instances by creating a new AppDomain and then creating a Grid Service Wrapper in that domain. That GSW then loads the assembly for the actual grid service. By having the service's assembly loaded and the grid service initialized from within the new AppDomain, the remainder of the AppDomains in the container are protected from potential bugs in service author's service creation code.

We also wish to support running services that have the local host access privileges of particular users (i.e. the Windows equivalent of running a service as a particular UNIX user id). Note that this does not mean simply having a service that can spawn computational jobs under a certain user id, but having the service itself access local resources as a particular user. We are currently investigating two approaches to this problem. The first is to have a hierarchical container system in which a single master container dispatches to one of a set of other containers, each running under a different Windows user id. Each of these "user containers" could only create services that are running under a particular user id and hence every contained service would run as the container's owner. This approach is similar to the one used by the GT3 Managed Job Service for running computational jobs (except that this would also apply to the services themselves). A second option is to have any thread active in a given service run under the service author's WindowsIdentity. In this way, multiple threads with different ids may be operating within the same container.

Finally, it is not clear given this design, if we will need to provide some sandboxing capabilities above the inherent .NET security mechanisms (e.g., evidence-based security, policy evaluation in the CLR). While these .NET capabilities will be greatly leveraged, it is not clear if they, alone, are sufficient given our constraints. For example, the meta-factory's ability to instantiate services unknown to the container deployer might create challenges.

7. An Attribute-Based Programming Model for Grid Services

One of the key challenges for a "Grid programmer" today is how to expose functionality as an OGS-compliant Grid Service. One of the principle differences between programming a Grid Service and simply coding the service logic is that a certain amount

of meta-data is needed to actually expose that service logic as a grid service. This meta-data consists of instructions to the service's hosting environment about what elements (methods and data) should be exposed to clients and the conditions under which clients can access them (policy). This goes beyond simple automatic generation of service WSDL and client-side proxies to describing events that must be analyzed at runtime. In OGS.NET, programmers annotate their code with attributes that are subsequently used by both pre-processing tools (e.g., to generate static information such as WSDL documents) and by the service's hosting environment (e.g. to detect policy events). We argue that an attribute-based model can ultimately make creating a Grid Service as easy as creating a Web Service.

Although OGS.NET supports a number of different attributes, they are primarily divided into three classes, those that expose methods, those that expose data and those that express policy. The `[OGSIPortType]` tells the GSW that a service implements a particular port type. This makes it easy for a service author to support the various port types defined in the OGS specification by including an attribute. For example, the `[OGSIPortType(typeof(NotificationSourcePortType))]` attribute says that the service implements the functionality of the NotificationSource port type and causes the GSW to instantiate a NotificationSource port type to serve requests for instances of this service. The `[WebMethod]` attribute (which should be familiar to Web Service programmers) is also used by OGS.NET. Any method marked with this attribute will be exposed as a method on the port type in which it is defined. OGS.NET's WSDLGenerator will automatically place that method in any port type definitions and can automatically create WSDL bindings for that method. For authors requiring more precise control of bindings (such as those implementing the functionality for a given WSDL document), the `[SoapDocumentMethod]` attribute can be used to specify the SOAP binding exactly.

Service data can be exposed by using the `[SDE]` attribute on either service (or port type) class, or on a particular data member. The `[SDE]` attribute defines a Service Data Element that the service/port type should expose. When used at the class level, the attribute allows complete specification of all the SDE parameters (name, mutability, nilability, min and maxOccurs, etc.). When placed on a data member, this attribute essentially exposes that programming-language specific member as an SDE. For example, when placing this attribute on a integer member of a

port type, an SDE will be created with the name of that data member, type int, min and maxOccurs of 1, nilable = false and mutability set to mutable. In addition, whenever a call to findServiceData references this SDE, the GSW will return the current value of this data member. In this way, dynamic SDE can be created for which the programmer need only manipulate language specific variables.

Finally, service policy can be specified via class-level attributes (method level policy support is currently being added). OGSINET 2.0 supports definitions of security policy through the [Integrity], [Confidentiality], [Message] and [Token] attributes. [Integrity] and [Confidentiality] allow specification of digital signing and encryption requirements respectively. Each attribute specifies the portion of the message to which the requirement applies and the token(s) that can be used to either sign or encrypt the message portion. [Message] can be used to define other requirements of the message. Currently, only the message age can be specified and “compliant” messages must be less than the given age (based on their WS-Timestamps). Lastly, [Token] is used to specify the cryptographic tokens that must be (or cannot be) used to sign or encrypt messages. Various token types (e.g. X509, Kerberos, Username) can be specified although currently only X509 tokens are implemented. Each token can be parameterized with a token-specific string. For X509 tokens, this string allows DN and CA to be specified. When the policy of checked against a message using an X509 token, if DN or CA was specified, the token must contain the correct DN, CA or both. We note that certain security policy decisions are appropriate for the grid service author (e.g. which functions require administrative access) while certain decisions are appropriate for the grid service deployer (e.g. which CAs are acceptable). While it is appropriate for grid service author decisions to be represented “in code”, it this is probably not appropriate for grid service deployer decisions. Currently, we are working on defining what policy authors and deployers wish to control and how each should specify this in OGSINET.

More details on OGSINET’s programming model see “Attribute-base Programming for Grid Services” [8] and “OGSINET Programmer’s Reference” [9].

8. Discussion

In order to achieve wide-scale inter-operability among OGSINET-compliant implementations, it is important to leverage work currently being done, or planned to be started, on other hosting environments. To date, GT3 is

the only other complete implementation of the OGSINET specification and therefore provides the most leverageable work. Currently, OGSINET leverages this work to perform baseline compatibility tests. Both GT3 and OGSINET come with graphical browsers which can be used to navigate through and interact with the services that live in a container. By pointing the GT3 browser at an OGSINET container and hitting the “load” button, we can test a number of functions. The opposing test with the OGSINET browser and the GT3 container is also performed. This test is non-trivial because it involves activating a service (the container registry), subscribing to and receiving notifications, performing handle resolution and invoking service methods. We then test the implementations further by exercising the functions of the GridServicePortType (finding service data, termination, etc) and the FactoryPortType (by creating new services). These tests ensure that both containers can inter-operate whether a particular they are operating as the server or the client.

A deeper issue, common to all OGSINET containers, is how they will inter-operate “below the spec”. The OGSINET specification does not fully define (nor should it necessarily) the exact content of each message exchanged between grid services. OGSINET provides a framework for request/response messages to a standard set of port types, but the exact semantics of those messages is not always fully defined. For example, OGSINET cannot define what a GSR will look like for all possible transport/messaging formats. Further specification of the exact results returned by common system components, e.g. Container registry services, will allow client and services from different implementations to inter-operate. We are currently engaged in an effort with Jarek Gawor and the GT3 team to make OGSINET and GT3 inter-operate such that unmodified clients (written for either container) can access services running in both hosting environments. We hope that this work will further define the OGSINET-compliance process and guide other container implementers whose experiences can also be filtered back into the process.

Another useful product of the creation of the OGSINET container was experience with the OGSINET specification itself. We are in a unique position to reflect on different aspects of OGSINET and how useful they are to the process of creating grid applications. The first way in which OGSINET was useful was that building on top of web services allowed us to leverage the deeply integrated support of web services that Microsoft has placed in .NET. It also allows access to web service tools (e.g. Microsoft’s wsdl.exe can be

used to generate client stubs for OGSINET grid services).

The utility of the factory paradigm has been debated [5], but in developing OGSINET, it was proved useful in some circumstances. Firstly, factories and instance provide natural abstraction for exposing computational jobs as services. Users are used to the idea that jobs “spawn” as individually addressable/monitorable entities. Second, the meta-factory concept provides a mechanism by which services not known to the container when it starts can be deployed. In a factory-less world, it is unclear how new types of services (not just new instances of services) can be instantiated dynamically.

Other OGSINisms such as asynchronous notification and handle resolution are useful precisely because OGSINET defines functions for achieving these common grid computing needs. Such facilities will be needed by a grid system and will likely be defined in different, incompatible ways, if left to the individual systems. Here OGSINET fulfills its main function by defined mechanisms for common grid tasks.

9. Conclusion

In this paper, we have presented OGSINET, an OGSINET-complaint hosting environment based on the Microsoft .NET platform. OGSINET forms a bridge between the native Windows world and the OGSINET-based systems in the Unix world, such as GT3.

Many of the future plans for OGSINET have been presented throughout this paper. The immediate plans include concentrating on inter-operation with GT3 and improving/evaluating performance of the OGSINET container itself. We will expand the security and policy capabilities of the container and enrich the programming model by the introduction of new attributes. A greater understanding of the ways in which OGSINET is used will be gained as it begins to be used by the community. This understanding will shape the future of OGSINET and hopefully provide feedback useful for all OGSINET-complaint systems.

10. References

[1] Foster, I., Kesselman, C., Nick, J., and Tuecke, S. 2002. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[2] Globus Toolkit 3.0.2. 2003. The Globus Alliance. Available at: <http://www.globus.org/toolkit/gt3-factsheet.html>.

[3] IBM Corporation and Microsoft Corporation. 2002. Security in a Web Services World: A Proposed Architecture and Roadmap. <http://www-106.ibm.com/developerworks/webservices/library/ws-secmap/>.

[4] Microsoft Corporation. 2003. .NET Framework. <http://www.microsoft.com/net/>.

[5] Parastatidis, S., Webber, J., Watson, P. and Richbeck, T. 2003. A Grid Application Framework Based on Web Services Specifications and Practices. Available from <http://www.neresc.ac.uk/ws-gaf/>.

[6] Sandholm, T., Tuecke, S., Gawor, J., Seed, R., Maguire, T., Rofrano, J., Sylvester, S. and Williams, M. 2002. Java OGSINET Hosting Environment Design – A Portable Grid Service Container Framework. Globus Toolkit 3 Alpha 2 docs.

[7] Tuecke, S., Czajkowski, C., Foster, I., Frey, J., Graham, S., Kesselman, C., Vanderbilt, P., and Snelling D. 2002. Grid Service Specification – Draft 11/4/02. OGSINET Working Group, Global Grid Forum. <http://www.ggf.org/ogsi-wg>.

[8] Wasson, G. and Humphrey, H. 2003. Attribute-based Programming for Grid Services. *GGF9 Workshop on Designing and Building Grid Services*.

[9] Wasson, G. 2003. OGSINET Programmer’s Reference. in the docs directory of the OGSINET 2.0 distribution.

[10] Web Services Enhancements for Microsoft .NET. <http://msdn.microsoft.com/webservices/building/wse/default.aspx>