OGSI.NET: An OGSI-compliant Hosting Container for the .NET Framework

Glenn Wasson, Norm Beekwilder, Marty Humphrey
wasson@virginia.edu | nfb5z@virginia.edu | humphrey@cs.virginia.edu
Grid Computing Group
Computer Science Department
University of Virginia
Charlottesville, VA 22904
Draft of May 17, 2003

1. Introduction
The Open Grid Services Architecture (OGSA) represents a new vision of both the grid and web services. By defining standard communication protocols and formats, OGSA represents the means to build truly large-scale, interoperable grid systems. The Open Grid Services Infrastructure (OGSI) is a set of WSDL specifications that are consistent with the OGSA specification and which are to be implemented on a variety of hosting platforms. To date, work has been focused on hosting environments for various Java platforms (J2SE, J2EE) primarily operating on UNIX systems. The work described in this paper develops a hosting environment for the Microsoft .NET framework.

In addition, we believe that the OGSA/OGSI specification processes will benefit from the creation of hosting environments on additional platforms. We seek to not only leverage the work being done on the Java OGSI environment in defining both types and interfaces, but to influence both the OGSI and OGSA specifications based on our experience with the .NET platform. The capabilities enabled by the .NET framework may be both important for grid systems and unconsidered by current work.

2. Basic Architecture
The two highest-level design goals in developing OGSI.NET were: (a) to support the dynamic creation of stateful grid service instances that persist between client invocations and (b) to use IIS to receive requests from clients. The first goal requires that services maintain state (which they must do to meet the definition of grid services [1]) and that they persist “in memory” between client invocations. This persistence requirement was because it was felt that it would be too inefficient to create a new “server object” for each request (possibly reloading large amounts of state from previous requests). The goal of using IIS came from the desire to take advantage of the close integration of Microsoft software and to highlight potential inter-op issues that may arise from not using Apache.

The basic design of OGSI.NET is to have a container entity that “holds” all the service instances running on a host. The container process consists of a collection of 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 OGSI.NET (Tech Preview 1.1) 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 OGSI.NET architecture by sending a message to the IIS web server. In order to support arbitrary names for grid services, OGSI.NET uses an ISAPI filter to intercept requests at an early stage in the IIS request chain. This filter re-writes the request so that IIS will dispatch it to OGSI.NET’s ASP.NET HttpHandler. This HttpHandler dispatches the request to the OGSI.NET 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 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.

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 OGSI 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 searches its port types to find which implements the requested method. The GSW then uses reflection on the selected port type to get a handle to the proper method and invokes it, passing in any parameters and message specific data (e.g. SOAP header data). The data returned by the invocation is serialized by the GSW and sent back to the dispatcher as a binary array. The dispatcher sends that array back to IIS for return to the client.

3. System Elements

This section discusses the major components of OGSI.NET: the dispatcher, the service wrappers, 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 URIs 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 return a byte stream to the dispatcher (if appropriate), which the dispatcher will 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 an enhanced because the dispatcher does not “process” the incoming or outgoing messages. If the dispatcher did marshall/unmarshall 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 can be effected.

3.2. Grid Service Wrapper

The Grid Service Wrapper or GSW encapsulates the various functional units of a grid service. Each AppDomain in the container has a GSW and each GSW wraps a grid service instance. The GSW provides a convenient programming model for the grid service author by allowing:

- Pluggable, service-specific message serializers/deserializers
- Easy specification of port types supported by this service
- Pluggable support for port types supported by a service, but not written by the service author (e.g. the grid service port type)
One of the difficulties encountered by both this project and the ANL OGSI project in Java was how to allow a grid service author to “inherit” many different port types that they might wish their service to support in a language with single inheritance. Both projects are now moving in a direction in which the grid service author decorates their service with information that allows tooling or the runtime system to add appropriate port type support without “in code” inheritance. This project uses the grid service wrapper for this purpose.

A grid service author decorates their service code with .NET attributes which will be recognized by the container runtime when the assembly is loaded. When a GSW is instantiated in an AppDomain (see Factories), it loads the assembly containing the appropriate grid service. The service’s class tells the GSW which of the various serializer/deserializer modules registered with the container are usable by this service. It also lists all the port types supported by the service. The GSW uses this information to instantiate instances of all the service’s port types, both those specific to the service and those defined by the OGSI specification and included with OGSI.NET.
The service’s service data elements SDEs are initialized by the GSW and exposed through the interfaces defined in the OGSI specification. Currently, both XPath and “ByName” queries are supported.

3.3. Light-Weight Service Wrapper

The Grid Service Wrapper depends on the functionality of the container process in order to handle requests and dynamically create or destroy service instances. While the container presents a powerful abstraction to service authors on the server-side, there are occasions when services are useful on the client-side (e.g. notification sinks). These services have the OGSI-compliant interfaces, but have limited functionality and so do not require the full container underneath them (nor factory services to create them). In OGSI.NET, such services are called light-weight services and they 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 (a GSW contains references to the light-weight services it has created, but the LWSW has no equivalent)
- have no SDEs
- are terminated when the “parent” grid service, or client terminates (or when explicitly destroyed by the parent/client)
- service parameters (i.e. they are not configurable with individual params at startup, but they can use the container’s global params if they are being created on the server side)

OGSI.NET uses light-weight services for notification sinks and client-side handler resolver services. Note that light-weight services are not restricted to client-side use (and are used internally to receive notification messages within the OGSI.NET container).

3.4. Factories

Factories are services that create instances of other services. In OGSI.NET, 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 URI). 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.

3.5. Message Handlers

Message handlers perform message format specific processing on a service instances incoming and outgoing messages (including exceptions the service may throw). OGSI.NET TP1.1 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. The message handler will deserialize the request message that arrives from the dispatcher, determining the function to invoke and creating any needed parameter objects. When the request is completed, 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 dispatcher handles the transport protocol while the message handlers deal with the messaging protocol.

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

4. Service Function Invocation

¹ As of TP1.1, OGSI.NET 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.
Each grid service instance provides some functions that clients can invoke. In serving an invocation, IIS routes a request to the dispatcher and the dispatcher routes it to the correct GSW. The GSW performs the actual invocation on the service instance and returns the results back along the reverse path.

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

1. ISPAI filter: An ISAPI filter routes the message to the managed-code HttpHandler, if that message is bound for the OGSI.NET container.
2. HttpHandler routing: The HttpHandler dispatches out of the IIS/ASP.NET system and into the OGSI.NET container. The 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 the `ProcessRequest` method, which takes the raw message as a parameter.
4. Process SOAP headers: If the request message is SOAP encoded, the GSW processes the SOAP headers of the raw message by running it through the Web Services Enhancements (WSE) pipeline [3].
5. Determine function name and parameters: The GSW finds the name of the function to invoke on the grid service and decodes any parameters from the body of the request message using the message handler.
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 into a byte array.
9. Send results back to client: The GSW sends the result array to this 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 Grid Service Specification [3] (whose current draft is now called the OGSI Specification) 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.

---

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 name, an ISAPI filter was used because the request could be intercepted at an early stage in the processing (before IIS begins to dispatch) and a request with any name can therefore be handled by modifying the name of an request bound for the container.
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 Grid Service 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. Although the behavior of this garbage collection mechanism is not precisely mandated by the Grid Service Specification, there may be architectural advantages if multiple hosting environments behave similarly (certainly clients may benefit from roughly consistent semantics); as such, we are currently investigating the service destruction semantics in the Globus Toolkit 3 hosting environment produced by Thomas Sandholm’s team and how they should influence the workings of OGSI.NET.

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 Grid Service Specification. In TP1.1, service-specific SDEs are specified in the service’s constructor and a set of standard SDEs possessed by all services are created and initialized by the GSW. However, this data does not persist between restarts of the container. Future versions of OGSI.NET will support persistent SDEs by storing them to permanent media using one of two mechanisms. SDEs will be annotated as persistent and this will cause the container to save the SDEs value whenever it changes. While this mechanism is convenient for service authors to use in saving small data elements, the automatic save-on-update paradigm does not work well for large data structures that may be modified many times in a short period. Therefore, we will provide the ability to annotate services such that their state is saved when they are unloaded from the container or the container is shutdown. This annotation will allow the service author to provide custom Activate() and Deactivate() methods to perform this state saving and reloading.

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 indicative of the broader class of “hosting environments”. First, we provide standards-based message layer security to the service instances (e.g. WS-Security). This is done via the Web Service Extensions (WSE) pipeline run by the message handlers 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 domain. 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 will have the name of the appropriate assembly to load 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. Currently we are pursuing the first option while we investigate the operational feasibility of the second option.

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, support for custom state saving for Grid Services might create challenges.

7. Remoting as a GSR

Grid services can be identified/invoked by both their Grid Service Handle (GSH) and their Grid Service Reference (GSR). The GSH is a unique, system-wide name that can be used to find more detailed information necessary to communicate with the service. This information is represented by the GSR, which can hold protocol and other information. A HandleResolver service is used to map a GSH into a GSR. A client with a GSH would typically call a HandleResolver to get a service’s GSR and then use the GSR to communicate directly with the service. We believe that remoting offers a new type communication mechanism (and hence a new type of GSR) that has not been previously considered.

OGSI.NET Tech Preview 1.1 provides experimental support for the remoting GSR. A GSR for a service that supports remoting has the same format as a GSRs used for SOAP communication, but each <port> element may additionally contain a <remoting:address> element. Note that using remoting to communicate with a service does not effect that service’s WSDL, it just adds an additional address type to the port description. Also, using the remoting GSR does not effect the service’s ability to have a SOAP GSR as well.

There are multiple “flavors” of remoting and OGSI.NET currently supports only Microsoft binary format sent over http. While other flavors may also be supported in future releases, binary-over-http was initially selected because the binary encoding is more space efficient than SOAP and the firewall friendliness of http is desirable. A remoting address’s location field is a URI of the format:

protocol://host:port/container_name/path

where protocol is currently http, host and port are the machine and port on which the remote container is listening, container_name is the Microsoft remoting system’s name for the remote object being communicated with (in this case the container) and path is the path to the service within the container. An example element is:

<remoting:address
location="http://localhost:8989/OGSIContainer.rem/samples/counter/basic/
CounterFactoryService" xmlns:remoting="http://www.cs.virginia.edu/GCG" />
It should be noted that the Grid Service specification allows a service to have multiple GSRs for a single GSH and each client need not be given the same GSR by a resolver. The remoting GSR may only be appropriate for clients running on Windows machines and so either the resolver must be able to determine this when returning a GSR, or the resolver must return all available GSRs and let the client decide which one to use.

One interesting issue with remoting connections is what their endpoints should be. Since the Microsoft remoting system allows each service instance to be a “remotable” object, it would be possible to have clients contact the instances directly and avoid the container’s dispatcher. While this may increase efficiency, it is not the approach taken by OGSI.NET. All remoting connections are to the dispatcher (i.e. the container itself) and activate the same dispatch mechanism as connections coming through IIS. This allows the dispatcher to perform container-level routing on all requests (e.g. to different containers in a hierarchy), not just those from IIS. This approach is also important because it prevents clients from needing access to the assemblies containing the remote service types (or appropriate interfaces to those types). We discuss the client-side use of remoting in section 7.

Remoting security is an open concern. While it is possible to insert encryption/decryption facilities in the remoting stack, and remoting messages can carry out-of-band data in the form of headers, TP 1.1 does not fully utilize these capabilities. For example, TP 1.1 remoting messages use headers to communicate out-of-band service-specific data, Microsoft does not currently provide support for the WS-* specifications (e.g. WS-Security) over remoting connections.3

7.1. Authoring Services using Remoting
Through the use of OGSI.NET’s message handlers, service authors need not be concerned with the transport or messaging protocols used to access their service. This does not imply that service authors cannot take advantage of certain properties of those mechanisms (if the message handler’s expose them), merely that the authors need not be aware of these details. In order to make a service accessible via a remoting GSR, all that must be done is instruct the GSW to load the remoting message handler into a service instance’s AppDomain by adding the appropriate assembly name to the service’s section of the container’s config file.

7.2. Accessing Services through Remoting
The experience of using a remoting connection from the client’s perspective is also important. Since the WSDL of a service is not messaging specific, OGSI.NET leverages Microsoft’s wsdl.exe proxy class generation tool. A remoting proxy for a service can be created by using wsdl.exe to generate a proxy based on the service’s WSDL and then modifying the base class to derive the proxy off OGSI.NET’s RemotingProxy base type. This is similar to the way in which a wsdl.exe generated proxy is modified to allow it to use the functionality of the WSE. Using this proxy in client code is the same as using any other wsdl.exe generated proxy.

8. Attribute Programming Model for Services
.NET uses attributes extensively to annotate classes with useful meta-data. OGSI.NET uses attributes to bring similar functionality to the service programming model. The OGSIPortTypeAttribute 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 OGSI specification by including an attribute. For example, the [NotificationSourcePortType] attribute says that the service implements the functionality of the NotificationSource port type. The OGSIOperationOverrideAttribute allows a service author to

---
3 Although remoting messages can actually be encoded using SOAP and so the WSE could be used to generate the appropriate SOAP headers before sending the message over the remoting connection. However, there seems little benefit to providing to using soap-over-http remoting and so we have opted for the binary format.
annotate a method as overriding the default implementation of one of the port types described by the 
OGSIPortTypeAttributes. When the GSW determines the name of a method to invoke, it looks for 
any operation overrides on that method. If one exists, it will be called. Otherwise, the GSW will search its 
list of implemented port types to find the method name and therefore call the default implementation. The 
OGSISoapHeaderAttribute and OGSIRemotingHeaderAttribute allow a service author to 
annotate a method as expecting to receive supplementation information when that method is invoked in 
the headers of the invoking message. The OGSISoapHeaderAttribute is needed because the .NET’s 
SoapHeaderAttribute does not specify the type of object to be placed in the SOAP headers. This 
means that only simple types can be serialized. In order to allow the message handlers to serialize and 
deserialize arbitrary types, the OGSISoapHeaderAttribute provides meta-data describing the type of a 
header value with a particular QName. The OGSIRemotingHeaderAttribute does not need to 
provide type information because remoting’s binary format includes type information automatically. 
However, there is no analog of the SoapHeaderAttribute for remoting headers and so the 
OGSIRemotingHeaderAttribute provides this. Finally, the SchemaRootAttribute tells a light-
weight service where to find its schema. For full-fledged grid services, this is handled by the container. 
However, since light-weight services do not have a container underneath them, they need this attribute 
information.

9. Leveraging Current OGSI/OGSA work
It is important to leverage the large body of work being done in support of the Globus Toolkit 3 (GT3). 
Specifically, two goals of OGSI.NET are 1) to inter-operate with services running in Java-based hosting 
environments and 2) to allow client to use the same software (unmodified) to access services running in both 
GT3 and the OGSI.NET hosting container. This is not as simple as following the Grid Services Specification 
because that specification is not sufficient to fully define the operation of a hosting environment. For 
example, the Grid Service Specification does not specify the name of the indexing service running in a 
container or the name of a local handle resolver for a container. However, both clients and service authors 
need to have well-defined values for these names in order to make use of them. If clients (or other services) 
wish to call services running in any type of container, many such details must be worked out.

The primary way in which the GT3 work is leveraged today is via GT3’s WSDL files and data type 
definitions (.xsd files) for standard types defined in the Grid Service Specification, such as service data 
elements. Through the wsdl.exe and xsd.exe tools from .NET, these definitions are operationalized into C# 
code and used by the OGSI.NET container. Wsdl.exe was designed to create proxies for web service and not 
OGSI-compliant grid services, so these proxies may have to be modified depending on the needs of the 
clients. However, the WSDL files from GT3 can be used unmodified.

10. References
Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 
22, 2002.

OGSI Hosting Environment Design – A Portable Grid Service Container Framework. Globus Toolkit 3 Alpha 2 
docs.

wg.