

# Grid Services Development Framework Design

*Draft Version 0.13 – Note, this design is still a work in progress*

Thomas Sandholm, Jarek Gawor  
{sandholm, gawor}@mcs.anl.gov

## Abstract

This document describes the internal design of an implementation of the Open Grid Services Architecture (OGSA), focusing on the core infrastructure elements a.k.a. the Open Grid Services Infrastructure (OGSI). We present the key design elements on the server side, and on the client side as well as the development environment used to prototype the implementation. The development framework implementation (OGSADF) can be viewed as the glue between the transport- and marshalling engine, and the hosting environment of application code. Dynamic creation of stateful service instances, lightweight discovery and inspection, and asynchronous notifications are all integral features of the framework. We describe how these features were implemented, and challenges we faced when implementing them. This document also lays out the programming model for service providers who want to plug their services into the framework. An example client-side programming model currently supported by the framework is also described. We then take a closer look at the Web services, and programming language toolkits used to implement various pieces of the framework. Finally, we describe how these tools were used to implement the Grid Security Infrastructure (GSI) in OGSADF.

## Table of Contents

1	Introduction.....	3
2	WSDL Design.....	3
2.1	Structure and Namespace Usage.....	3
2.2	Types, Messages, and Faults.....	4
2.3	Extensibility Elements .....	4
2.4	Templates.....	4
2.5	Definition Conventions.....	4
3	Server Side Components.....	5
3.1	Dispatcher .....	5
3.2	Persistent Services .....	5
3.3	Factories.....	5
3.4	Lifetime Management.....	6
3.5	Notification Sources.....	6
3.6	Service Properties .....	6
3.7	WSDL and WSIL Generation.....	6
3.8	Service Data Containers.....	7
3.9	Registries.....	7
3.10	Class Name Conventions .....	7
3.11	Class Overview .....	7
3.12	Writing an OGSA Service .....	7
4	Client Side Components .....	8
4.1	Stub Generation .....	9
4.2	Discovery and Inspection.....	9
4.3	Notification Sinks .....	9
4.4	GUI and Demo Framework .....	9
4.5	Writing an OGSA Client.....	10
5	Development Environment .....	10
5.1	Hosting Environments .....	10
5.2	Configuration .....	10
5.3	Build and Test Environment.....	10
5.4	Debugging and Logging .....	11
5.5	Additional Tools .....	11
6	Security .....	11
6.1	Service Security .....	11
6.2	Client Side Security .....	11
7	Future Work.....	12
8	Acknowledgements.....	12
9	References.....	12

# 1 Introduction

The Open Grid Services Architecture (OGSA) [1][2] describes and defines a Web services based architecture comprising a set of interfaces and their associated behaviors to facilitate distributed resource sharing in heterogeneous multi-institutional dynamic environments. The main goal of the implementation was to provide a framework that makes it easy to integrate, develop and use services in a grid environment. The framework implementation (OGSADF) can be viewed as the glue between the transport- and marshalling engine, and the hosting environment of application code. By providing standards-based, customizable implementations of the interfaces defined in the GS specification [2], most of the grid behavior will be transparent both to the hosting environment, as well as the application code. A further objective of the reference implementation was to exemplify key OGSA concepts, as an aid to specification implementers, hosting environment providers, and service providers. The APIs, runtime components, design conventions, and the development environment used by the framework are described in detail.

OGSADF builds upon Web services technologies like SOAP[3], WSDL[4], and WSIL[5] to support management of distributed state; lightweight inspection, and discovery; and asynchronous notifications. We will describe important design decisions made in all of these areas from a developer's point of view. All external components are exposed through WSDL interface descriptions directly derived from the GS specification. Hence, we start by looking at the WSDL design. Then we describe the server side component architecture, which has been the emphasis of our initial work, and how to write services that plug into the framework. Thereafter we discuss the client side support offered to discover, inspect and use Grid services. Note that both client side, and server side components can reside in a client (e.g. in the form of notification processing) or in a server (e.g. in the form of a gateway). The state-of-the-art of tools supporting Web services has had a big impact on how OGSADF was designed, and the currently used toolset is hence described. As an example the currently provided security support depends highly on the development and hosting environments chosen to implement OGSA. An overview of how we implemented the Grid Security Infrastructure (GSI) is therefore given in conjunction with the development environment discussion.

## 2 WSDL Design

WSDL provides an abstract framework for describing Web services in enough detail to send messages to service providers. Since it does not mandate any programming model, transport nor encoding mechanism, the high-level interface definitions can potentially be reused in many environments. However, the absence of inheritance imposes some challenges to making the descriptions reusable.

### 2.1 Structure and Namespace Usage

A WSDL definition is divided into 5 basic parts: types, messages, portTypes, bindings, and service descriptions. In OGSADF, these parts reside in separate XML namespaces [6] to promote reusability, and flexibility. However, message, and portType definitions were found to be very tightly coupled, so they reside within the same namespace in the current model. The reason for splitting up the various definitions across namespaces was to allow them to be easily included in any higher level WSDL definition by using the import construct. Further, the type definitions are in common for all the core services. The service part only serves as a template, and is fully determined first at run time, so it is also reusable among service instances (see 2.4, and 3.5). The core service interfaces (as defined in [2]): grid service, factory, registry, notification, handle map; don't expose any service definitions, but are rather used as building blocks for composing service implementations.

## **2.2 Types, Messages, and Faults**

A limited set of custom (a.k.a. complex) types are defined for efficient tool supported marshalling of common message parts. These include notification subscription, service registration and keep alive/soft state management types. Most of the types however allow for extensibility elements to be plugged in (more in this in the next section) using XML Schema[7] any constructs. A key design point is that all input parameters and output parameters of the core service interfaces are defined in a single, XML Schema-defined complex type. This allows for maximum flexibility, and definition reuse since the WSDL message elements cannot be extended while the complex types defined in an XML Schema can. Faults are simply specific reusable messages in WSDL, and are thus also defined in a separate namespace.

## **2.3 Extensibility Elements**

Extensibility elements are heavily used in document style specifications like WSDL and WSIL to allow vendors to come up with their own standards within a certain community. In a similar way, we want the grid service interfaces to allow for community standardization and customization. Due to the flexibility required to achieve this, a document/literal model is commonly used in SOAP. The XML any construct is used to denote that arbitrary XML can be passed in a particular position in the message type. If the deserialization infrastructure understands the message type it can demarshal it, and hand it to the application transparently without the application even being aware of extensibility elements being used. If not, it can hand it up one level in the marshalling chain (potentially all the way to the application) until someone recognizes the type and demarshals it. If the type has been passed all the way up to the application level without being recognized and mapped to a strongly typed programming language construct, then it is exposed as an XML Infoset type, such as a W3C DOM Element. Examples of extensibility elements applying this technique in the framework are: factory input parameters, query expressions and results, and service data elements.

## **2.4 Templates**

As mentioned in section 2.1, the WSDL definition for a service is not fully determined until runtime to allow for dynamically created (stateful) services, therefore the WSDL definition only serves as a template. In particular the endpoint in the service element is not fully qualified by the service description author, but by the server side run time (see 3.5). The service element must, however, be present to define protocols (bindings) that may be used; and to let the framework know what portTypes a particular service supports.

## **2.5 Definition Conventions**

In order to simplify the readability of WSDL files, a number of naming conventions are used. Although most of the definitions are not for human consumption, our approach is to start out from a WSDL document when defining a grid service, similar to how one would start out by defining an IDL document in CORBA, so the definition must be kept as concise and clear as possible. The conventions are summarized in Table 1. File names containing these definitions follow the same conventions as defined in the Table 1, with the difference that lower case and underscores are used to separate words.

WSDL Element	Suffix Convention	Example
User defined XML (complex or simple) Type	Type	HandleType
User defined Element (used to wrap XML types in doc/lit mode)	Element	ServiceDataElement
Fault	Fault	InvalidHandleFault
PortType	PortType	RegistryPortType
Binding	<binding>Binding	FactorySOAPBinding
Service	Service	VORegistryService
Definition	Definition	FactoryDefinition

**Table 1: WSDL conventions**

### 3 Server Side Components

The reference implementation is based on object oriented programming concepts like inheritance and polymorphism. The design could however still serve as a reference point for other hosting environments, since the basic component functionality distribution would be the same. The server side part of the framework is built on top of a marshalling engine, e.g. handling SOAP marshalling. It comprises dispatchers, PortType Skeletons, service provider APIs, and run time services. The skeletons are provided in two forms, delegation skeletons to tie in grid service behavior into a user service using delegation, and base skeletons to be extended by user services. Note that base skeletons are, to a certain degree, container specific whereas delegation skeletons can potentially be reused in many container implementations. The base skeletons typically implement a set of common PortTypes by delegating to the respective delegation skeletons.

#### 3.1 Dispatcher

A custom dispatcher is used to intercept calls from the marshalling engine and dispatch them to the correct service instance encoded in the URL used to contact the service (e.g. the endpoint in the soap:location element in the WSDL service). The dispatcher contacts a hierarchical repository of service nodes to look up the instance to pass back to the marshalling engine, which eventually makes the invocation. The dispatcher is also used when a HTTP GET inspection is invoked on the Grid service handle to get the WSDL description of a service instance.

#### 3.2 Persistent Services

A number of services should be easily accessible at any time using well-known names defined in the configuration to facilitate bootstrapping. Examples of such services may include factories and registries. These services are called persistent since they are guaranteed to be available when the container starts up. Since containers like Java Servlet Engines[8] do not load the service code into the container until it is being invoked, we took the approach of doing a 'lazy load' of these services the first time a request comes in, to add them to the internal repository used by our dispatcher. From then on, persistent services are treated the same way as transient services by the framework.

#### 3.3 Factories

Persistent services follow the common model of standard stateless web services fairly closely, and are easy to implement using standard Web services tools. The GS specification [2], however, mandates that it must be possible to create stateful services at run time using factories. Although not mandated by OGSADF, a factory is typically a persistently configured service that is

responsible for creating services of a particular type. In the simplest case, the only thing the service provider has to do is to tell the framework what PortTypes the instances support that the factory can create (commonly a factory is responsible for creating one type of instance) and respond to a create callback to actually instantiate an instance of the new service. The framework then generates a WSDL document for the instance, and adds it to the internal repository. Both the factory, and the instance may be configured to publish themselves in external registries as well (see 3.11). A factory also exposes itself as a read-only registry to allow discovery of all services created by a particular factory.

### **3.4 Lifetime Management**

The lifetime of a grid service instance starts when either the container starts up or when a factory creates it. The lifetime ends either when the client of the service explicitly invokes destroy on the service, or when the termination time of the service has elapsed. In the current design, the factories are sweeping its instances periodically and destroying those with expired lifetimes. The application code may subscribe to a notification when an instance is destroyed, otherwise both changes in termination time through keep alive calls and destruction calls are handled by OGSADF, and are hence transparent to the service provider.

### **3.5 Notification Sources**

The notification source service is provided as a delegation skeleton and can hence be easily added to any service implementation. The notification source exposes an API to add topics exposed by the service, and to send notifications to the source on a topic. The delegation skeleton handles remote subscriptions, and sends asynchronous notifications to the handles that have registered interest in a topic. Some services like factories and registries have built in notification sources, and expose topics to allow clients to get notified when new services come on line.

### **3.6 Service Properties**

The delegation and base skeleton design makes it flexible to compose services supporting the desired PortTypes. But how do all these decoupled delegation skeletons share state with the service instance? –This is done through thread safe management of a properties table used to look up and register properties of any type keyed with a well-defined name. These properties are mainly intended for internal use in the framework code, but service providers can also get access to them by extending the base ServiceSkeleton class, or by using the ServiceProperties interface. Example of properties stored here are: the generated WSDL document, and the path to the instance in the internal repository. Further, configuration variables of persistent services are stored in these properties, and some of them are propagated down to instances created by that factory. Instances can, for instance, always get a reference back to the factory that created them.

### **3.7 WSDL and WSIL Generation**

As mentioned above, WSDL is generated using template schema definitions, describing what PortTypes and Bindings a service supports. All instance specific information like endpoint location is however generated when a service is created (either by its factory or by the container). OGSADF also supports WSIL generation based on the internal repository of services (or branches of it), or just a list of services. All grid services make use of the WSDL generation, and all factories and registries make use of the WSIL generation to expose a list of their contained services.

### 3.8 Service Data Containers

All grid services must support discovery of service data elements, some of which are well defined by the GS specification [2]. In order to facilitate this, a Service Data Container is provided for all services in OGSADF. It exposes an API to easily add, find, and remove service data elements. The Service Data Container maintains a map of XML elements published by its service. This map is searched when remote queries are executed on the service using the findServiceData operation, which must be supported by all grid services. A query language framework is provided by OGSADF, which makes it easy to add support for new query languages. Currently searches by service data name, and service data type are supported. As XML Query and XPath like query expressions get more widely supported in the available web services tools, it will hence be straightforward to add in support for these in the service data container implementation.

### 3.9 Registries

So far we have only mentioned APIs, Skeletons and dispatchers, but OGSADF also provides support for some base services built on top of the core service interfaces described above. Two registries are supported. One allows a container to be inspected for all its services; and another, called the virtual organization registry, allows services to register themselves with the registry remotely, and publish service data elements that can be searched. Both registries support notification of registry updates, and can be subscribed to by any clients. The registries can be used as bootstrapping services when browsing through a collection of services in a virtual organization.

### 3.10 Class Name Conventions

Similarly to the conventions mentioned for WSDL in 2.5, there are a number of conventions for classes in OGSADF summarized in Table 2.

Type of Class	Suffix Convention	Example
Skeletons that should be used for delegation	DelegationSkeleton	FactoryDelegationSkeleton
Skeletons that should be extended	Skeleton	FactoryServiceSkeleton
Internal interfaces	-	RegistryProvider
Marshalling engine interceptors	Handle	PersistentServiceHandle
Service Dispatchers	Provider	RPCURIProvider

**Table 2: Class Name Conventions**

### 3.11 Class Overview

Figure 1 shows a UML class diagram summarizing the class relationships on the server side in OGSADF. In the figure, we note that all services in the framework have built in support for querying their internal service data using the GridService interface. We also see that factories make use of service sweepers to destroy instances with expired timeouts, and they have a notification source implementation to send out notifications, for instance, when services are created and destroyed. All services in a container are put into a hierarchical structure of service nodes to facilitate internal service lookups and WSIL generation.

### 3.12 Writing an OGSA Service

The following steps need to be carried out to add a new service to OGSADF:

- 1) Write a WSDL PortType definition making use of OGSA types, and faults and or defining its own.
- 2) Write a WSDL Binding definition specifying the bindings that are supported by the service, e.g. SOAP, HTTP.
- 3) Write a WSDL Service definition specifying the PortTypes that the service supports, either defined in step 1 or defined by OGSI.
- 4) Implement a factory. The factory must extend the FactorySkeleton, provide callback implementations to create service instances, and define the type of the instances that can be created.
- 5) Implement the service. This could be done either by extending the ServiceSkeleton class directly or by writing a skeleton that extends from ServiceSkeleton and delegates to the implementation. Note that with the delegation skeleton approach the skeleton instance has to be returned in step 4. The delegation approach should typically be used if legacy code is to be integrated, or if container independence is sought.
- 6) Configure the factory. The container configuration file has to be made aware of the factory, the class that implements it, the schemas it supports, the schemas its instances support, weather it or its instances should be published to a remote registry etc.

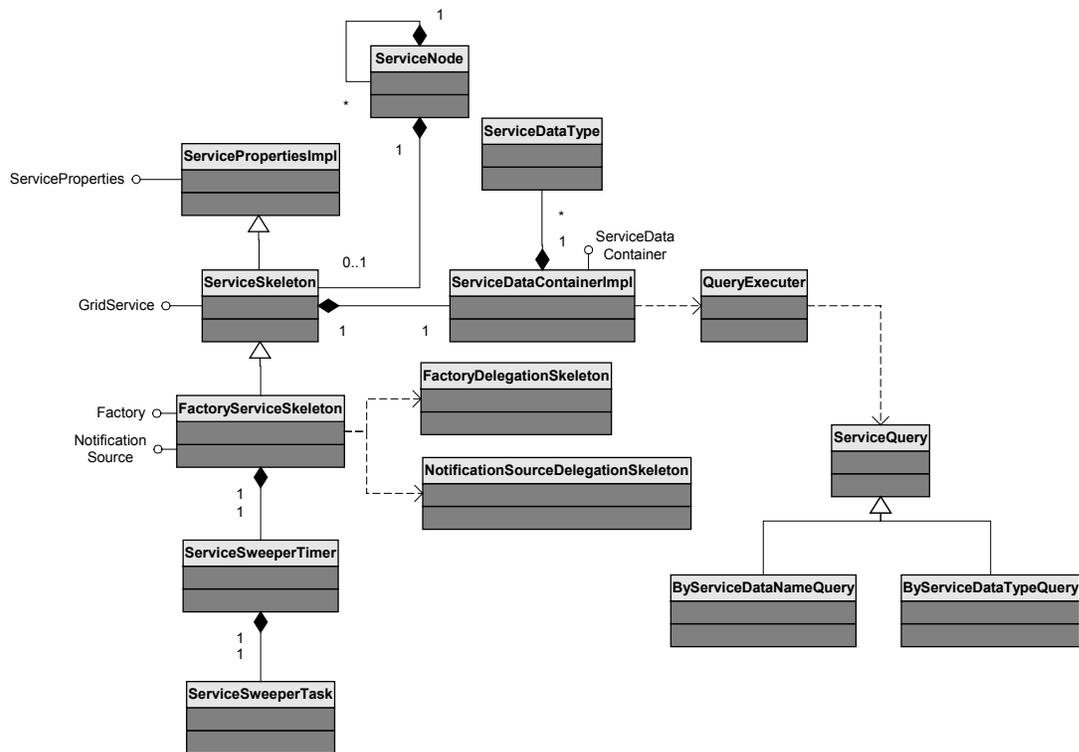


Figure 1: Server-side class relationships

## 4 Client Side Components

An OGSADF client, in general, only needs to get a Grid Service Handle (GSH) as defined in [2], then invoke a HTTP GET with the ?WSDL option set to get the Grid Service Reference (GSR),

or alternatively use the GS HandleMap interface if the reference needs to be refreshed. The WSDL document returned is intended to be standardized enough to let any WSDL enabled Web services tool construct proxies for its target hosting environment (currently .NET and Apache Axis proxy generators are supported). OGSADF in its current implementation only makes use of static WSDL to proxy generation, but future work includes supporting dynamic proxy generation and binding negotiation.

#### **4.1 Stub Generation**

Although, the most flexible approach is to dynamically generate all the proxies (at run time), based on the WSDL service definitions, there are cases where static (compile time) stub generation is more suitable. Compile time generation can give performance gains, and allows the hosting environment to provide helper classes and wrappers around the proxies to simplify the programming model, and customize user interfaces. The static compilation comes at a cost of flexibility in binding selection, and WSDL inspection. A combination of the two approaches may be used in some hosting environments, e.g. the PortType interfaces may be known at compile time, but the bindings and service definitions are discovered and inspected at run time.

#### **4.2 Discovery and Inspection**

OGSA and hence also OGSADF emphasize lightweight discovery and inspection. All services are automatically made targets for queries and inspection requests by the framework, and some higher level services enable the service publish-, and lookup- (e.g the registries described in 3.10) pattern, a.k.a. the Service Oriented Architecture (SOA) in [9]. All services written for OGSADF will expose a WSDL file for inspection, and a set of Service Data elements depending on what service is offered. This means that all OGSADF services are discovery services in some sense, which allows for dynamically browsing through and inspecting service offerings suitable to a client's needs. Some of this functionality is exemplified in the OGSADF GUI framework described in 4.5.

#### **4.3 Notification Sinks**

OGSADF clients can subscribe to notification topics provided by services implementing the NotificationSource port type. In practice this means that all clients have to embed a server-side hosting environment to expose NotificationSink services that the server can use to send out notifications. In many Web services tools this proves to be a challenge since most tools are still very client/server centric, however the peer-to-peer (P2P) [10] architecture is rather the rule than the exception in grid environments, and hence has to be supported by the framework. OGSADF provides a lightweight, embedded hosting environment specifically targeted towards client side sinks, as well as a set of APIs making it easy for clients to register for notification topics, and to receive callbacks. A NotificationSinkManager allows clients to register callback objects and listen for local events triggered by a remote message arriving. The NotificationSinkManager also hides the details of setting up sink services and multiplexing a collection of client sinks over the same physical transport port.

#### **4.4 GUI and Demo Framework**

In order to demonstrate the dynamic nature of OGSADF, and to show how discovery and inspection can be used to look up, query and use services as they become available, a GUI framework is provided. The core part of the GUI is minimal, it only understands how to get a GSR from a GSH, and how to map a WSDL port type discovered in a GSR to a GUI panel. Service providers can easily add support for new port types through configuration. All GUI panel implementations are given a GSH and, a default GSR endpoint in order to make invocations on a

service supporting a particular port type. These panels will hence typically have the stub code for that port type compiled in (static stub generation as described in 4.1) to provide a user-friendly interface.

#### **4.5 Writing an OGSA Client**

Many different models are available to the client for invoking on OGSADF exposed services. Any tool supporting WSDL may have its own programming model. Below we describe one possible model supported by the current implementation.

- 1) Generate a proxy from a WSDL definition without instance specific information, but with port type and binding information.
- 2) Get a GSH from a service supporting the port types used in step 1 from a well know registry.
- 3) Invoke HTTP GET with the ?WSDL option on the GSH to get the GSR, and extract the endpoint URL.
- 4) Pass in the endpoint URL found in step 3 to the proxy generated in step 1, and start making invocations on the service.

### **5 Development Environment**

Up until now, we have only described OGSADF from a high-level design point of view in order to be abstract enough to apply to many hosting and development environments. In this section we present the actual environments supported by the current OGSADF implementation.

#### **5.1 Hosting Environments**

Three different hosting environments are supported: 1) servlet container, 2) standalone HTTP container 3) embedded HTTP container. All these containers make use of the Apache Axis[11] Java SOAP Engine. Since the OGSADF code only interfaces with the Axis Engine, and the service provider code only interfaces with OGSADF, both of these are unaware of the hosting environment they are running in. In other words, both the OGSADF jar and the service jars can easily be loaded into all hosting environments without any modifications. Further, the configuration files used for the different hosting environments are the same.

#### **5.2 Configuration**

The configuration is based on the Apache Axis Web Services Deployment Descriptor (WSDD) XML language. Some extensions are provided to make it easy to access global configuration for a container, and to get configuration specific to persistent services. OGSADF, for instance recognizes the “persistent” parameter option flag within a service element in WSDD, and then sets up and initialize the service as a persistent service (during container bootstrapping).

#### **5.3 Build and Test Environment**

The build environment is based on Jakarta Ant [12] compliant XML make-file targets. Targets are used to e.g. generate stubs using the Apache Axis WSDL2Java tool, compile the code, generate API documentation, optionally deploy the code in a Jakarta Tomcat [13] Servlet Engine server, run the demo based on the GUI framework, and run the tests using Junit [14] extensions to Ant. The tests can be run using the embedded hosting environment in order to test all the client and server side code in a controlled environment (e.g. both residing in the same JVM), or against a Servlet Engine server using the Servlet based hosting environment.

## **5.4 Debugging and Logging**

OGSADF makes use of the same logging framework as Apache Axis, which is the Jakarta Commons Logging API [15]. This API is a thin bridge, which allows you to plug in specific logging APIs. Currently it is used with Log4J [16] in OGSADF. Different log targets like files, and consoles can be configured with thresholds, and class module log level generation can be customized.

## **5.5 Additional Tools**

In addition to the tools mentioned, OGSADF also makes use of WSDL4J [17] to parse WSDL files; and Java COG [18], IAIK [19], as well as GSI extensions to Tomcat and Axis for the GSI implementation.

# **6 Security**

In our implementation of the OGSA framework we rely on transport layer security. With transport layer security each hosting environment must be modified/adapted to support the specific transport protocol. Currently we provide a GSI-enabled HTTP protocol as our secure transport layer and support GSI extensions for the Tomcat engine and the standalone HTTP server. Since the GSI-enabled HTTP protocol is not entirely compatible with the standard SSL-enabled HTTP protocol we use 'httpg' as its protocol name to distinguish between the two protocols. The GSI extensions for the Tomcat engine integrate seamlessly with an existing Tomcat server. No code modifications are required for Tomcat or servlet code. Once the GSI support is enabled, any servlet can be accessed with GSI. If delegation was performed during GSI authentication, the delegated credentials are exposed as part of the servlet environment. This way, any servlet invoked over the 'httpg' protocol is able to get hold of the delegated credentials if it needs to. Both in the Tomcat engine and the standalone HTTP server authorization is controlled via a grid-map file. However, in our current implementation all services execute as the same user as the hosting environment is running as. All authorized users are able to access all services.

## **6.1 Service Security**

With transport level security, any service in the framework can be accessed securely without any source code modifications to the service. A service can get hold of the delegated credentials (if delegation was performed) by retrieving them from the current context of the SOAP engine (just like any other context data). In fact, in the case where the SOAP engine is running within Tomcat, the entire servlet environment is part of the context of the SOAP engine. To ensure that a service can be accessed securely, the soap:location element in the WSDL of the service can be modified to specify 'httpg' as the access protocol of the service. Once a service instance is created or a service is registered with a registry it will be accessible with the 'httpg' protocol.

## **6.2 Client Side Security**

For the client side we provide a GSI-enabled HTTP protocol transport module that easily integrates with the Axis SOAP engine. The GSI attributes such as the credentials, the authorization type, and the desired delegation mode can be set on any client by setting

appropriate GSI properties. The GSI authentication will be performed transparently once a service is invoked. Of course, a client must have valid credentials and the service handle must specify the appropriate protocol type.

## 7 Future Work

Future work includes support for new hosting environments like J2EE[20], and .Net[21]; support for dynamic binding selection using WSIF; and various additional bindings, e.g. GSI over SOAP.

In addition to this we are also working on moving over existing grid services for resource allocation management [23], and data transfer services [24] to OGSADF.

A C-based hosting environment is also to be prototyped.

## 8 Acknowledgements

We would like to thank Ravi Madduri, John Bresnahan, and Peter Lane for their help on prototyping initial versions of this framework.

## 9 References

1. Foster, I., Kesselman, C., Nick, J. and Tuecke, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002, [www.globus.org/research/papers/ogsa.pdf](http://www.globus.org/research/papers/ogsa.pdf).
2. Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S. and Kesselman, C. Grid Service Specification. Globus Project, 2002, [www.globus.org/research/papers/gsspec.pdf](http://www.globus.org/research/papers/gsspec.pdf).
3. Gudgin, M., Hadley, M., Moreau, J. and Frystyk Nielsen H. SOAP Version 1.2 Part 1: Messaging Framework. W3C, Working Draft, 2001, [www.w3.org/TR/soap12-part1/](http://www.w3.org/TR/soap12-part1/)
4. Christensen, E., Curbera, F., Meredith, G. and Weerawarana., S. Web Services Description Language (WSDL) 1.1. W3C, Note 15, 2001, [www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl).
5. Brittenham, P. An Overview of the Web Services Inspection Language. 2001, [www.ibm.com/developerworks/webservices/library/ws-wsiloover](http://www.ibm.com/developerworks/webservices/library/ws-wsiloover).
6. Bray, T., Hollander, D. and Layman, A. Namespaces in XML, W3C, Recommendation, 1999, [www.w3.org/TR/REC-xml-names/](http://www.w3.org/TR/REC-xml-names/)
7. Fallside, D.C. XML Schema Part 0: Primer. W3C, Recommendation, 2001, [www.w3.org/TR/xmlschema-0/](http://www.w3.org/TR/xmlschema-0/).
8. Coward, D. Java Servlet Specification Version 2.3. Sun Microsystems, 2001.
9. Graham, S., Simeonov, S., Boubez, T., Daniels, G., Davis, D., Nakamura, Y. and Neyama, R. Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI. Sams, 2001.
10. Oram, A. Peer-to-Peer: Harnessing the Powers of Disruptive Technologies, O'Reilly, 2001.
11. Apache Axis, The Apache SOAP Project. [xml.apache.org/axis](http://xml.apache.org/axis)
12. Jakarta Ant, The Jakarta Project. [jakarta.apache.org/ant/](http://jakarta.apache.org/ant/)
13. Jakarta Tomcat, The Jakarta Project. [jakarta.apache.org/tomcat/](http://jakarta.apache.org/tomcat/)
14. JUnit, [www.junit.org](http://www.junit.org)
15. Jakarta Commons, The Jakarta Project. [jakarta.apache.org/commons](http://jakarta.apache.org/commons)
16. Log4J, The Jakarta Project. [jakarta.apache.org/log4j](http://jakarta.apache.org/log4j)
17. WSDL4J, IBM. [www-124.ibm.com/developerworks/projects/wsdl4j/](http://www-124.ibm.com/developerworks/projects/wsdl4j/)

18. von Laszewski, G., Foster, I., Gawor, J., Smith, W. and Tuecke, S. *ACM 2000 Java Grande Conference, 2000*. [www.globus.org/cog](http://www.globus.org/cog)
19. IAIK, [jcewww.iaik.tu-graz.ac.at/](http://jcewww.iaik.tu-graz.ac.at/)
20. Shannon, B. Java 2 Platform Enterprise Edition Specification, v1.3. Sun Microsystems, 2001.
21. .net. Microsoft. <http://microsoft.com/net/>
22. Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C., Grid Information Services for Distributed Resource Sharing. In *10th IEEE International Symposium on High Performance Distributed Computing*, (2001), IEEE Press, 181-184
23. Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W. and Tuecke, S. A Resource Management Architecture for Metacomputing Systems. In *4th Workshop on Job Scheduling Strategies for Parallel Processing*, Springer-Verlag, 1998, 62-82.
24. Allcock, W., Chervenak, A., Foster, I., Kesselman, C., Salisbury, C. and Tuecke, S. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Datasets. *Journal of Network and Computer Applications*, 23:187-200, 2001.