

Globus Toolkit 3 Core – A Grid Service Container Framework

Authors:

Thomas Sandholm sandholm@mcs.anl.gov

Jarek Gawor gawor@mcs.anl.gov

Date:

2 July 2003

Abstract

The core infrastructure of Globus Toolkit 3 (GT3 Core) is based on the Open Grid Services Infrastructure (OGSI) primitives and protocols. The main design goal has been to make the OGSI technology easy to use, reuse, and extend when developing new Grid applications. In addition to an open source implementation of all the OGSI defined protocols, which can be used as a reference for other OGSI implementations, GT3 Core also provides various hosting environments built around a container abstraction. The container enables portable OGSI compliant Grid services to be developed without any knowledge of the underlying protocols and transport bindings. Our implementation has been built on top of, and hence leverages, state-of-the-art Web services technology. GT3 Core can be seen as the set of building blocks we consider essential for all Grid applications. The OGSI primitives implemented offer support for soft-state management, inspection, notification, discovery and global instance naming. Additionally, GT3 Core is comprised of a security infrastructure, and a number of system-level services, such as logging-, management-, and administration Grid services. Furthermore, a development environment with code generation tools to simplify and accelerate development of new services, is included

Contents

PART I - Background	3
Introduction.....	3
1 What is a Grid Service?	3
2 Web Services vs. Grid Services.....	3
3 Service Composition.....	4
4 Standards based Systems Integration and Performance.....	4
5 Schema Design.....	4
5.1 SOAP Encoding Models.....	5
5.2 Extensibility Elements	5
5.3 WSDL Templates and Run-Time Decoration.....	5
PART II - Overview.....	6
6 GT3 Core at a Glance	6
7 Grid Service Container	7
8 OGSF Reference Implementation.....	8
9 Security Infrastructure	8
9.1 Transport-Level Security	8
9.2 Message-Level Security.....	9
9.3 Declarative Security.....	9
9.4 Programmatic Security.....	9
10 System-Level Services.....	9
11 Hosting Environments	10
11.1 J2EE Support	10
12 Virtual Hosting Environment Framework	10
PART III – Programming with GT3	12
13 Java Programming Model.....	12
13.1 Server Programming Model.....	12
13.2 Client Programming Model	14
14 Service Data.....	14
15 State Management.....	16
16 Development Environment and Tools	18
PART IV – Standards & Future Work.....	19
17 Endorsed Java Specifications.....	19
18 Endorsed XML Specifications.....	19
19 Future Work.....	20
Acknowledgements.....	21
References.....	21

PART I - Background

Introduction

Globus Toolkit® 3 (GT3) is based on a new core infrastructure component compliant with the Open Grid Services Architecture (OGSA) [1], and is an open source implementation of the Open Grid Services Infrastructure (OGSI) [2]. The implementation is intended to serve as a proof of concept for OGSI, and to be used as a reference for other implementations. We refer to the common core infrastructure component in GT3 as GT3 Core. Some efforts have been made to standardize parts of this Java based framework within the Global Grid Forum [3].

GT3 Core offers a run-time environment capable of hosting Grid services. The run-time environment mediates between the application and the underlying network, and transport protocol engines. GT3 Core also provides development support including programming models for exposing, and accessing Grid service implementations.

1 What is a Grid Service?

The term Grid service has been used in many different contexts, and generally means any service offered to clients in a Grid environment.

In this document we adopt a more stringent definition of a Grid service, to allow us to distinguish our Grid service framework from other service frameworks available in various Grid environments. Hereinafter a Grid service is a service that is compliant with the Open Grid Services Infrastructure specification [2], and which exposes itself through a Web Services Description Language (WSDL) interface [4].

So, a Grid service in this context could be viewed as a standard Web service adapted to the requirements typically found in a Grid environment. In order to make this as transparent as possible to Grid application developers, GT3 Core implements most of the OGSI functionality on behalf of users, and providers of Grid services.

2 Web Services vs. Grid Services

Web service technology available today offers a powerful application-to-application integration framework. There is a widely used, standard interface and binding description language (WSDL). And there is a popular type model commonly used in conjunction with service descriptions (XML Schema). What pieces are missing then to make Web services as widely adopted in real-world applications as more traditional distributed computing models like CORBA, and J2EE? -Well, state management, global service naming, and reference resolution are popular answers. But, more important are common behaviors and semantics defined for services using some interface description language. Both CORBA and J2EE take the approach of defining higher-level services like Messaging and Security that can be provided by service implementers and used by

applications. In OGSI, a slightly different approach was taken to define the behavior of Grid services. Instead of specifying full-fledged services, a set of service primitives are specified that can be used to build and compose services. This design ensures that there is a nucleus of behavior in common to all Grid services that can be leveraged by meta- and system level infrastructure services. This is particularly useful in Grid architectures utilizing agent and management software to dynamically cope with changing runtime properties of the network. OGSI, for instance, ensures that all services have basic introspection, discovery, and soft-state management (client driven leasing of server managed state) capabilities.

3 Service Composition

As mentioned in the previous section, OGSI was designed to enable service composition based on a core set of distributed computing primitives. This design can be leveraged when exposing the interface of your service using the standard OGSI WSDL PortTypes [2]. However, providing the same level of service composition capability on an implementation level is an entirely different problem. Luckily it is a problem that the GT3 Core infrastructure solves as well. The foundation of the GT3 Core framework is just a set of extensions to a standard Web services engine (JAX-RPC in the Java case)., These extensions consist of custom handlers and dispatchers that can be enabled or disabled through configuration. Following the same pattern we also allow services to be composed by simply plugging in various primitives into the configuration of the service (often known as the declarative programming model). If you, for example, want to have a service that can create other services, send out notifications when services are created, and provide a list of services that have been created, then you can plug in our implementation of the `FactoryProvider`, `NotificationSourceProvider`, and `ServiceGroupProvider` into your service configuration. We also allow you to use the same provider design (described in more detail in section 13.1) to make it possible for other service developers to compose and reuse your own service implementation primitives.

4 Standards based Systems Integration and Performance

As with conventional Web services, our initial focus has been on providing standards-based middleware that can be used to integrate heterogeneous systems in a Wide Area Network. The currently supported WSDL binding, SOAP over HTTP, has been tuned for minimal latency, but to be able to interoperate seamlessly and build on top of currently available Web service technology, support for high throughput of large data loads has been outside the scope of our current work. Using Globus Toolkit 2 terminology, we have focused on providing middleware to support GRAM and GridFTP control channels, but GridFTP data channel support (over a Web services protocol) is on the roadmap of future work.

5 Schema Design

We provide a set of WSDL, and XML schema definitions of interfaces based on the normative schemas in the OGSI specification [2]. To promote interoperability, we have extended these schemas to use the SOAP 1.1/HTTP 1.0/1.1 binding. A key design focus

has been to allow reuse of core type, message, fault and port type (WSDL interface) definitions. Two techniques have been used to accomplish this 1) we split up all the different levels of the definition in separate XML namespaces that can be imported both from our schemas as well as user service schemas, 2) we try to push as much logic as possible into XML Schema type definitions, and thus allowing us to make use of the XML Schema type model, which is more flexible than WSDL itself. Since SOAP over HTTP is currently the only binding with ubiquitous tooling support, it is the only binding we support for all of our interfaces. That being said, the bindings are separate from the type and interface definitions, so new bindings could easily be added as they become supported by the various Web services toolkits.

5.1 SOAP Encoding Models

In WSDL, two orthogonal encoding techniques can be used to communicate over SOAP: 1) SOAP encoding, and 2) literal encoding.

SOAP encoding refers to SOAP messages complying with the Section 5 Encoding Rules in [8], and literal encoding refers to encoding left up to the WSDL type element definitions. Literal encoding is more flexible than SOAP encoding since it does not put any restrictions on the format of the XML payload. This is a critical requirement of the OGSi specified Service Data Elements, and hence we define all of the core port type definitions using literal encoding. However, since the majority of today's Web services deployments use SOAP Encoding we allow any services built on top of our framework to use SOAP Encoding for their interfaces.

5.2 Extensibility Elements

Because most of our WSDL interface definitions are intended to be reused by higher level Grid services, we needed to support a flexible model for extending and plugging in additional typed XML payloads inside the predefined framework messages. The technique used to achieve this is based on the XML Schema any construct. We also provide both clients and services with an option to customize how they want to deserialize an incoming XML Schema any payload. The XML can be deserialized into 1) a typed programming language object, 2) a DOM Element Infoset, or 3) a string buffer.

5.3 WSDL Templates and Run-Time Decoration

When providing a new service on top of our framework, the first step is to provide a WSDL definition of the service type. This definition can import and reuse the provided core types and interfaces, and it is used at runtime by the framework when the service is activated in order to receive requests. During service activation, deployment and configuration settings are used to decorate the WSDL definition with instance specific information before it is returned to the client. One example of such instance specific information is the endpoint URL used to contact the specific instance. This endpoint URL may, for example, depend on factory creation operation data.

PART II - Overview

6 GT3 Core at a Glance

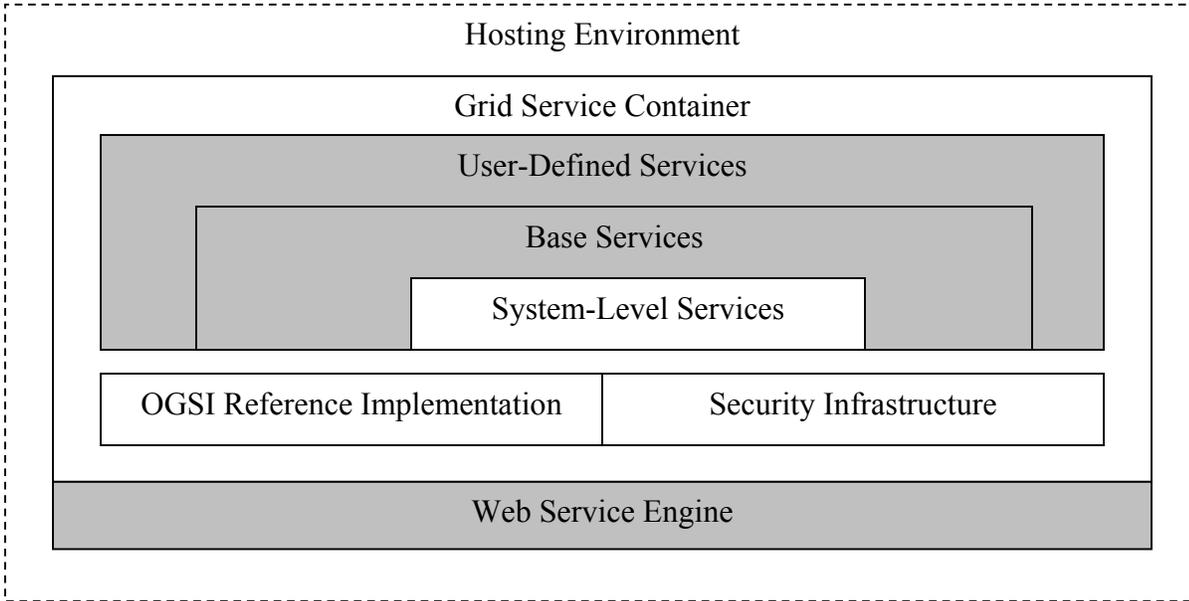


Figure 1: GT3 Core Architecture

The white boxes in Figure 1 represent components provided by GT3 Core. Together they make up what we consider the essential building blocks for Grid services. The *OGSI Reference Implementation* provides implementations for all OGSi specified interfaces, as well as APIs and tools to make it easier to develop OGSi compliant services. The *Security Infrastructure* implementation provides SOAP as well as transport level message protection, end-to-end mutual authentication, and single sign-on service authorization; essentially a rendering of the GSI implementation known from Globus Toolkit 2 in an OGSi environment. These two building blocks do not provide any run time services but serve purely as a base for other services. GT3 Core, however, also contains some infrastructure level run-time services that are generic enough to be used by, and in conjunction with all other Grid services. These so called *System-Level Services* are built on top of the OGSi Reference Implementation as well as the Grid Security Infrastructure. GT3 also ships a number of higher-level or *Base Services*, like Program Execution, Data Management, and Information Services. A detailed discussion of these services is outside of the scope of this document, since they are not part of the generic core infrastructure. Base Services are typically built on top of the OGSi and GSI components, as well as in some instances the System-Level Services component. *User-Defined Services* is the term for higher-level services, not included in the toolkit, that are built on top of any subset of GT3 components including Base Services

All these services and primitives interact with an abstract OGSi run-time environment we call the *Grid Service Container*. The purpose of the container is to shield the application

from environment specific run-time settings, such as what database is used to persist service data. The container also controls the lifecycle of services, and the dispatching of remote requests to service instances. An important design goal of the container has been to make it as implementation agnostic as possible and thus allow for many different implementations. We also hope that the interfaces that constitute the container will be standardized in order to make it possible to host services in alternative implementations, similar to how EJBs can be hosted in alternative J2EE implementations today. The container extends, as well as encapsulates the interfaces defined by a standard *Web Service Engine*, which is responsible for implementing XML Messaging.

Finally, the Web Service Engine and Grid Service Container are hosted in a *Hosting Environment*, which implements traditional Web Server functionality such as the transport protocol (e.g. HTTP). See section 11 for an overview of hosting environments currently supported by GT3 Core.

7 Grid Service Container

Providing system level functionality on behalf of, and transparent to users is a well-proven concept in the field of Application Server Providers, and it is most typically exemplified in the Enterprise Java Bean (EJB) component model [5]. The EJB model is based on the notion of a container that hosts various application/business logic components. The application logic components can be deployed into the container with varying quality of service (QoS) and behaviors implemented by the container, e.g. transaction-, security-, and database management. We adopt the same model for our work, where the QoS and behaviors implemented by the container are defined in the OGSF specification [2].

Further, our container must be flexible enough to be deployed into a wide range of hosting environments in order to accommodate the heterogeneity of today's Grid deployments. For example, a Grid service could be implemented as an enterprise B2B application serving a large number of concurrent users, as well as a lightweight entry point into a Grid scheduling system for batch submissions of a very restricted number of jobs to be executed in a remotely sandboxed user account. So, if a service is developed to comply with our container interface contract, it can be deployed in all hosting environments supported by GT3. In order to make this goal a reality, we had to make the programming models implied by the container contract simple enough to allow very lightweight deployments, but still provide some added value over today's state-of-the-art Web services programming models. In particular, this meant that we did not want to mandate (but neither preclude) runtime environment infrastructures support such as the CORBA Portable Object Adapter (POA) [6], or the J2EE Enterprise Bean Container [7]. Our container, however, makes use of many concepts from both the POA, as well as the EJB programming models. This allows our container to easily be used as a dispatcher into implementations complying with any of these higher-level models.

The design decisions mentioned so far are very similar to what is offered in state-of-the-art Web service toolkits, and programming models; so what additional functionality does

our container provide? –There are three major functional areas, all being integral parts of our core container implementation, that best summarize the value added in our Grid services container compared to conventional Web services toolkits:

- 1) Light-weight service introspection, and discovery supporting both pull and push information flows,
- 2) Dynamic deployment, and soft-state management of stateful service instances that can be globally referenced using an extensible resolution scheme,
- 3) A transport independent Grid Security Infrastructure supporting credential delegation, message signing, and encryption as well as authorization

8 OGSi Reference Implementation

The OGSi Reference Implementation is a set of primitives implementing the standard OGSi interfaces, such as: GridService, Factory, Notification(Source/Sink/Subscription), HandleResolver, ServiceGroup(Entry/Registration). Typically, a service provider does not have to interact with these interfaces directly, but just needs to configure them for the service to be provided. The implementation of the GridService interface is essentially our base container implementation, while the Factory interface implements most of the state management in the container. These two implementations are thus fundamental parts of our container implementation and not likely to be replaced by other implementations (although they could be extended by service providers). The implementation offered for the other primitives should, however, be seen more as a reference implementation that could easily be replaced by more robust implementations (even though the provided implementations will suffice in most scenarios). Our implementation makes sure that all the mandated service data are populated with the required values and that services making use of these components have the expected OGSi behavior.

9 Security Infrastructure

GT3 Core provides transport- and message-level security. Both are based on GSI and PKI standards. The transport-level security was initially provided for compatibility with GT3 C clients. The use of transport-level security is discouraged and its support is not guaranteed in future GT3 versions. We recommend the use of message-level security instead. The message-level security support is based on the WS-Security [9], XML-Signature, and XML-Encryption standards.

The GT3 Core security infrastructure is based on the Java Authentication and Authorization Service (JAAS) framework. It allows Java Grid Services to remain independent from underlying authentication mechanisms. We also provide declarative and programmatic security.

9.1 Transport-Level Security

For transport-level security, we provide a GSI-enabled HTTP-based protocol called “httpg”. The protocol is essentially equivalent to the “https” protocol, but provides support for credential delegation.

9.2 Message-Level Security

Message-level security is done entirely on the SOAP level. Therefore, it can be used with any underlying transport that supports the SOAP protocol.

We provide two message-level security mechanisms: GSI Secure Conversation (session-based) and GSI XML Signature (per-message security). With GSI Secure Conversation a client first establishes a security context with the service using a Secure Conversation Service. Once the security context is established, the client uses it to sign or encrypt the requests. The actual security context is established using the GSS-API. Although GSS-API supports multiple security mechanism only the GSI protocol is currently supported. A shared secret key is not required for the GSI XML Signature method. The client simply uses an X.509 certificate to sign the request. With the GSI XML Signature method a request can be passed through any number of intermediary servers, which will all be able to look at and validate the request.

9.3 Declarative Security

In order to shield the service provider from the details of the security implementation, we provide a declarative approach to customizing the security requirements of a service. This is achieved by allowing the configuration of the security requirements of a service using service deployment descriptors. A descriptor can be used to define a list of authentication mechanisms that a client must use to access a service. It can also be used to define the identity that a given method of a service should run as, as well as to set the authorization method that a service requires.

9.4 Programmatic Security

GT3 Core also provides a security API in cases where declarative security is not sufficient to express the security properties of a service. The security API mainly consists of the *org.globus.ogsa.impl.security.SecurityManager* and *org.globus.gsi.jaas.JaasSubject* classes. The *SecurityManager* class defines methods that allow the principal name of remote user to be determined, whereas the *JaasSubject* class defines methods that allow the JAAS Subject object associated with the current thread of execution to be obtained.

10 System-Level Services

GT3 Core includes some general-purpose services to facilitate the use of Grid services in production environments; the *Admin*-, *Logging*-, and *Management* services. The services work independently of each other, but they can also easily be used in conjunction with one another from a single administrative client.

The *Admin* service is used to ping a hosting environment, and to shut down a container gracefully. The *Logging Service* allows you to modify log filters and group existing log producers into more easily manageable units at run time. Additionally, the *Logging Service* can be used to monitor a size-adjustable backlog of recently logged messages, or to subscribe to a customizable view of log message events. Log filter changes can be runtime targeted, or made to persist across service container lifecycles. The *Management Service* provides a monitoring interface for the current status, and load of a service

container. It also allows simple management operations to, for instance, activate and deactivate service instances.

11 Hosting Environments

We currently support four different hosting environments for Java: 1) Embedded, utility to be used in clients or lightweight servers to expose Grid services; 2) Standalone, lightweight server program (essentially the embedded hosting environment with an additional server mainline with startup options); 3) Servlet, our container inside of a standard Java Servlet Engine[12]; 4) EJB, our container inside of an EJB application server.

Depending on security, reliability, scalability, and performance requirements, any of these hosting environments can be picked as target environment for Grid service implementations. Note that 1 through 3 are completely transparent to the Grid service implementation, whereas 4 is transparent to existing EJB implementations.

11.1 J2EE Support

The Servlet, and EJB environments described above are compatible with the Web container, and the Enterprise Bean container as defined in J2EE. J2EE 1.4 [7] furthermore mandates JAX-RPC support, which we in turn also support in our container. J2EE currently only provides Web services support for Stateless Session Beans [13], which is one reason why we have built a dispatcher generator for our framework that takes care of state management, and Grid services behavior on behalf of an Enterprise Bean. The EJB can be either a Stateful Session Bean, or an Entity Bean. This model of integration has the additional benefit of allowing us to leverage the core infrastructure built for all the other hosting environments. Furthermore, we provide integration with JMS [7] messaging in order to leverage the QoS of a JMS compatible queuing system such as MQSeries.

12 Virtual Hosting Environment Framework

A Grid service is typically hosted in the same container as its factory. There are however cases when you want to distribute Grid services created by a factory among a number of remote containers. Examples of use cases include load balancing and user account sandboxing. The distribution often depends on server specific QoS requirements and should thus not be exposed to the clients, who should be able to interact with the services as if they were all in the same container. The architecture could also simplify firewall traversal because it allows all requests to be routed through a single entry point.

We have implemented infrastructure that helps you set up a virtual hosting environment for your service. It includes routing handlers; a virtual hosting environment redirector; reference and handle rewriters; a customizable hosting environment starter and redirection exception APIs. We have also extended our security implementation to allow, for instance, secure end-to-end context establishment negotiations between the client and

the local hosting environment, containing the actual service, to be redirected via the virtual hosting environment.

A typical redirection scenario consists of the following steps:

1. The client looks up a factory handle of the service in the virtual hosting environment.
2. The client sends a create service request to the factory
3. The factory throws a redirect exception containing some information to base the redirection decision on.
4. The router intercepts the redirect exception and passes it on to a hosting environment starter.
5. The hosting environment starter that is configured for the service gets the exception and creates a target endpoint based on the information in the exception, and then starts the local hosting environment, which will host the actual service.
6. The router now forwards the create service message to the newly constructed and started endpoint, and relays the result back to the initial client as if the reply had come from the factory initially contacted. In this step a routing table is updated in the virtual hosting environment to allow all subsequent requests to the service to be redirected to the correct local hosting environment.
7. A call on the service is made to the virtual hosting environment. It finds a routing entry and hence forwards the request on to the local hosting environment. Note that in order for this step to be performed as expected the handles and references passed to the client must all be rewritten to point to the virtual hosting environment. This is done in the local hosting environment based on routing information passed in SOAP headers. Apart from the routing entry, which is typically maintained on a per-hosting-environment basis, the virtual hosting environment does not maintain any information about the services created in the local hosting environment. This is a very important feature in order to achieve scalable load balancing.

PART III – Programming with GT3

13 Java Programming Model

We have focused on providing support for writing Java Grid services in GT3 Core. The main goal has been to make it as easy as possible to write your own services and deploy them into the container framework without having to worry about providing mandated OGSi functionality.

13.1 Server Programming Model

The core Grid service server programming model is depicted in Figure 2.

Grid Service Base

A GridServiceBase object is the base of all Grid services and implements the standard OGSi GridService PortType. It also provides APIs to modify instance specific properties (either set by configuration or at run time), as well as APIs for querying and modifying service data. The base functionality can be seen as the functionality known at development time. At deployment and run time, additional functionality can be added in using Operation Providers, described in the next section. In GT3 Core we provide two implementations of the GridServiceBase interface. One to be used for transient (created by an OGSi Factory) Grid services called GridServiceImpl, and another one called PersistentGridServiceImpl to be used for persistent (created through configuration entries and always available in a container) Grid services.

Operation Providers

A service can be created by simply extending from GridServiceImpl, or PersistentGridServiceImpl, but it is not recommended because of its limited flexibility. Recall that the base implementation ‘locks’ the service at development time into supporting a certain interface and it is then hard to reuse or customize the features of this service at deployment- and run time. By directly inheriting from one of our implementation classes you also create a dependency on our container implementation, which would make it harder to port your service to other OGSi compliant containers in the future. The solution adopted to solve these problems in GT3 Core is called the *Operation Provider* model or the dynamic delegation model. Instead of extending from the base implementation classes, you only provide an implementation of the operations (as defined in WSDL) that you would like to expose to remote clients. Your provider can then be plugged into any service at deployment time through configuration to make the service support your operations.

Grid Service Callback

The GridServiceCallback interface defines a number of lifecycle management callbacks that you can optionally implement to manage the state of your service. See section 15 (State Management) for further details about this interface.

Factory Callback

A factory callback can be implemented to provide custom factories for your services. It can, for instance, be used to create services in remote hosting environments. Most implementations are, however, likely to use the dynamic factory callback implementation we provide, which allows you to, through configuration, specify the implementation class that the factory should create.

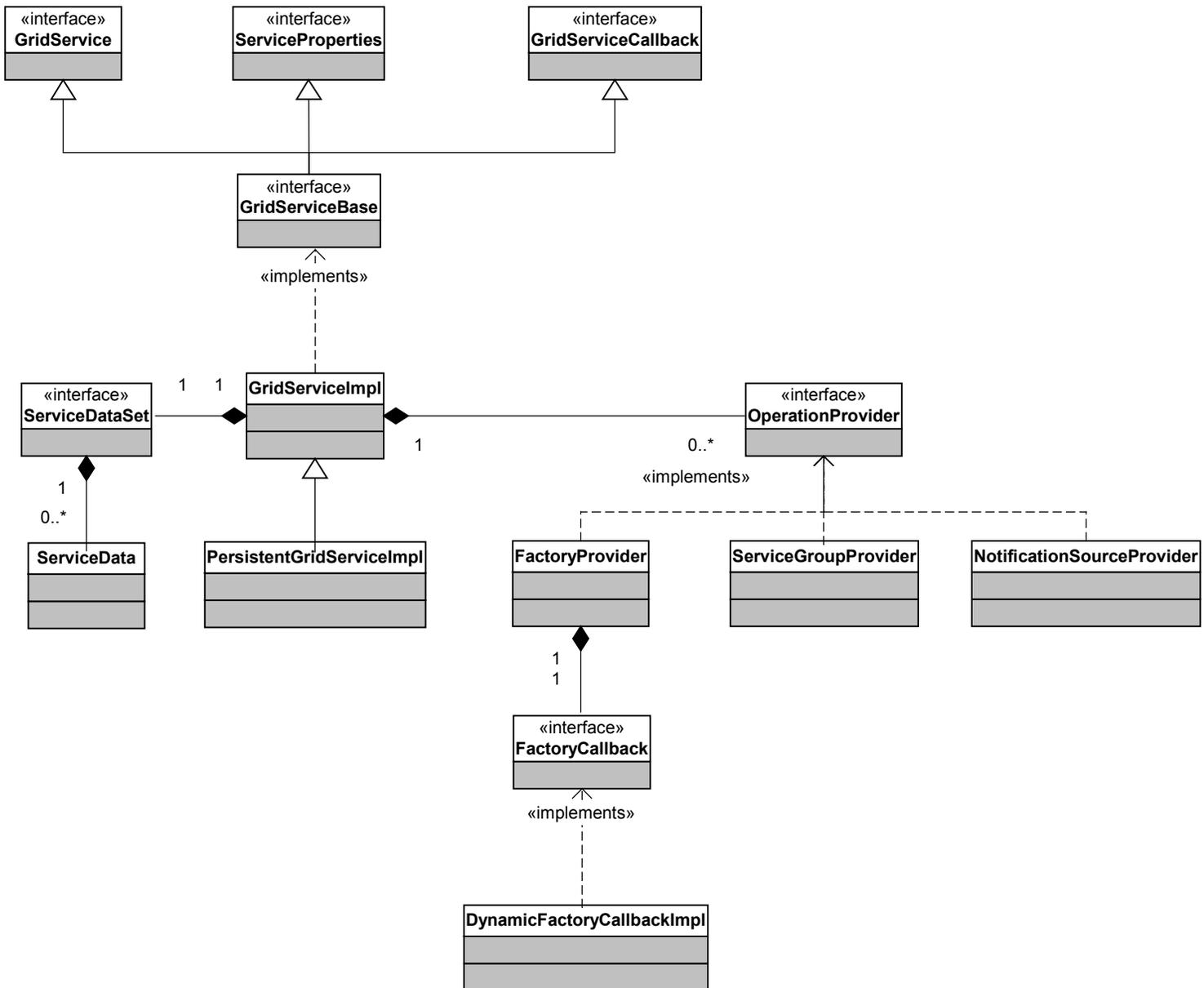


Figure 2: Server Programming Model

13.2 Client Programming Model

The Grid Service Client programming model is depicted in Figure 3.

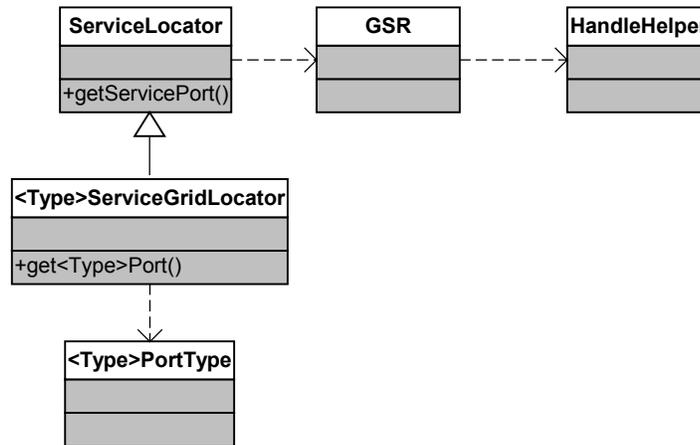


Figure 3: Client Programming Model

A Grid service client can be written directly on top of the JAX-RPC client APIs. However, for convenience and ease of use, we provide a number of utility classes simplifying GSH to GSR resolution, and GSR introspection. Further, we provide a custom stub generator extending the JAX-RPC stubs to integrate these utilities seamlessly into the client programming model. A client will typically get a handle from a registry, or through some out of band mechanism, to a well-known service instance, such as a factory. The handle is passed into a ServiceLocator that constructs a proxy, or stub, responsible for making the call using the network binding format defined in the WSDL for the service. The proxy is exposed using a standard JAX-RPC generated PortType interface (sometimes referred to as Service Endpoint Interface). Note that this interface is identical to the one used on the server side to implement the service.

14 Service Data

In addition to populating all services with the OGSi mandated service data, we also provide APIs to dynamically add service data that conform to the service data descriptions in WSDL (if present) to your service instance. A service data wrapper can either hold the actual service data values or be associated with a callback that will be called every time the values of that service data entry are requested. As an alternative approach, when you generate your Grid service from a java class, we also allow you to add in meta data comments (doclets) that will result in service data being automatically created and hooked into a callback to your java implementation class.

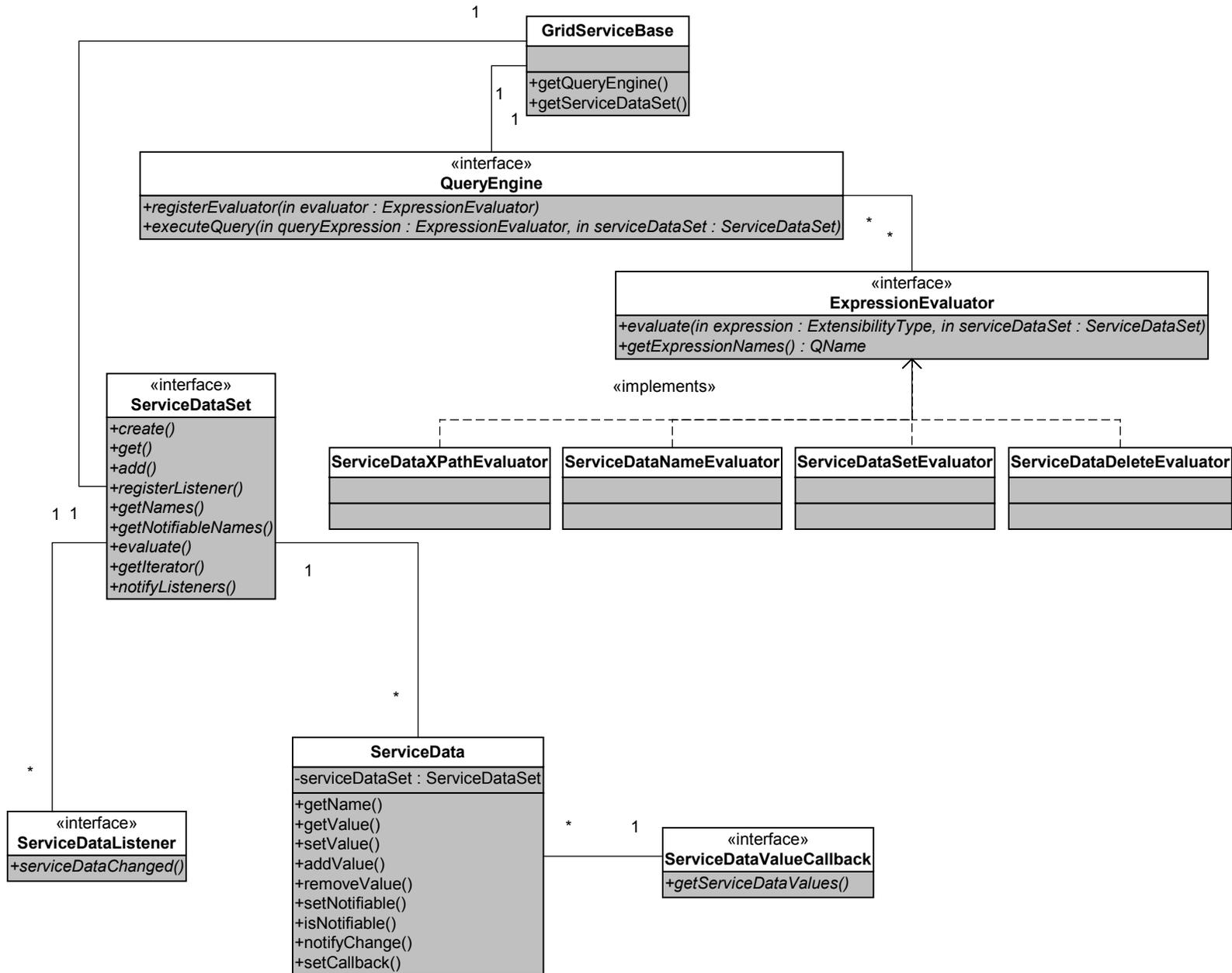


Figure 4: Query Framework

The Grid Service Specification [2] provides a very open ended interface for executing queries and subscribing to notifications. As a direct result of this we want to provide the same kind of flexibility to the Grid service developers, without introducing unnecessary complexity. The result of this work is depicted in figure 4. The framework allows you to plug in your own evaluators for notification and query expressions, as well as providing your own query engine implementation.

The central APIs are the `ServiceDataSet`, which holds a collection of service data that can be queried and subscribed to, and the `ServiceData` wrapper, which allows you to hold the service data values or implement a callback for them. The `ServiceData` class can also be used to trigger notifications on the service data. The evaluator and engine interfaces all operate on these two central APIs.

An evaluator can be used both for query execution and notification filtering, and it could be reused among all services in the container or just for specific services.

15 State Management

In order to make use of our container as a front end to more elaborate object models such as CORBA and EJB, there is some minimal support required not to make our framework become a scalability bottleneck. Automatic service activation and deactivation for more efficient and scalable memory management is supported by both CORBA and EJB. For this reason, we provide a customizable activation and deactivation framework inside of our container. When our container starts up, by default, no service instances are activated or created (not even the statically deployed persistent services). The services are then activated on first use. In cases where you can have a large number of lightweight service instances just idling and taking up memory, which may be the case for `NotificationSubscription` or `ServiceGroupEntry` services, the container can proactively deactivate these services on a Time To Live (TTL), and Least Recently Used (LRU) basis. A service that has been deactivated still holds all meta data required to activate it in memory. If you want all traces of deactivated services to be removed from memory in order to later be recovered from some persistent storage like a database, you can make use of `ServiceLoaders`. A service loader is responsible for dynamically deploying as  as activating a service in the container the first time the service is invoked. A third alternative we provide for designing scalable large-scale service applications is a concept we call lazy creation. When a factory creates the service it is neither deployed nor activated, only a unique handle is created, which can be returned to the client in an OSGI `ServiceLocator`. When the client uses the handle to resolve it into a reference, a lazy creation callback is invoked, which can be implemented to dynamically deploy and activate the service. We use the lazy creation approach in our `ServiceGroupEntry` implementation. Note that service deactivation, service loading, and lazy creation are all completely transparent to the user of the service.

A common QoS required by Grid services is that state persists even between server cycles. In order to support this we allow services to checkpoint their state, which upon reactivation can be recovered, into the web service deployment descriptors. This feature should be used with care, and should not be seen as a replacement of a database for storing more elaborate service state (just configuration properties ideally). GT3 also offers support for persisting service data in a native XML database (`xindice`). It is, however, an optional feature that is not part of GT3 Core, and it is thus beyond the scope of this document. Figure 5 gives an overview of the state management support in GT3 Core.

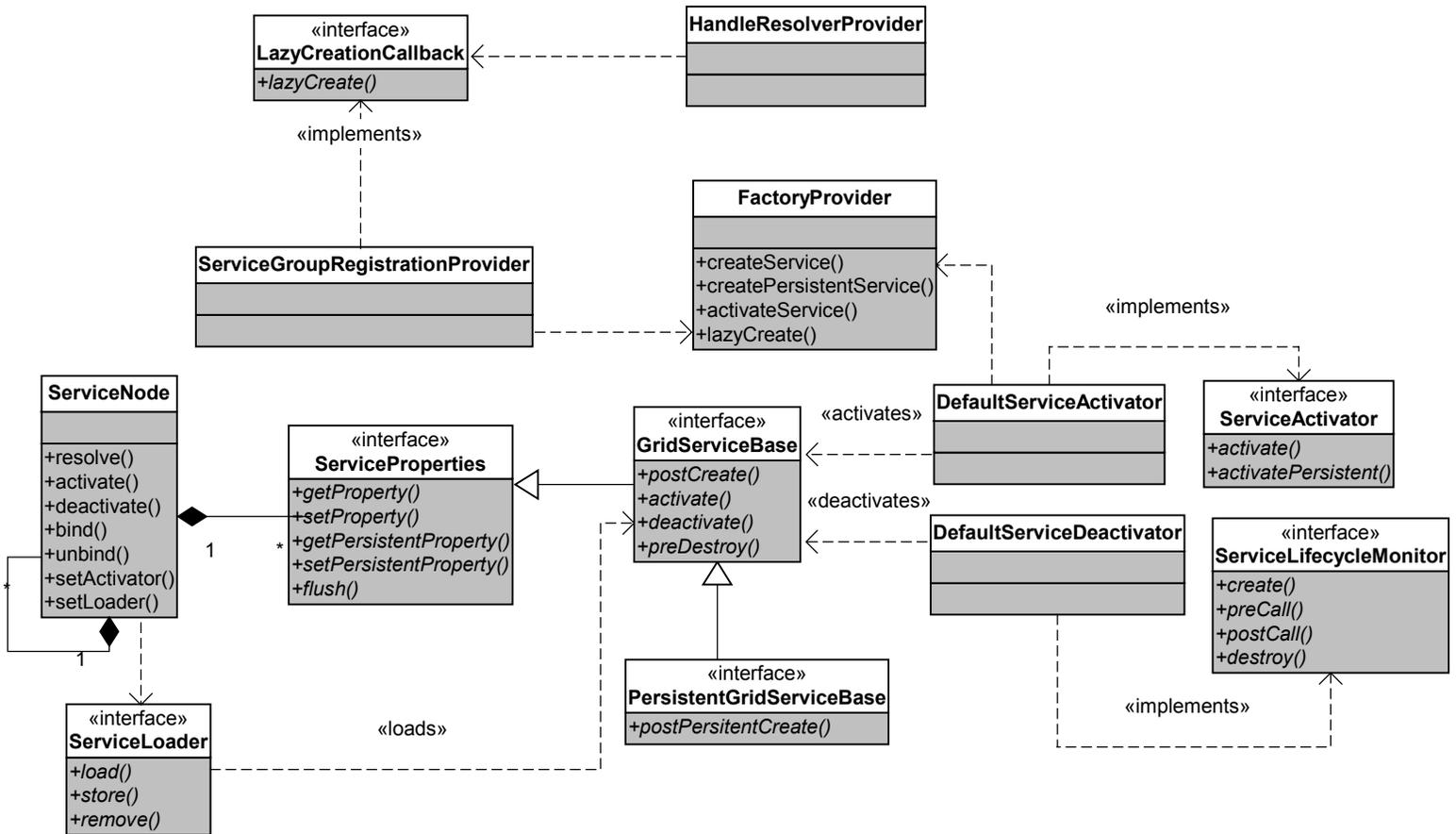


Figure 5: State Management Model

There are two types of services: persistent, and transient. A persistent service is deployed when added to the deployment descriptor through some out-of-band administration mechanism. Transient services are created by an OSGI compliant Factory service and deployed at runtime. In addition to these service types there is also a persistent, and a transient lifecycle model. A service with a persistent lifecycle model can recover properties by using the ServiceProperties interface, and it can checkpoint its state to the deployment descriptor, so that it can be reactivated after a server restart. A service with a transient lifecycle model is not able to make checkpoints into the deployment descriptor, and all properties are transient. Note, that a service of persistent type will, however, keep the configuration that was set up out-of-band even if it is using the transient lifecycle model.

Service Property	Options
Service Type	Persistent (default), Transient
Lifecycle Model	Persistent, Transient (default)
Activation	Startup, Lazy (default)
Deactivation	None (default), TTL+LRU

Table 1: Summary of state management support

16 Development Environment and Tools

GT3 Core is built on top of Apache Axis [20], and the Java CoG Kit [21].

We use JUnit [22] unit tests, and custom written ant stress tests to test the functionality of the core. We also have a compatibility test suite to test compliance to the GS specification.

Jakarta Ant [23] is used for all Java build, distribution, test and container deployment tasks. We also provide some custom Ant targets to simplify service development and deployment, such as Java interface to WSDL generation.

In order to familiarize developers with Grid services we provide a demo framework to test our sample services interactively. It is also straightforward to plug in new services into the demo. It is typically used to test a service implementation before exposing it to clients. The demo framework is written as a pluggable service browser allowing you to customize your own gui panels, or use a default introspection (DII) based gui to test your services deployed in our container.

A large number of sample services are provided to highlight specific features of the container. We also provide a step-by-step tutorial describing how to develop a Grid service from scratch using our tools and APIs.

PART IV – Standards & Future Work

17 Endorsed Java Specifications

Specification	Functionality Provided
JAXP [14]	XML Parsing (DOM and SAX)
JAX-RPC [11]	WSDL and XML Schema to Java mapping and vice versa. Client programming model.
J2EE [7]	Overall architecture of containers and APIs for Java Enterprise application servers, including Servlet Engines and Enterprise Bean containers.
EJB [5]	Container and programming model used inside of J2EE servers for backend application components
JMS [7]	Java Messaging Service used by J2EE
JNLP [15]	Java Network Launch Protocol (implemented by Sun's WebStart) used for mobile Java code and zero client deployment

18 Endorsed XML Specifications

Specification	Functionality Provided
XML Namespaces [16]	WSDL and XML Schema reuse and pluggability of extensibility elements
XML Schema [17]	SDEs and core PortType types are all defined using XML Schemas
WSDL [4]	Definition language for all Grid service interfaces and basis for GSR encoding
SOAP [8]	Currently supported WSDL binding for Grid services. Provides transport agnostic message packaging framework.
WS-Security [9]	Provides message level security using SOAP Headers
XML-Encryption [18]	Allows SOAP Bodies to be encrypted
XML-Signature [19]	Allows SOAP Bodies to be signed

19 Future Work

Security Support

We are looking into integrating our security model more tightly with the J2EE and WS security models. This includes support for JAAS, WS-Policy, and SAML. There is also ongoing work on an OGSi compliant community authorization service.

JAXB Support

We currently use the JAX-RPC 1.0 serialization framework for XML Schema any marshalling which is a bit awkward since it is tied to SOAP. Once JAXB is supported by JAX-RPC we should be able to make use of it instead to convert between Java objects and XML document instances complying with XML Schema Definitions.

High-Throughput XML Messaging

We want to investigate protocols and bindings allowing us to send large amounts of data with high throughput through Grid service interfaces. An OGSi based GridFTP service is intended to result from this effort.

Additional Grid Service Hosting Environments

So far we have been focusing on client side C support, but a lightweight OGSA based C hosting environment is also under consideration. We are also working with collaborators to provide .NET and Python hosting environments for Grid services. Once OGSi becomes more implemented interoperability test suites must be set up and agreed upon.

WS-Addressing

If the OGSi specification adopts the WS-Addressing specification to model instance endpoints, we will move to this model as well to perform the instance dispatching and routing in GT3 Core.

Acknowledgements

We would like to thank our GT3 Core contributors from IBM including: Mike Williams, Tom Maguire, Mark C Vallone, Tom Seelbach, John Wiley, and their development teams.

We would particularly like to thank Rob Seed for helping out with prototyping and designing EJB, and service data annotation support.

We are also grateful for all the invaluable feedback we received on various core features from the Globus OGSA service development team including: Ravi Madduri, Peter Lane, Mike D'Arcy, Rachana Ananthakrishnan, Ben Clifford, Joe Bester, Stuart Martin, Samuel Meder, Samuel Lang, John Bresnahan, Alain Andrieux, and Pawel Plaszczak.

Further, we would like to thank everyone on the GT3 developers and users mailing lists for their insightful comments on our work.

Finally, we would like to thank Lisa Childers for her continuous reviews of drafts of this document; and Steve Tuecke, and Karl Czajkowski for setting the technical directions of this work.

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 .
2	Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C, Maguire T., Sandholm, T., Snelling, D., and Vanderbilt, P., Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum, June 2003.
3	Sandholm, S., Tuecke, S., Gawor, J., Seed, R., Maguire, T., Rofrano, J., Sylvester, S., Williams, M. Java OGSI Hosting Environment Design – A Portable Grid Service Container Framework. Globus Project, 2002, http://www.globus.org/ogsa/java/OGSIJavaContainer_2002-07-19.pdf
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 .
5	Sun Microsystems. Enterprise Java Beans Specification, Version 2. JSR 19 www.jcp.org/aboutJava/communityprocess/final/jsr019/
6	OMG. Common Object Request Broker: Architecture and Specification, Revision 2.2. Object Management Group Document 96.03.04, 1998.
7	Sun Microsystems. Java 2 Enterprise Edition (J2EE). java.sun.com/j2ee/
8	W3C: SOAP 1.1: http://www.w3.org/TR/SOAP/
9	IBM, Microsoft, VeriSign, 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-security.asp
10	The Globus Project, 2002. http://www.globus.org/gt2
11	Sun Microsystems. Java API for XML-based RPC. JAX-RPC 1.0, JSR 101. java.sun.com/xml/jaxrpc/

12	Sun Microsystems. Java Servlet 2.3 Specification. JSR 53. www.jcp.org/aboutJava/communityprocess/final/jsr053/
13	Sun Microsystems. Web Services for J2EE, Version 1.0, JSR 109. http://jcp.org/aboutJava/communityprocess/first/jsr109/
14	Sun Microsystems. Java API for XML Processing 1.1 Specification, JSR 63. http://jcp.org/aboutJava/communityprocess/final/jsr063/
15	Sun Microsystems. Java Network Launching Protocol 1.0.1 Specification, JSR 56. http://www.jcp.org/aboutJava/communityprocess/final/jsr056/
16	Bray, T., Hollander, D. and Layman, A. Namespaces in XML, W3C, Recommendation, 1999, www.w3.org/TR/REC-xml-names/
17	Fallside, D.C. XML Schema Part 0: Primer. W3C, Recommendation, 2001, www.w3.org/TR/xmlschema-0/ .
18	Imamura, T, Dillaway, B, Simon. E. XML Encryption Syntax and Processing. W3C, Candidate Recommendation, 2002, http://www.w3.org/TR/xmlenc-core/
19	Bartel, M., Boyer, J., Fox, B., LaMacchia, B., Simon, E., XML Signature Syntax and Processing. W3C, Recommendation, 2002, http://www.w3.org/TR/xmldsig-core/
20	Apache Axis, The Apache SOAP Project. xml.apache.org/axis
21	von Laszewski, G., Foster, I., Gawor, J., Smith, W. and Tuecke, S. <i>ACM 2000 Java Grande Conference, 2000</i> . www.globus.org/cog
22	JUnit, www.junit.org
23	Jakarta Ant, The Jakarta Project. jakarta.apache.org/ant/