Creating and Managing Grid Services

Omer F. Rana[1], Ali Shaikhali[1], and Gregor von Laszewski[2]
[1]Department of Computer Science and Welsh e-Science Centre, Cardiff University, UK
o.f.rana@cs.cardiff.ac.uk
[2]Mathematics and Computer Science Division, Argonne National Laboratory, US
gregor@mcs.anl.gov

Services and the Grid

There are many definitions of what constitutes a "Grid" – the most common of which identifies a Computational Grids as: "a collection of distributed computing resources available over local or wide area networks -- and which appear to an end user as one large virtual computing system". A significant effort has been invested by the Grid community into building software systems that may be used to manage such a virtual computing system. The Grid approach is an important development for both the business and scientific community [Laszewski03], and promotes a vision for sophisticated scientific and business collaborations. Until recently most of this effort was focused on managing resource ensembles, and co-ordinating access to such ensembles in a secure and efficient manner. The focus of this work was primarily driven by the need to handle differences between different kinds of hardware architectures (Intel vs. Sun vs. SGI), operating systems (Solaris vs. Win2000 vs. IRIX) and data repositories (structured databases vs. filesystems). To establish a ``virtual organisation'', it is also necessary to allow individual systems administrators -- managing one or more resource ensembles -- to exercise their individual authority. Each of these systems administrators would have control over their resources, and would not allow some centralised system to override access rights that they have defined for external users coming to their system. The predominant interest within existing Grid systems remains on `resources' -- the hosting platforms and storage devices which may be combined together to establish a virtual system. Increasingly, here has been a recognition that it is also important to capture details about software programs and libraries which are run on such resources.

Consequently, there has been an increase in interest recently within the Grid community towards "Service Oriented" Computing. Services are seen as a natural progression from component based software development [Stevens02], and as a means to integrate different component development frameworks. A service in this context may be defined as a behaviour that is provided by a component for use by any other component based on a network-addressable interface contract (generally identifying some capability provided by the service). In the simplest case, such a contract specifies the set of operations that one service can invoke on another (a set of method calls, for instance, when a service is implemented using object-oriented technologies). Generally, such an interface may also contain additional attributes which are not specifically related to the functionality of the called service, and can include aspects such as:(1) the performance of the service, (2) the cost of accessing and using the service, (3) details of ownership and access rights associated with a service, etc. The interface also allows the discovery, advertising, delegation and composition of services. Consider an automobile company that has the need to share its compute and data resources among its employees to enable them to share and access these resources as part of an international collaborative design process for developing a new car. We can identify the following participants involved in such a collaboration:

- *Grid services and toolkit developers*: these users care about what Grid tools and resources are available within the Grid infrastructure, to guide the software development process.
- *Grid administrators*: these users want to know about the status of resources in a production mode setting, which includes controlling, monitoring, and utilization related information.
- *Grid application users*: these users care about having a transparent high-level view of the Grid that exposes information related to a convenient problem solving environment rather than the maintenance and development of complex Grid applications.

Hence a Grid information framework needs to support not only a diverse set of information, but also needs to server a diverse set of Grid users.

A service stresses interoperability and may be dynamically discovered and used. According to [OGSA], the service abstraction may be used to specify access to computational resources, storage resources, and networks in a unified way. How the actual service is implemented is hidden from the

user through the service interface. Hence, a compute service may be implemented on a single or multi-processor machine – however, these details may not be directly exposed in the service contract. The granularity of a service can vary – and a service can be hosted on a single machine, or it may be distributed. The "TeraGrid" project [Tera] provides an example of the use of services for managing access to computational and data resources. In this project, a computational cluster of IA-64 machines may be viewed as a compute service, for instance, thereby hiding details of the underlying operating system and network. A developer would interact with such a system using the Open Grid Services Architecture (OGSA) toolkit, derived from Globus [Globus], and consisting of a collection of services and software libraries. Hence, it is assumed that a Computational Grids composed of a number of heterogeneous resources, which may be owned and managed by different administrators. Each of these resources may offer one or more services. A service can be:

- A single application with a well defined API (which also includes wrapped applications from legacy codes). In this case, the software application is offering some service (which may vary in granularity from a simulation kernel to a molecular dynamics application, for instance). In order to run such an application, it may be necessary to configure the operating environment (such as set up `PATH` variables which point to the location of particular libraries), and make available third party libraries.

- A single application used to access services on other resources – managed by a different systems administrator.

- A collection of coupled applications, with pre-defined interdependencies between elements of the collection. Each element provides a sub-service that must be combined in a particular order.

- A software library containing a number of sub-services, which are all related in some functional sense. For instance, a graphics or a numerics library etc.

- An interface for managing access to a resource (this may include access rights and security privileges, scheduling priorities, license checking software etc).

A distinguishing feature of a service is that it does not involve persistent software applications running on a particular server. A service, primarily, is executed only when a request for the service is received by a "service provider". The service provider publishes (advertises) the capability it can offer, but does not need to have a permanently running server to support the service. There is also a need for the environment which supports the service abstraction to satisfy all pre-conditions needed to execute a service. A service may also have "soft state" – implying that the results of a service may not exist forever. Soft state is particularly important in the context of dynamic systems such as Computational Grids where resources and user properties can vary over time. The soft state mechanism also allows a system to adapt its structure over time, depending on the behaviour of participants as they enter and leave a system at will. The soft state approach allows participation for a particular time period, subsequently requiring the participant to renew their membership. Participants not able to renew their membership are automatically removed from the system.

OGSA is a distributed interaction and computing architecture based around the Grid service. It aims to integrate Grid systems with Web services – and defines standard mechanisms to create, name, and discover Grid service instances. The core concept within this architecture is the notion of a ``Grid Service'', defined in the Web Services Description Language (WSDL), and containing some pre-defined attributes (such as types, operations and bindings), and provides a set of well-defined interfaces that require the developer to follow specific naming conventions. A new tag `gsdl` has been added to the WSDL document to enable description of Grid services. Once the interface of a Grid service has been defined, it must be made available for use by others. This generally involves publishing the service within one or more "registries". The standard interface for a Grid service includes multiple bindings and implementations (such as the Java and C# languages) – and development may be undertaken using a range of commercial (such as Microsoft's Visual Studio .NET) or public domain (IBM's WSTK) tools. A Grid service may be deployed on a number of different hosting environments, albeit all services require the existence of the Globus toolkit. Grid services implemented with OGSA are generally transient – and are created using a Factory service. Hence, there may be many instances of a particular Grid service – and each instance can maintain internal state. Service instances can

exchange state via messaging. Figure 1 indicates how a user may access Grid services via a Portal interface. Connectivity between the user and a number of organisations is achieved using XML/SOAP messages. Each organisation in this instance is offering a particular set of services – such as mathematical libraries, graphics routines, etc –encoded as WSDL services. A user can pick and combine a number of such services to implement an application. Each organisation may itself support a Local Grid – containing compute and data resources internal to an organisation. The ability to integrate a number of Local Grids is a significant advantage of using Web Service technologies to build Computational Grids.

When creating a new service, a user application issues a `create Grid service` request on a Factory interface – leading to the creation of a new instance. This newly created instance will now be automatically allocated some computing resources. An initial `lifetime' of the instance can also be specified prior to its creation, and allows the OGSA infrastructure to keep the service "alive" for the duration of this instance. This is achieved by sending it `keepalive` messages. The newly created instance is assigned a globally unique identifier called the Grid Service Handle (GSH) – and is used to distinguish this particular service instance. The implementation of a Grid Services involves [OGSADF]:

1. Write a WSDL `PortType` definition, using OGSA types (or defining new ones)
2. Write a WSDL Binding definition, identifying ways in which one could connect to the service, e.g. using SOAP/HTTP, TCP/IP etc
3. Write a WSDL Service definition based on the `PortTypes` supported by the Service, and identified in (1)
4. Implement a factory by extending the `FactorySkeleton` provided, to indicate how new instances of a Service are to be created
5. The factory must be configured with various options available – such as schemas supported, etc.
6. Implement the functionality of the Service, by extending the `ServiceSkeleton` class. If an existing code (legacy code) is to be used in some way, then the "delegation" mechanism should be used. When used in this mode, the factory returns a skeleton instance in step (4)
7. Implement code for the client that must interact with the Service

Extensibility Elements available within WSDL play a significant role in OGSA to enable users to customise their Grid Services. This is useful to enable a community or group of users to define special XML tags or queries that may be relevant within a particular application context. In OGSA, such extensibility elements include: *factory input parameters*, *query expressions*, and *service data elements (SDEs)*. Factory input parameters provide the user with a mechanism to customise the creation of a new Grid services. Query expressions enable a user to utilise a specialist query language or representation scheme, and SDEs enable the definition of specialist XML tags. A Math service which makes use of Numeric Service bindings is illustrated in code segment 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="MathServiceDefinition"
 targetNamespace="http://samples.ogsa.globus.org/math"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:grid-service-
bindings="http://ogsa.gridforum.org/service/grid_service_bindings"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

<import location="numeric_bindings.wsdl"
 namespace="http://samples.ogsa.globus.org/math/numeric_bindings"/>

<import location="../service/grid_service_bindings.wsdl"
namespace="http://ogsa.gridforum.org/service/grid_service_bindings"/>

<service name="MathMatrix">
  <documentation> … </documentation>

<port binding="numeric_bindings:NumericSOAPBinding"
 name="NumericPort">
```

```
 <soap:address location="http://localhost:8080/ogsa/services"/>
</port>

<port binding="grid-service-bindings:GridServiceSOAPBinding"
 name="GridServicePort">
 <soap:address location="http://localhost:8080/ogsa/services">
</port>
</service>
</definitions>
```

Code Fragment 1

Converting Existing Software

A number of tools have emerged recently which enable automatic creation of Web Service interfaces (WSDL and SOAP based) from existing Java based implementations, such as SOAPswitch [actional02] and `Java2WSDL` in the Web Service Toolkit [WSTK]. Such tools are essential to bootstrap interest in Web Services, by allowing existing applications to be made available as Web Services. Generally, there is little interest in re-writing an existing application as a Web or Grid Service – as significant investment has already been made in designing, implementing and maintaining existing software. This is particularly relevant for the science and engineering community, where a numeric solver or a visualisation routine, for instance, have been developed over many years, and conversion of these to a new technology often requires a considerable investment. Further, many designers of the initial system may not be available, and the exact reasons for particular design choices are not apparent (and often not well documented). The uptake of Grid Services within such a community is therefore also likely to be constrained by how existing codes may be efficiently re-used, and suitable tools are needed to prevent re-writes of existing codes. Wrapping of existing codes as services may be undertaken at different levels:

- *Wrapping executables*: This is the wrapping "as-is" approach, and is important when no source code is available. The execution environment of the original needs to be maintained, such as any particular dynamic link libraries that need to be referenced during execution.

- *Wrapping Source*: This is the "source-update" approach, whereby additional code needs to be written to interface with the existing application (generally the I/O routines). The original executable program needs to relinquish some control to the wrapper. This approach also requires some data type conversions to be undertaken between the original and the new types supported by the wrapper. Web Services make this process somewhat simpler, as a developer can specify a namespace to declare particular data types present in the original code, and subsequently reference these in the wrapper.

- *Source-split Wrapping*: This is the "unit-wrapping" approach, where the source code is divided into individual units, and can be wrapped separately. This approach is particular relevant when the code being wrapped is a software library or contains a collection of independent routines.

- *Application-supported Wrapping*: This is the "app-wrap" approach, whereby the wrapper is adding new functionality to the code, and not simply acting as a mediator between the Web Service and the existing application.

Regardless of which approach is adopted, the aim is to generate a WSDL document for every identified service, the `PortType` and the associated bindings. Additional properties about the service may be encoded in the Service Data Elements. An alternative approach, adopted in the Web Services Invocation Framework [WSIF], is to provide multi-protocol bindings -- enabling automatic invocation of the relevant binding based on the content carried in the request. Hence, bindings are providing for COM objects, for HTTP requests etc. Once a WSDL interface has been defined, it is important to place this into a registry service, so that other applications can automatically discover and make use of the service. In the context of Web Services, the discovery mechanism is generally provided by the Universal Description, Discovery and Integration (UDDI) registry – which may be organised along the

same ways as Domain Name Servers. Hence, services may register with one or more registry services (using the same identifier), and may be discovered by search distributed over one or more registries.

Current UDDI implementations are however limited in scope, in that they only allow search to be carried out on limited attributes of a services – namely on `service name` (which may be selected by the service provider), `key Reference` (which must be unique for a service), or based on a `categoryBag` (which lists all the business categories within which a service is listed [uddiwhitepaper]. Furthermore, public UDDI registries may contain a lot of listings for business that no longer exist, or sites that are no longer active [Mello02]. The interaction between UDDI registries is also an importance concern -- and there is no main consensus atthe present time of who should own the root UDDI registries. We extend UDDI as "UDDIe" to address some of these restrictions, and provide three main extensions: (1) support for "leasing" -- to enable services to register with UDDI for a limited time period (to overcome the problem of missing or inconsistent links), (2) support for search on other attributes of a service -- achieved by extending the `businessService` class in UDDI with `propertyBag`, and (3) extending the `find` method to enable queries to UDDI to be based on numeric and logical (AND/OR) ranges. The extensions allow UDDIe to co-exist with existing UDDI implementations -- and enable a query to be issued to both simultaneously. We describe these extensions and their use in subsequent sections.

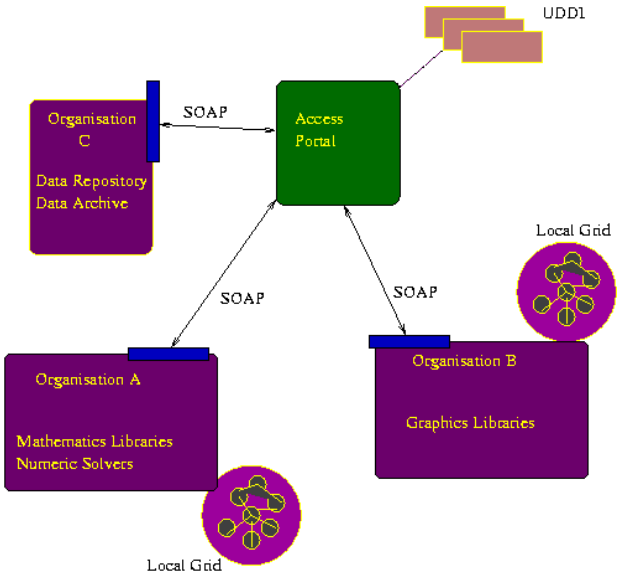

Figure 1: The general framework for enabling interaction between multiple Grid systems based on Web Services

The Role of Service Discovery and UDDI

Service discovery involves search through local and remote registries, to discover services which match a particular criteria defined by the user. In Grid computing, this was generally restricted to hardware resources being made available as services, and registered in the Metacomputing Directory Service (MDS) – now renamed to the Monitoring and Discovery Service. The MDS makes use of the Lightweight Directory Access Protocol (LDAP) to hierarchically structure the available resources within organisational domains. Each node within the LDAP structure contains properties associated with a particular resource. A user or application program could query the LDAP server to discover the type of hardware architecture (an Intel or a Sun machine, for instance), the operating system supported, and details about the job manager that is available at a given resource. A variety of query terms are available as `grid_info_search`, which could be configured to discover a particular property of a resource. Each organisational domain participating in the Grid would need to run a Grid Resource Information Service (GRIS), which would periodically update a Grid Index Information Service (GIIS) maintained at some central site. The GIIS/GRIS together constitutes the discovery service currently provided in OGSA/Globus. Hence, service discovery is supported via an "Information Service", that must relay information about Grid infrastructure to its users. All of these user communities require an

elementary set of functionality to be supported by Grid information service. This functionality includes:

- **lookup and retrieval** of information that can be pinpointed to a particular resource or object,
- **query and search** of information to retrieve a collection of related resources or objects,
- **creation** of new information upon request that is otherwise not stored or cached in the Grid,
- **event forwarding** to relay the information based on dynamic events within the Grid infrastructure,
- **aggregation** for topical retrieval and organization of information,
- **filtering** of information to reduce the amount or useful information communicated between services, to increase performance and reduce the amount of information to be stored,
- **storage, backup, and caching** of the information to make them available for easier retrieval or enhance the performance,
- **security, protection, and encryption** in order to enable important security related activities such as access control, authentication.

To support these functionalities it is convenient to provide a number of services that may be used by the Grid users. Figure 2 lists a number of corresponding Services implementing such functionality. It is important to notice that there may be many different services based on the diverse user communities and needs in each category. In most cases it is common to provide a number of high level services that combine certain aspects of the base functionality. A common example is the implementation and use of a directory service to support the functionalities of lookup and retrieval of information that is classified in a hierarchy. Such a service need to provide support for both white and yellow pages.

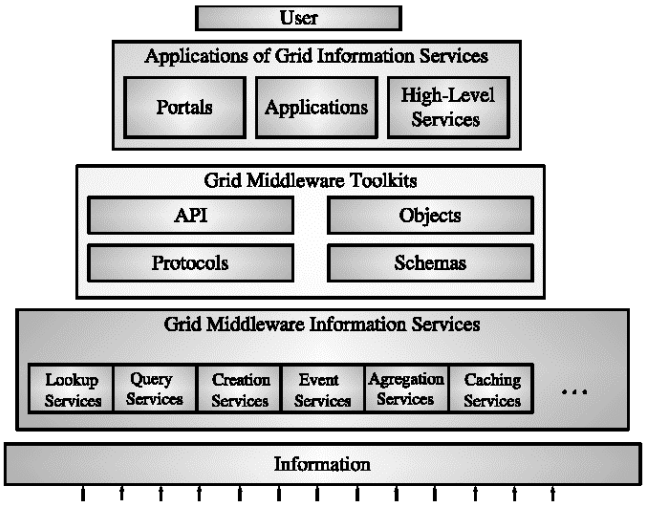

Figure 2: A Grid Information Service builds the bridge between information about resources on the Grid and the user community

Operational Requirements

To chose an appropriate implementation strategy for a Grid information service a number of important requirements have to be considered. Some of these requirements can be expressed in quantitative terms such as scalability and performance; others are more subjective such as expressiveness, deployability, and performance. Typically use patterns for Grids operate on a large scale (e.g., hundreds and thousands of resources) and have demanding performance requirements. Hence, an information infrastructure must permit rapid access to frequently used, and dynamically changing information. We list such operational requirements that include capability, security, and scalability concerns and quote from [Laszewski97]:

- **Scalability and cost:** The infrastructure must scale to large numbers of components and permit concurrent access by many entities. At the same time, its organization must permit easy discovery of information. The human and resource costs (CPU cycles, disk space, network

bandwidth) of creating and maintaining information must also be low, both at individual sites and in total.

- **Uniformity:** The goal should be to simplify the development of tools and applications that use data to guide configuration decisions. A uniform data model is required, as well as an application programming interface (API) for common operations on the data represented via that model. One aspect of this uniformity is a standard representation for data about common resources, such as processors and networks.

- **Expressiveness:** The data model should be rich enough to represent relevant structures within distributed computing systems. A particular challenge is representing characteristics that span organizations, for example network bandwidth between sites.

- **Extensibility:** Any data model that is defined will, by necessity, be incomplete. Hence, the ability to incorporate additional information is important. For example, an application can use this facility to record specific information about its behaviour (observed bandwidth, memory requirements) for use in subsequent runs.

- **Diversity. Multiple information sources:** The required information may be generated by many different sources. Consequently, an information infrastructure must integrate information from multiple sources.

- **Dynamicity:** Some of the data required by applications is highly dynamic, for example, network availability or load. An information infrastructure must be able to make this data available in a timely fashion.

- **Flexibility:** It is necessary to both read and update data contained within the information infrastructure. Some form of search capability is also required, to assist in locating stored data.

- **Security:** It is important to control who is allowed to update configuration data. Some sites will also want to control access.

- **Deployability:** An information infrastructure is useful only if is broadly deployed. In the current case, we require techniques that can be installed and maintained easily at many sites.

- **Decentralized maintenance:** It must be possible to delegate the task of creating and maintaining information about resources to the sites at which resources are located. This delegation is important for both scalability and security reasons.

Support in UDDI

An alternative approach to discovery is provided in the UDDI registry, where UDDI provides a standard method for publishing and discovering information about Web services. The UDDI Project is an industry initiative that attempts to create a platform-independent, open framework for describing and discovering services offered by one or more businesses. UDDI is therefore more generic in nature, compared to GRIS/GIIS, and is also relevant in the context of storing details of Grid services. The particular issues that UDDI attempts to address include:

- Making it possible for organisations to quickly discover the right business from millions of others currently on-line

- Defining how to enable interaction with the business to be conducted, once it has been discovered

These issues are intended to increase the ability of a business to discover new customers, and expand its market reach. It is useful to note that service discovery can happen at application design time or at runtime. For instance, a user may search a service registry via a browser interface, and manually evaluate the results. Alternatively, the application may itself issue a `find` request to the registry, and

utilise analysis logic to evaluate the results. Current efforts in UDDI development, such as work being undertaken by the OASIS technical committee [OASIS-UDDI] aims to address issues of security and access privileges, ranging from mechanisms to encrypt SOAP messages, to support for a security policy that mediates interaction between Web Services. There is also work underway to add additional features to UDDI, such as providing support for new match mechanisms – such as `caseInsensitiveMatch`, `diacriticSensitiveMatch` etc, and the capability to arrange services in UDDI based on various sort criteria.

It is important to note that UDDI is a registry and *not* a data repository – and therefore should not be used to store data values related to a particular service, but rather the metadata about the structure of the data. In the context of Grid computing, a registry may contain the WSDL documents about particular services, the current location of services, a fault or audit log identifying error conditions that occurred on particular services, or access rights about one or more services, for instance. It should not be used to record input or output data generated from a service. The primary reason for this is to keep registry service lightweight, and enable ease of administration and replication of content within them. However, a registry may have references to real data stored in a structured database or a file system. There may be two types of UDDI registries: those that are private to a given organisation, and those that are to be made available for public use. Private registries may contain services that are only useful in a local context, and may be hidden behind company firewalls. An organisation may also utilise private registries to record temporary information about processes active at a given time. Public registries contain services that may be discoverable by other users, and must be advertised to external users. UDDI registries may also be arranged in a hierarchy, whereby public registries available at a given institution may be aggregated to form a single regional UDDI registry. Such a registry may not replicate the content of each UDDI node, but provide references to content providers.

It is useful to note that UDDI is primarily meant as a registry for business services – and data structures within UDDI reflect this aspect of it. Information about a business that can be stored in UDDI may be classified as:

- *White pages*: These contain basic contact information and identifiers about a company, including business name, address, contact information and unique identifiers such as its Dun-and-Bradstreet (DUNS) numbers or tax IDs. This information allows others to discover Web Service based upon business identification. In the context of Grid Computing, white pages could provide the retrieval of an IP address or the amount of memory available on a particular resource.
- *Yellow pages***:** These contain information that describes a Web Service using different business categories (taxonomies). This information allows others to discover Web Wervice based upon its categorization (such as flower sellers, or car sellers etc).
- *Green pages*: These contain technical information about Web Services that are exposed by a business, including references to specifications of interfaces, as well as support for pointers to various file and URL-based discovery mechanisms.

The data structures in UDDI are primarily meant to support business services, and do not provide an effective representation of needs of the science and engineering community. Generally, these data structures allow encoding of properties related to White, Yellow and Green pages defined above. These are therefore not adequate on their own for representing services in scientific disciplines, as no similar categorisation/taxonomy of scientific areas is currently in use. There is also no unique naming scheme (such as DUNS/tax IDs) in for representing providers of science or engineering services. There are on-going efforts at the Global Grid Forum to alleviate some of these concerns.

UDDI and OGSA

The UDDI—OGSA bridge is illustrated in figure 3, and based on [Colgrave03]. To bootstrap the system, a UDDI registry may contain references to a HandleResolver, a Factory or a ServiceGroup, for instance. The information maintained in UDDI could therefore be used to create a new instance of a service – this is based on the assumption that WSDL descriptions of services are registered with UDDI, and SOAP/HTTP binding is required. Interaction between a Globus environment and UDDI would be mediated via the OGSA—UDDI bridge, enabling the description of Grid Services with a particular

lifetime (achieved by defining a ServiceGroup which conforms to these properties). A service may join one or more of such Groups – via the ServiceGroupRegistration interface. A UDDI-aware HandleResolver would be used to identify service descriptions which have been registered within UDDI. There is also no security support at present for authenticating users with the UDDI registry, although this is a significant requirement for business services to interact. The general consensus is that public UDDI registries should contain information that is universally accessibly. Use of X509 based digital certificates via the Grid Security Interface (GSI) is a useful way to provide secure access to the UDDI registry – whereby the UDDI—OGSA bridge verifies user credentials before accepting a query.

The UDDI—OGSA bridge is therefore meant to connect two environments without necessitating a major change in either. It is a useful compromise which can still enable the use of existing UDDI implementations, by relating key structures in Grid environments (such as Globus) with the UDDI registry.

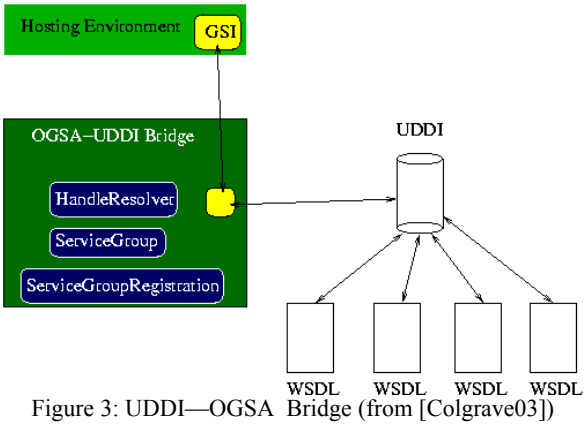

Figure 3: UDDI—OGSA Bridge (from [Colgrave03])

UDDIe: UDDI Extensions and Implementation

The approach adopted in UDDIe is different from the approach taken in the UDDI—OGSA bridge, in that additional functionality has been added to UDDI to enable its use with Grid Services. This functionality also makes UDDIe a generic registry service for Web Services, that may also find use in other applications. Information within UDDIe is also stored as a WSDL document, but can make more effective use of the Extensibility Elements supported in OGSA. A set of user defined attributes can now be recorded as part of a `propertyBag` in UDDIe, thereby allowing a user to discover services based on user defined attributes.

UDDIe uses an XML schema which extends that used in standard UDDI [uddiwhitepaper], by using the same specification and standards for the registry data structures and Application Programming Interface (API) specification for inquiring and publishing services. Extensions in UDDIe are based on four types of information: business information; service information, binding information; and information about specifications for services. A service may be discovered by sending requests based on service information. The extensions provided in UDDIe consist of the following:

- *Service Leasing*: Service providers may want to make their service available for limited time periods (for security reasons, for instance) – or the service may change often. There is a lack of support within existing UDDI implementations for services which change often, or unreliable hosting environments which may become un-accessible. To overcome this, UDDIe supports "Finite" and "Infinite" leases – where a finite lease can be immediate, or specified as a future lease. When using finite leases, service providers must define the exact period for which the service should be made available for discovery in the registry. The lease period is restricted by the maximum allowable lease period defined by the UDDIe administrator. For example, if a service provider is interested in publishing a service in UDDIe for two hours, but the maximum granted lease is one hour, publication of the service will be rejected by the registry. A "futurelease" allows a service provider to make the lease period start at a future time – the service will only be discoverable once this lease has been activated. The futurelease concept is supported to enable a systems administrator to directly encode dependencies in

service invocation. Alternatively, service providers may want to publish their services for an infinite period of time. Such leases are allowed in UDDIe, but only if the ratio of finite/infinite lease services is within a threshold (a parameter set by the UDDIe administrator).

Effectively, an Infinite Lease is used to define persistent services, which must be maintained in UDDIe for long time periods. These may include system critical services which need to be held in the registry for a Grid system to work effectively – examples include the Monitoring and Discovery Service (MDS), or various job managers that may be available. Conversely, every user defined service must be granted a Finite lease, and would require a user to renew this periodically.

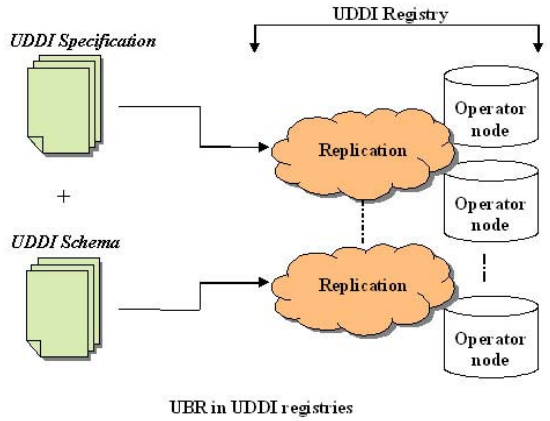

Figure 4: Replication UDDI nodes to form the Universal Business Registry

- *Replication*: The UDDI Business Registry (UBR), illustrated in figure 4, is conceptually a single system built from a group of UDDI nodes that have their data synchronized through replication. The UBR is intended to be a public registry that contains a number of different UDDI services. A series of operator UDDI nodes each host a copy of the content, thereby replicating content among one another. Content may be added to the UBR at a single node, and that operator node becomes the content master. Any subsequent updates or deletes of the data must occur at the operator node where the data was inserted. UDDIe can be used as a private operator node that is not part of the UBR. Private nodes do not have data synchronized with the UBR, so the information contained within is distinct. The availability of private nodes is significant if an organisation considers sharing their service content a security problem. This is useful in instances where a company does not want to expose certain service offerings and business processes to others – for instance, suppliers set up to handle large contracts may not be able to handle individual customers. It is not clear who should own and manage the "top most" UDDI nodes in the context of Grid computing. Currently, major UDDI vendors such as IBM and HP are expected to own the top most UDDI registry for business services, however, it is unclear how such a structure should be managed for Grid services.

In UDDIe a `businessService` [uddiwhitepaper] structure represents a logical service -- and is the logical child of a `businessEntity` -- the provider of the service. Service properties are contained in the `propertyBag` entities -- such as the Quality of Service (QoS) that a service can provide, or the methods available within a service that can be called by other services(an important feature missing in current UDDI implementations). Figure 1 illustrates the attributes associated with a `property` -- and consists of a `propertyName`, `propertyType` and `propertyvalue`. Some of these are user defined attributes -- such as `propertyType` -- and can be `number`, `string`, `method` etc. Range based checks, for instance, are only allowed if the `propertyType` is a number.

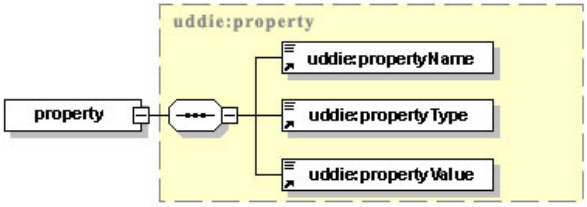

Figure 5: Property Attributes

An example of using `PropertyBag` is illustrated in code segment 2. The data types that can be supported in the `PropertyType` may be extended using `xsd` (XML Schema Description) tags, and a user defined namespace. The `propertyFindQualifier` is part of the query, whereas the other aspects of the `propertyBag` are also used to define particular properties.

```
<propertyBag>
 <property>
   <propertyFindQualifier>GREATER_THAN</propertyFindQualifier>
   <propertyName>CPU</propertyName>
   <propertyType>number</propertyType>
  <propertyValue>800</propertyValue>
 </property> …
</propertyBag>
```

Code Segment 2

The API for interacting with the registry system extends three classes within existing UDDI implementations. The extensions provided in the API include:

- `saveService`: This set of APIs is mainly used for publishing service details. This has been extended from the original UDDI system to introduce dynamic metadata for services. Such metadata could be used to represents attributes such as cost of access, performance characteristics, or usage index associated with a service, along with information related to how a service is to be accessed, and what parameters the service will return.

- `findService`: This set of APIs is mainly used for inquiry purposes. In particular we extend this set of APIs from the original UDDI to include queries based on various information associated with services, such as `Service Property` and `Service leasing`.

- `getServiceDetails`: This set of APIs is mainly used for requesting more detailed information about services, such as `BusinessKey` information etc. We extend this set of APIs to include `Service Property` information as well

- `renewLease`: This set of APIs is used by both the operator/UDDI administrator to control leasing information, and by the service provider (SP) to renew and set leasing information. The leasing concept works as follows; lease information must be associated with every service, either for a limited duration or for an infinite time period. The maximum number of infinite leased services is controlled by the operator to efficiently maintain the registry. In the case of limited duration, the SP provides a start-from date and an expiration date for the lease period. The operator has control on setting up the default leasing period. Moreover, if a lease expires the SP could always renew the expired lease, provided that the request falls within the allowed number of times to renew a particular lease, which is controlled by the operator/UDDI administrator. When the lease period expires, the service becomes invalid and clients cannot make use of the expired services. It is important to continually renew a lease or request an infinite lease -- and an event manager is used to alert all connected users if the lease of the service is about to expire.

- `startLeaseManager`: This newly defined set of APIs is used to monitor lease constraints, by generating processes to monitor and update the lease period. A service with an expired lease is removed from the registry. The operator has control over how often to run these

processes. The lease manager is started when the UDDIe registry is first initialised, and must run as a background process. The lease manager must be configured by the UDDIe administrator to support a particular ratio of services with infinite and finite leases.

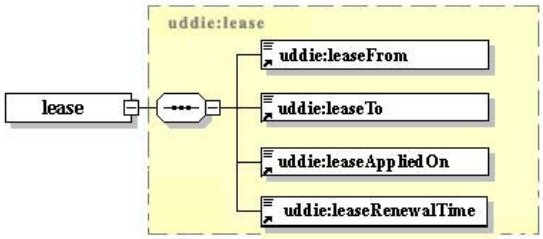

Figure 6: The "Lease" element

In addition to these sets of APIs, we introduce support for a *Qualifier-based* search – to find services based on some property along with a qualifier value, such as `LESS_THAN, GREATER_OR_EQUAL, EQUAL_TO, NOT_EQUAL_TO` and `GREATER_THAN`. This set of qualifiers may be extended by the user. Support for logical operations is introduced to enable querying for properties with logical `AND/OR` operators. The search for services which match a particular request is based on the following set of operations:

element result set: A set containing all service keys which match the value of the element
total result set: A set which contains all the element sets

FOR EACH element in the find_service message DO
  Fetch the services which matches the element value
  Add the services' key into the *element result set*
  Add the element result set into the *total result set*
END For Loop

IF Logical OR is required THEN
  final result set = Union of all *element result set* in the *total result set*

IF Logical AND is required THEN
  final result set = Intersection of all *element result set* in the *total result set*

UDDIe also provides support for error checking, and includes support for the following types of errors:

- *Range checks*: used to ensure that the range specified for a particular property does not exceed their limits
- *Existence*: this error check is used to ensure that a particular property actually exists, and is returned to the user via a DispositionReport

```
<dispositionReport xmlns="urn:uddi-org:api_v3">
  <result errorno="***">
    <errInfo errCode="***"/></result></dispositionReport>
```

- *Duplicates*: this check is used to ensure that a particular service does not have two definitions for the same `PropertyType`, or that a particular property is not replicated. This is achieved via the `Unique` tag.

- *Lease checking*: a number of checks are provided to support leasing (soft state), such as InvalidLeaseDateException (a data quality check to ensure that the lease data conforms to the required format), a RenewalTimeExceededException (a rule-based check to ensure that a particular lease has not been renewed more than a given number of times), and InfiniteLeaseOutofBoundException (a rule-based check to ensure that the number of Infinite to Finite leases does not exceed the threshold set by the systems administrator).

We believe these extensions to the UDDI registry and query mechanisms provides flexibility un undertaking search, thereby making UDDI a more powerful search engine. The ability for UDDIe to co-exist with standard UDDI version is also an important aspect of this work -- as we do not break compatibility with existing UDDI deployments. Code Segment 3 illustrates a Math Service with service properties added, and to be used within UDDIe. The description of the Math Service illustrates the use of SOAP bindings for performing a particular request, and the use of Quality of Service attributes of a service (as its properties). The `QoS` tag indicates that this service requires a CPU rating of 100, and a disk storage rating of 150 to execute successfully. Hence, the WSDL interface for the service indicates operations supported by the service, mechanisms for connecting to the service, and performance characteristics that need to be adhered to if the service is to run successfully.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:impl="http://MathService"
      xmlns:intf="http://MathService-Interface"
     xmlns:tns1="http://DefaultNamespace"
     xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://MathService-Interface">


 <wsdl:message name="getSumResponse">
    <wsdl:part name="return" type="SOAP-ENC:string"/>
 </wsdl:message>
 <wsdl:message name="getSumRequest">
    <wsdl:part name="symbol" type="SOAP-ENC:string"/>
 </wsdl:message>

 <wsdl:portType name="MathService">
   <wsdl:operation name="getSum" parameterOrder="symbol">
     <wsdl:input message="intf:getSumRequest"/>
     <wsdl:output message="intf:getSumResponse"/>
   </wsdl:operation>
 </wsdl:portType>


wsdl:binding name="mathSoapBinding" type="intf:MathService">
   <wsdlsoap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="getSum">
     <wsdlsoap:operation soapAction="" style="rpc"/>
     <wsdl:input>
       <wsdlsoap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://MathService-Interface" use="encoded"/>
     </wsdl:input>
     <wsdl:output>
        <wsdlsoap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://MathService-Interface" use="encoded"/>
     </wsdl:output>
   </wsdl:operation>
 </wsdl:binding>

 <wsdl:QoS>
    <cpu_count>100</cpu_count>
    <disk_storage>150</disk_storage>
 </wsdl:QoS>
```

```
</wsdl:definitions>
```

Code Segment 3

Download

UDDIe may be downloaded as an open-source software, protected via the GNU Public License, from http://www.cs.cf.ac.uk/User/A.Shaikhali/uddie/ . It extends the public domain implementation of UDDI available at `uddi.org`, and consequently requires a number of additional programs to be available to run – such as `uddi4j`, a Web server that supports Java Applets (such as Tomcat), a relational database management system (such as Oracle, MSSQL, etc), JDBC drivers for the particular database selected, and a Java virtual machine (shipped as part of the JDK). A pre-compiled version of UDDIe is also available at the above site, and has been tested with Tomcat server 4.1, Oracle 9i, the Oracle Thin driver, and JDK 4.0. A Java client for UDDIe is also provided at this site, and may be used as a stand-alone application, or embedded into an existing Java program. An Applet implementation of the client is provided via the Web site to allow external users to experiment with UDDIe on-line.

Uses

UDDIe can be utilised in any applications where providers register their service for a limited duration, or where users search for services based on range-based/logical criteria. This may include business providers who want to register their services with attributes which are numeric -- such as service cost, or service quantity, or when specifying performance or Quality attributes associated with a service. Service registration for limited durations is particularly useful when the environment within which such a service is used is dynamic and may change often. Here, an infinite lease for a service may not be suitable – or a service provider may predict a future time when their service may be usefully deployable and opt for a future lease. To use UDDIe, a user must define service properties, as illustrated in code fragment 1. A UDDIe proxy is established to enable a user to publish and interrogate the registry. A service called `Math` is then registered -- and properties associated with the service specified. These properties are then registered into a `propertyBag`.

```
UDDIeProxy proxy = new UDDIeProxy();
proxy.setInquiryURL("http://localhost:8080/uddie/inquiry");
proxy.setPublishURL("http://localhost:8080/uddie/publish");

//Get Authorization by sending a username and password
// for the owner of the business

AuthToken token = proxy.get_authToken("ali", "ali");

//Define service name and add them to a vector
//The maximum allowed names is 5

Name name = new Name("Maths");
Vector names = new Vector();
names.add(name);

// Define Service propertiesProperty

property= new Property("CPU", "number", "50");
property.setPropertyFindQualifer(PropertyFindQualifiers.GREATER_THAN;
Property property2 =   new Property("RAM", "number" , "30");
property2.setPropertyFindQualifer(PropertyFindQualifiers.EQUAL_TO);
Vector properties = new Vector();properties.add(property);
properties.add(property2);
PropertyBag bag = new PropertyBag();
bag.setPropertyVector(properties);
```

Code Fragment 4

We then define a `FindQualifier` that may be used to search for the registered service – and issue a query:

```
// Define Find Qualifer for property exact match (Logical AND)
FindQualifier findQualifier = newFindQualifier("exactPropertyMatch");
FindQualifier findQualifier2 = newFindQualifier("exactNameMatch");
FindQualifiers qualifiers = new FindQualifiers();
Vector qualifiersVector = new Vector();
qualifiersVector.add(findQualifier);
qualifiersVector.add(findQualifier2);
qualifiers.setFindQualifierVector(qualifiersVector);

// Send the query
ServiceList list = proxy.find_eService(null, names, null, bag, null,
qualifiers , 5);
```

Code Fragment 5

Once a query has been issued, details of the services which match our requests are printed using a`for` loop, and returned in the `ServicesInfos` object, as illustrated in code segment 6. The first loop (using index `i`) returns a list of services, and the second (using index `j`) returns properties associated with each service.

```
ServiceInfos infos =   list.getServiceInfos();
Vector services =   infos.getServiceInfoVector();

for ( int i=0;i<services.size();i++){
ServiceInfo service =   (ServiceInfo)services.get(i);
System.out.println("Service  returned Name: " +
service.getName().getText());System.out.println("Service  returned
Key: " + service.getServiceKey());
eServiceDetail serviceDetail =proxy.get_eServiceDetail(
service.getServiceKey());
Vector serviceVector =serviceDetail.getBusinessServiceVector();
BusinessService returnedService
=(BusinessService)serviceVector.firstElement();
Lease lease = returnedService.getLease();System.out.println("lease
expiration date: " +lease.getLeaseExpirationDate());
PropertyBag bag2 =  returnedService.getPropertyBag();
Vector propertiesFound =  bag2.getPropertyVector();

for (intj=0;j<propertiesFound.size();j++){ Property propertyFound =
(Property)propertiesFound.get(j);
System.out.println("Property Name: "
+propertyFound.getPropertyName());
System.out.println("Property Value: "
+propertyFound.getPropertyValue()); }
```

Code Fragment 6

Figure 7 illustrates how a client can use the UDDIe registry. A request is received from an external client using HTTP, and parsed by the Servlet. A client must distinguish whether the request is a UDDI request or a UDDIe request when publishing information about a service. However, when searching for a service, both UDDI and UDDIe requests are treated identically – as UDDIe extends the `find_service` in UDDI with additional attributes. If these extensions are not implemented (as happens in existing UDDI registries) the default version of `find_service` will be invoked – however, if service properties are also specified in the call, then the updated version in UDDIe will be used. Java classes to implement UDDIe extensions are stored as a `jar` file, and can be invoked by the parser when the extensions are encountered. Publishing service information (as illustrated in code fragments 5 and 6) will require the user to define properties associated with a service, and add these to the `propertyBag`.

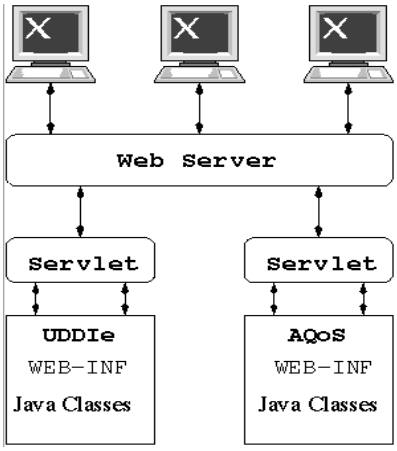

Figure 7: When using UDDIe, a client sends a request as an XML message to the Servlet, which parses the request, and forwards it to the registry

Quality of Service Management

We are currently making use of UDDIe to support Quality of Service (QoSM) management in the context of Grid Computing. A service in this case represents either a scientific code or a mathematical library, and each service has a description provided in a WSDL document. A service also has other attributes, such as bandwidth, CPU and memory requirements encoded in the service interface. Some of these parameters, such as bandwidth, packet jitter and loss are measured by a "Bandwidth Broker" (a network monitoring program that is used to derive these low level metrics [bb]). These attributes allow a service user to choose between services based on their QoS attributes, rather than just their functional properties. The system is implemented in the context of our G-QoSM framework [gqosm], whereby clients/applications send their requests for services with QoS properties to our QoS broker (code fragment 7 gives an example of the WSDL document used). The broker processes the request and submits the service request portion to UDDIe. The UDDIe registry replies with a list of services that support this particular query. Finally, the broker applies a selection algorithms to select the most appropriate service with respect to the client/application request and sends the result to the client/application. The selection algorithm that the broker uses is based on the Weighted Average (WA) concept. In this algorithm, we introduce the notion of a QoS importance level, whereby the client/application is requested to associate a level of importance, such as High, Medium or Low, with every QoS attribute. Based on this QoS importance level and the value of a QoS attribute, the algorithm computes the WA for every returned service and select the service with the highest WA.

```
<?xml version="1.0" encoding="UTF-8"?> <wsdl:definitions
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"  ... ]
targetNamespace="http://MyService-Interface">
<wsdl:messagename="printNameResponse">
</wsdl:message>  ...
<wsdl:QoS>
 <service_cost> 5 </service_cost>
 <network_bandwidth> 256K </network_bandwith>
<memory> 48MB </memory>
    ....</wsdl:QoS>
</wsdl:definitions>
```

Code Fragment 7

Conclusion

The role of the "service" abstraction in Grid computing is discussed, followed by various mechanisms about how such an information service may be used to publish and discover such services. The need to base such an information service on Web Services technology – such as UDDI – and how such technology may be connected to existing Grid middlware, such as Globus, is then outlined. The implementation of a registry for Web services, which extends UDDI, is specified. It is particularly useful when storing service properties with range based attributes – and also enables search for services based on these properties. UDDIe and WSDL provide an important mechanism for specifying and

deploying Web Services – especially when extending a WSDL document with additional attributes such as service quality and performance data. Using UDDIe, it is also possible to publish information about method calls (when wrapping object based implementations) available within a service (as its properties) and to subsequently search based on these method signatures. The extension via `propertybag` therefore allows a better way to make existing object based systems available as services – especially when used with tools such as SOAPswitch [actional02] and `Java2WSDL` [wstk].

Acknowledgement

We would like thank Rashid J. Al-Ali for his work on implementing Quality of Service properties using UDDIe, and members of the Welsh e-Science Centre.

The work of Dr Laszewski was supported by the Mathematical, Information, and Computational Science Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under ContractW-31-109-Eng-38. DARPA, DOE, and NSF support Globus Project research and development. The Java CoG Kit Project is supported by DOE SciDAC and NSF Alliance.

References

[actional02] Actional -- Web Services Management Solutions. "SOAPswitch". See Web site at http://www.actional.com/products/soapswitch/index.asp. Last visited: October 2002

[Colgrave03] John Colgrave, "UDDI and OGSI", presented at *Grid Information Services Workshop*, Edinburgh, UK, April 25, 2003

[gqosm] R. Al-Ali, O. Rana, D. Walker, S. Jha, and S. Sohail. "G-QoSM: Grid service discovery using QoS properties". *Computing and Informatics Journal, Special Issue on Grid Computing*, 21(5), 2002.

[Laszewski03] G. von Laszewski, G. Pieper, and P. Wagstrom, "Gestalt of the Grid," in *Performance Evaluation and Characterization of Parallel and Distributed Computing Tools*, ser. Wiley Book Series on Parallel and Distributed Computing, to be published 2003. [Online]. Available: http://www.mcs.anl.gov/~laszewsk/bib/papers/vonLaszewski--gestalt.pdf

[Laszewski97] G. von Laszewski, S. Fitzgerald, I. Foster, C. Kesselman, W. Smith, and S. Tuecke, "A Directory Service for Configuring High-Performance Distributed Computations," in *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, 5-8 Aug. 1997, pp. 365–375. [Online]. Available: http://www.mcs.anl.gov/~laszewsk/papers/fitzgerald--hpdc97.pdf

[OGSADF] Thomas Sandholm and Jarek Gawor, "Grid Services Development Framework Design", part of the *OGSA pre-release*. Available as: http://www.globus.org/ogsa/releases/TechPreview/ogsadf.pdf

[Stevens02] Service-Oriented Architecture introduction, part 1. See Web site at: http://softwaredev.earthweb.com/microsoft/article/0,,10720_1010451_1,00.html.

[wstk] IBM -- Alphaworks, "Web Services Toolkit". See Web site at: http://www.alphaworks.ibm.com/tech/webservicestoolkit/. Last visited October 2002.

[wsif] IBM -- Alphaworks, "Web Services Invocation Framework". See Web site at: http://www.alphaworks.ibm.com/tech/wsif/ . Last visited October 2002.

[metadata] E. Madja, A. Hafid, R. Dssouli, G.v. Bochmann, and J. Gecsei. "Meta-data modelling for Quality of Service Management in the World Wide Web". *Proc. of Int. Conf. on Multimedia Modeling*, 1998.

[Mello02] Adrian Mello, "Breathing new life into UDDI", Tech Update, ZDNET.com, June 26, 2002

[bb] B. Teitelbaum, S. Hares, L. Dunn, R. Neilson, R. Vishy Narayan,and F. Reichmeyer. “Internet2 QBone: Building a testbed for differentiated services”, *IEEE Network*, 13(5):8--16, September/October 1999

[uddiwhitepaper] UDDI Technical White Paper. Available as: http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf

[OASIS-UDDI] OASIS UDDI Specification Technical Committee. See documents at: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec