

# Distributed Data Repository Supporting Ad-Hoc Collaborations

Tomasz Haupt, Anand Kalyanasundaram, Igor Zhuk  
Center for Advanced Vehicular Systems, Mississippi State University

**Abstract:** *This paper presents the design and implementation of a distributed data repository for Grid environments that supports secure sharing of possibly confidential data by members of ad-hoc created groups. The system is composed of three independent services - metadata service, replica locator service and storage service. The group-based access control is achieved through augmentation of the user credentials required by the back-end storage service using a chain of authorization assertions added by anticipating services. The metadata service introduces a very flexible system for storing and querying the metadata. The metadata schemas define a variable length vector of attributes that can be easily modified by introducing new versions without any intervention of the system administrator. In addition, the user may organize the metadata into a tree hierarchy enabling the creation of custom views of data collections that are independent of the actual organization of the storage.*

**Keywords:** *Data management, Data repository, Grid computing, Data collaboration*

## 1 Introduction

The advent of the Internet, Web Services, and Grid Computing opened new opportunities for research in virtually every domain by providing the unprecedented access to distributed resources such as data, computing facilities and instruments. Numerous ongoing projects develop the necessary infrastructure to enable secure access to remote data, share the data between researchers within virtual organizations, perform data mining, and develop a mesh of information linked up in such a way as to be easily processable by machines on a global scale – the Semantic Grid.

This paper describes a distributed data repository system developed for Grid environments. This particular system focuses on providing support for ad-hoc collaborations that require sharing restricted-access data between members of dynamically created groups. The system is designed to support sharing thousands of data sets of a small to moderate size (Gigabytes) per group. The number of the data sets mandates support for metadata describing these data sets thus allowing for queries. The manner in which the data is described, that is, the metadata schema may vary from group to group, and in addition, it may change in time adapting to the evolving user needs. Therefore it is necessary to introduce a very flexible metadata system without compromising the performance of queries. The data may be stored at different locations that support different mechanisms or interfaces for data access. Finally, strong authentication and group-based authorization mechanisms are required.

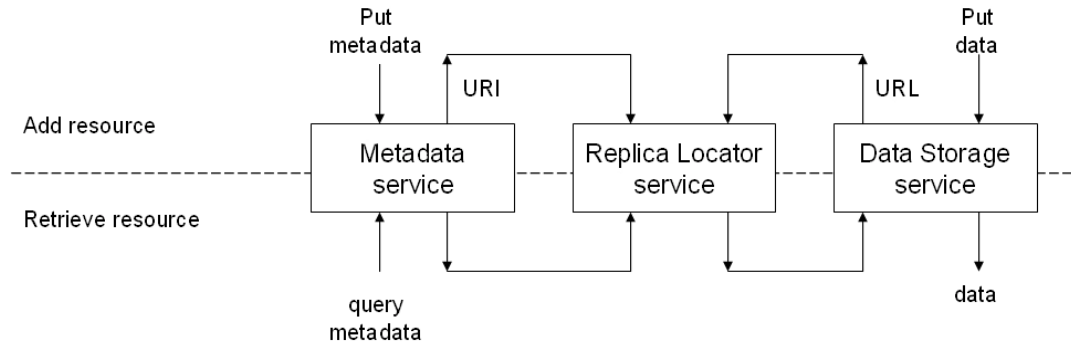
The repository is implemented as an aggregate of four independent services: metadata service, replica locator, data storage and authorization service, all hidden behind a common façade which exposes the API of the repository to remote, Web-based clients. The high-level model of the repository is presented in Section 2. Section 3 describes the metadata service, and Section 4 reveals details of the replica locator and data storage services. The security model of the repository and its implementation is explained in Section 5. Section 6 describes how the repository can be accessed by remote clients through a common façade. Finally, Section 7 summarizes the features of the system repository.

## 2 Architecture of the Repository

The model that drives the design and implementation of the repository is data centric: each data item is treated as an independent resource uniquely identified by its uniform resource identifier (URI). The URI is created automatically by the system: concatenation of the user-assigned logical name of the resource and a namespace. Consequently, it is required that the logical name is unique within its namespace. Each resource is associated with a metadata record – a variable length vector of attribute name-value pairs. The attributes capture the properties of the resource. Each metadata record contains both generic properties (such as ownership of the data, time of creation and data format) and application specific, user-defined attributes that facilitate queries. Notably the metadata records do not capture the physical location of the resources.

The storage mechanisms are hidden from the user, and the actual locations of the resources, usually expressed in terms of uniform resource locators (URL), are left to the discretion of the storage implementation. This is different from other

systems, such as SRB [1] where the user may specify the type of storage to be used. The support for explicit specification of the storage device is important when dealing with huge data sets, and certainly SRB has been carefully designed to handle such data sets correctly. Because of the separation of the metadata (a collection of logical resources identified by URI) and storage (a collection of physical resources identified by URL), a mapping service, referred to as Replica Locator service is needed. The repository thus comprises three independent services, metadata service, replica locator service and data storage service, as shown in Figure 1.



**Figure 1:** High-level architecture of the repository. To add a data resource to the repository the user puts a metadata record to the metadata service which returns a URI of the resource and puts the file to the data storage which returns a URL. The URI-URL pair is then put to the replica locator. To retrieve a data resource the user queries the metadata repository to obtain the URI of the resource (or uses a cached URI). The URI is resolved to the URL by the replica locator. The URL is used to retrieve the corresponding data resource.

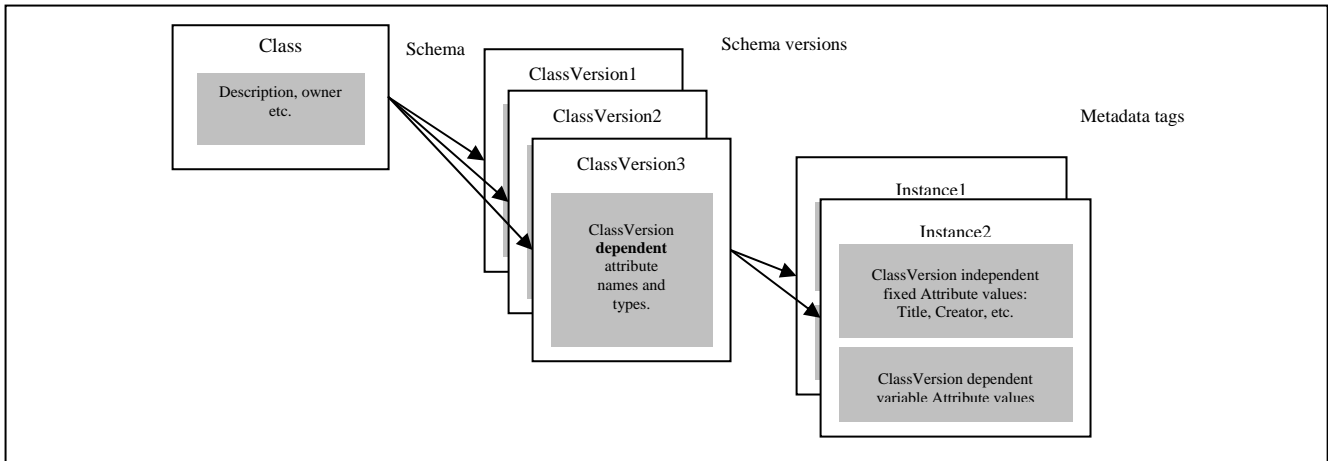
Creating the repository as an aggregate of three independent services has clear advantages. The separation of concerns simplifies the implementation, and in particular allows mixing and matching different implementations. For example, depending on the characteristics of the data sets to be maintained by the repository, the Storage service can be implemented as a simple file system accessible through a secure file transfer protocol such as gridFTP, or it can be implemented on top of a relational database system. If mass storage and high-performance protocols are needed, a heavy-duty system such as SRB can be used as the Storage service instead.

### 3 Metadata Service

A metadata record comprises a vector of attributes (name-value pairs) and can be represented as an XML document, (a set of) SQL table(s), a Java object, or otherwise. Each record is an instance of a *resource class* that defines the metadata schema for resources of that class. The schema defines the set of mandatory attributes for which the values must be provided, a set of optional attributes, the attribute types, and optionally their default values or acceptable range of the values. The attribute types can be either scalar (String, Long, Double, Date and Boolean, as defined by the SQL standard) or vector (vector of Strings, vector of Longs, etc) to support multi-value attributes.

The administration of the repository and the access control (Section 5) require that schemas for all resource classes contain certain attributes, such as creation date and resource ownership. Therefore the list of the resource class attributes is divided into two parts: a fixed, common attribute set modeled after the Dublin Core [2] and a variable set of attributes that change from class to class, depending on the application needs. Definition of the variable part of the schema is left to the user.

In practice, however, such a solution is not flexible enough, as the users often modify their metadata schemas as they progress in their research. Any modification to the schema after substantial data entries have been submitted to the repository introduces a risk of losing data integrity, short of re-entering all data sets with metadata conforming to the new schema. Therefore a notion of a *resource class version* has been introduced. It is the class version that actually defines the variable part of the metadata schema, so that a given resource class may have concurrently several schemas associated with it. It gives the user flexibility of querying all instances of a given class, regardless of the resource class version (if a query refers to a value of the attribute which not defined in a particular version, its value is set to null, as opposed to undefined if the value of an optional attribute has not been set), or querying only instances of a selected class version that guarantees the presence of the attribute value in question. The relationship between metadata entities are schematically shown in Figure 2.



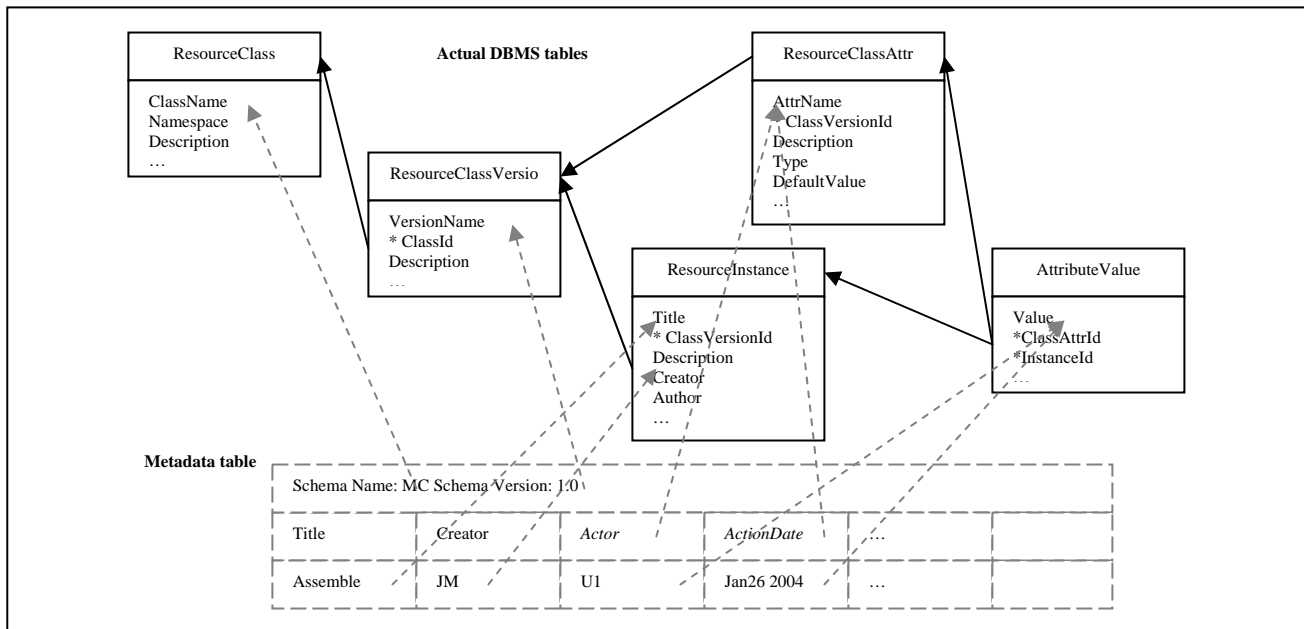
**Figure 2:** The relationship between the metadata resource classes, resource class versions, and instances

Having all instances of a certain resource class version in a single “bag” is not convenient.. It may require complicated queries to eliminate unwanted results and makes access control more difficult, at least for the end user. It is much more convenient to group data resources into collections or folders and assign access control privileges to the entire folder rather than individually. Also, by grouping data resources the user has better control over how data is organized. The folders allow metadata to be arranged like a hierarchical file system with different file types. For example, metadata that belong to different resource classes but are from the same experiment could be grouped into one folder.

The metadata service implements the following groups of operations:

- *Create, update, and delete* a class or a class version
- *Create, update, delete, and get* class of an instance
- *Add, update, delete and get* folders and metadata in folders
- *Search* for matching metadata given a search query

These operations emphasize adherence of metadata to its schema and deviate from models of other systems, e.g., the SRB, that supports metadata entry without a specific schema. The class version and instance functions allow the client to discover metadata requirements and reduce monotonous data entry errors.



**Figure 3:** Metadata schemas represented as RDBMS tables

Our implementation of the metadata service uses a set of RDBMS tables (Figure 3) with a J2EE container. Actual metadata (here *resource instances*) are stored in *ResourceInstance*. Classes, class versions, and class attributes are stored in the

corresponding tables. Common metadata attributes are stored along with an instance itself, and variable attributes are stored in *AttributeValue*, which is related to the resource instance and resource class attribute tables (Figure 3).

Not surprisingly, such a scheme of tables is similar to how most relational database systems store information about their tables and table columns. This allows the metadata to be searched using time tested, flexible SQL style queries in similar manner to how regular databases use SQL. Thus, these tables along with the query engine allow the metadata service to perform efficient queries on the virtual metadata tables. The J2EE architecture makes the implementation database independent and improves scalability of the metadata service.

## 4 Data Storage and Replica Locator Services

### 4.1 Data Storage Service

The primary concern of a data storage service is to allow efficient access to the data resources. While most specialized processing and visualization tools would use special protocols to access data storage, most common clients require a uniform interface to access different data storage services. For this reason, we use an adapter architecture [3]. Each type of data storage service, like GridFTP, requires its own adapter implementation in this model.

The adapter supports the following operations in the data storage service:

1. *openInputStream*: Allows a client to stream a file to the service. This operation is used for data storage from a streaming input.
2. *openOutputStream*: Allows a client to stream a file directly from the data service without intermediate storage
3. *copyFrom*: Copy data from a given source to the data service.
4. *copyTo*: Copy data to a given destination.
5. *getInfo*: Returns essential data resource information like size and creation date.

Our implementation uses GridFTP with its corresponding adapter for the data service.

### 4.2 Replica Locator Service

The replica locator service is concerned with maintaining physical URLs for a given metadata URI. It supports the following operations:

1. Insert replica: creates a map between the resource logical name (e.g., the resource URI) and physical location of the resource (e.g., the resource URL, or an identifier recognized by the Storage service)
2. ListReplicas: List all replicas with their physical locations for a given logical name
3. Remove replica: removes the resource physical location.
4. Remove a resource: the entry of the resource (i.e., URI) is removed together with all the mappings to the physical locations of the resource.

This model of the Replica Locator does not guarantee the consistency of the repository. Subsequent invocations of the service's methods may lead to the situations where a logical resource has no mapping to a physical location, or a physical resource may be left as an "orphan", i.e., there is no logical resource that corresponds to it. This is unavoidable, when the repository is implemented as independent services. The consistency of the repository is achieved using additional mechanisms described in Section 6.

The simplest implementation of the replica service could use a trivial one to many map to store the URI to URL mappings. Globus provides a ready-to-use replica locator service for this purpose. It also bundles a data replication service that could replicate files to a user specified location with its GT4 suite of services. More sophisticated implementations could recognize access patterns and replicate frequently used data resources to a local data service automatically.

## 5 Security Model and Authorization Service

### 5.1 Security model overview

The repository is designed to facilitate ad-hoc collaborations that require sharing of restricted-access data between members of dynamically created groups. Consequently, the repository must enforce strong group-based access control mechanisms.

The repository access control mechanisms are based on privileges assigned to groups to perform specified operations on the resources. Each resource is associated with one or more groups. It also has a set of permission flags which specify permissions for the group members to perform operations on the resources. The user may be a member of a several groups, with each group having different privileges, that is, rights to perform specific operations on the same resource. For example, let the resource permission flags be set to grant members of group A read access to a file and members of group B delete privilege to the same file. To download the file from the repository, the user must prove that she belongs to group A, and to delete the file from the repository, she must prove that she is a member of group B. If she is a member of both groups, she will be allowed to read and/or delete the file.

There are two special types of groups: a public group, and private groups. The public group comprises all repository users, and each user is a sole member of his or her private group. By default, the private group of the resource creator is the owner of the resource.

Each user, by default is the member of his private group and the public group, and has a right to create new groups. The creator of the group has the moderator privileges, that is, he has the right to accept other users as the group members. The owner of the resource (by default the creator of a resource) has the resource administrator privilege, that is, she has the right to set the resource permission flags. This forms foundations for the ad-hoc collaborations. In order to share restricted-access resources, the user creates a new group, and as a group moderator adds the collaborators to the group. Then the collaborators, including the group creator, “donate” resources they own to the group by modifying their permission flags.

Modifying the permission flags individually for each resource to be shared by a group may prove to be a very tedious procedure. Folders could be used to arrange data and provide a more convenient method to set privileges for a collection of resources. In the absence of the user specifying privileges for data, the privileges of the containing folder are copied to the data. Thus, the folders, much like their regular file system counterparts could be used for grouping related data all of which have the same privileges.

Since the metadata service (which can be distributed) is a catalogue of all resources available in the repository, it is natural that the resource permission flags are maintained there. Controlling the access to metadata is important since the metadata records may reveal too much information on the contents of the data, and in addition, the very existence of the data may be confidential.

The group membership is independent of the data (or metadata) and must be maintained separately by a service that is recognized by the metadata service as an authorization authority, which we refer to as Group Membership Authorization Service (GMAS). The access to a metadata record is thus granted to the requests from users that have a certification from GMAS on the group membership, and the metadata record has the permission flags set to allow the group to perform the operation on the metadata record specified in the request.

A successful query of the metadata system results in returning the URI of the metadata record that matches the query criteria, and by “successful query” we mean that the metadata record in question exists and the user has been granted access to it. This URI may be used to access the data, that is, to request the RL service to resolve it to a URL and request the Storage service to perform an operation on the data resource. It is easy to envision scenarios in which the user (or intruder) bypasses the authorization mechanisms associated with the metadata service, and therefore the storage service must be protected against an unauthorized access independently. As usual in the Grid environment, we assume that the access to the storage (through the storage service) is controlled following policies established by the storage stakeholders. For simplicity we assume here that the standard GSI[4] mechanisms are applied, that is, that the user must authenticate himself to the service using a X509 certificate issued by a certificate authority honored by the storage stakeholders, and the user has entry in an Access Control List such as the Gridmap file. Our design focuses instead on fine grained access control mechanisms for ad-hoc collaborations. To achieve that, the general storage access mechanisms must be combined with the rights implied by the membership to the group. Here it is assumed that the user has identical rights with respect to the data as with respect to the corresponding metadata. For example, if the user has the right to download the metadata record, she has the right to download the corresponding data as well. Authorization for the access of resource metadata is made by the metadata service. Consequently, the metadata service acts as the authorization service for the Storage service. It follows that there must a trust relationship between the metadata service and the storage service. This relationship can be established in two ways. Either the storage service directly trusts the metadata service, that is, it accepts authorization assertions issued by the Metadata service, or an intermediary is used: an authorization authority trusted by the Storage services signs the assertions on the metadata service’s behalf; GMAS may serve this function. The concept of the authorization authority is very similar in concept to Globus’ Community Authorization Service (CAS) [5].

In our current implementation, the metadata service and the storage service are deployed in the same security domain, and therefore we selected the direct trust between the metadata and storage services.

Finally, the Replica Locator service is protected the same way as the storage service. Since, in order to access any data resource the user must contact the metadata server to obtain the authorization assertions, the assertions are presented to the replica locator service. This is particularly important for removing entries from the replica locator.

## 5.2 Group Membership Authorization Service

The data and metadata access privileges are granted (or denied) based on the group membership. In a dynamic collaboration environment, groups are frequently created and destroyed, and users are added or removed from the groups. Hence, all assertions are time bound. The user retrieves such a time bound signed group privilege assertion (Figure 4) from GMAS and uses it to access the metadata service.

The GMAS serves to assert the rights of the user with services supported by GMAS. These rights are issued as SAML[6] assertions. The client embeds SAML assertions in the user's certificate when it contacts the respective services.

```
<Assertion AssertionID="_c4741fac99bb068426773c4f3de3af03" ...>
  <AttributeStatement>
    <Subject><NameIdentifier ...>CN=User,OU=erc.msstate.edu,O=Globus,O=Grid</NameIdentifier></Subject>
    <Attribute AttributeName="Groups" AttributeNamespace="urn:ecs:1.4:vo" ...>
      <AttributeValue xsi:type="xsd:string">uri://erc.msstate.edu/groups/Group1</AttributeValue>
      <AttributeValue xsi:type="xsd:string">uri://erc.msstate.edu/groups/Group3</AttributeValue>
    </Attribute>
  </AttributeStatement>
  <ds:Signature>
    <ds:SignedInfo><ds:CanonicalizationMethod .../><ds:SignatureMethod .../>
      <ds:Reference URI="#_c4741fac99bb068426773c4f3de3af03">
        <ds:Transforms>...</ds:Transforms><ds:DigestMethod .../><ds:DigestValue>...</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
    ...
  </ds:Signature>
</Assertion>
```

Figure 4: Group privilege assertion

To summarize, the authorization service implements operations for

- Adding, deleting and updating information about users and groups
- Managing user group membership
- Issuing SAML assertions for groups and optionally other service specific assertions.

Our GAMS implementation extends the Globus CAS implementation that authorizes the users using their X509 certificates. It adds two additional roles: (1) *manager* to add groups (2) *moderator* to add users to groups, to distribute the user and group administration and offload the burden of the GAMS administrators. It also stores additional information about users/groups and logs all activities regarding users and groups for accountability.

## 5.3 Access control for metadata service

In many cases, it is necessary to hide information stored in metadata from unauthorized users, or in extreme cases, the very existence of the data and corresponding metadata must be hidden. For these reasons the metadata service allows several access privileges to be set on its metadata class, class versions and instances. These privileges are:

- Select: Allows the metadata to be found during a query operation
- Read: With this privilege, the metadata and data can be read by the user
- Write: Provides ability to update metadata and overwrite existing data
- Create: Applicable to only class and class versions. If set on a class, it allows users to create class versions and if set on a version, it allows users to create instances of this class version.
- Delete: Allows the metadata and data to be deleted.
- Admin: This privilege allows the user to administer the object and modify the permissions to all groups on this object.

The Metadata service stores resource permission flags and enforces access privileges on its objects by comparing the resource permission flags with the group membership from the trusted authorization service (GMAS).

The Metadata service also acts as the authorization service for the Replica Locator and Storage services. When the user is granted access to a metadata record (read, write, or delete), the service generates the assertion in the same way as the GMAC generates assertions on the group memberships.

#### 5.4 Access control for replica locator

The Replica Locator grants permission to perform operations based on assertions issued by the Metadata Service. In the current implementation, the Replica Locator service trusts the Metadata service directly. An intermediary trusted Authorization service could be used instead.

#### 5.5 Access control for storage

Data access control in the storage service needs to be synchronized with metadata service privileges for data authorization. Most traditional systems such as SRB achieve this synchronization by combining the storage and metadata services. The services either share a user session or belong to a single administrative domain. But, this is a limitation in a distributed environment. In this case, the storage service should allow data access based on assertions from an authorization service. Here the Metadata service acts as the group-based (“community”) Authorization service that augment the native Storage services access control mechanisms. As in the case of the Replica Locator service, the Storage service trusts the Replica Locator service directly. An intermediary trusted Authorization service could be used instead.

### 6 Repository Façade: Remote Access to the Repository

Construction of the repository from the independent services has many advantages. Among the most important is the flexibility to accommodate multiple implementations of each service. For example, a light-weight combination of a file system and GridFTP can be used as the Storage service. However, if large files are involved, a heavy-duty SRB can be used instead.

On the other hand, it is difficult for the end user (or the user agent) to use such a distributed system. Many conceptually simple operations, such as retrieving a file, require making multiple requests to different services. The above mentioned flexibility of the system adds to the complexity of using the repository: the user must not only know the right order of service invocations, but also she need to know interfaces of the particular services and the communication protocols used by these services. Furthermore, since all services act independently from each other, maintaining repository consistency becomes difficult (e.g., each metadata record corresponds to at least one data resource and each data resource has associated metadata records).

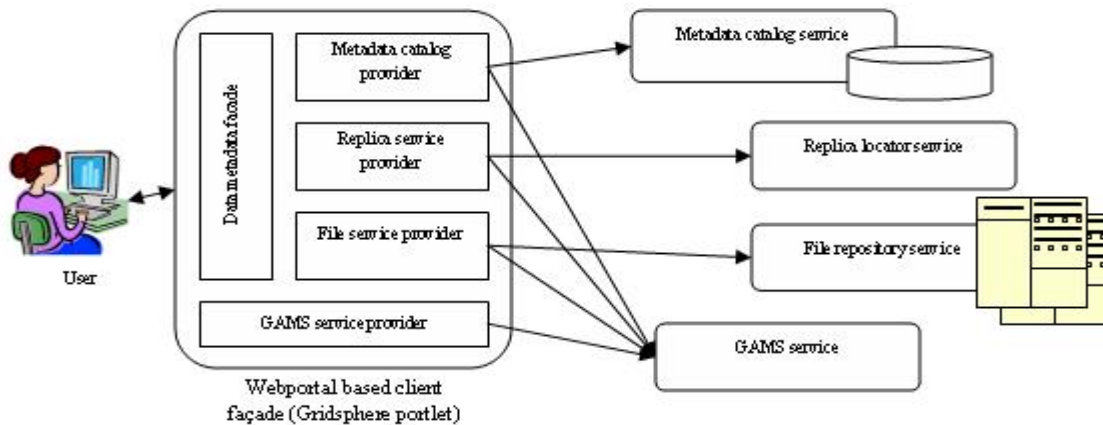


Figure 5: The architecture of the repository service

To ease these difficulties, all constituent services of the repository are hidden from the user behind a façade which is the middle-tier service that orchestrates the services and exposes a simple, intuitive interface for the end user. Now, given the URI of the resource (from querying the metadata service or otherwise), there is a single method of the façade to retrieve the file. The invocation of this method results in all actions needed to get the authorization assertions, resolving the URI to the physical locations and returning the data stream. To achieve that, the façade employs the business delegate pattern [7] that

delegates the actual implementations of the back-end service clients to service providers which understand the transport and assertion requirements of the corresponding services. In addition, the provider-based architecture makes it easy to retarget the repository to a different implementation of any of the constituent services – it requires just replacing the service provider without the need for modifications of the façade interface. The business logic of the façade also enforces the integrity of the repository. For example, a request to remove a data resource from the Storage automatically triggers appropriate actions of the Replica Locator and Metadata services initiated by the façade. The final architecture of the repository is given in Figure 5.

## 7 Conclusions

This paper presents and discusses design and implementation of a repository service for Grid environments that supports secure sharing of possibly confidential data by members of ad-hoc created groups. We found three factors to be critical for our design: the composition of the repository from independent services, the authorization mechanisms for such a distributed system, and the rich capabilities of querying metadata with flexible schemas.

The separation of the repository into independent services gives us the flexibility of mixing and matching different services as needed. In particular, it allowed us to concentrate on developing a flexible metadata service, rather than reproducing the functionality of existing systems such as SRB. Dealing with the distributed system is difficult for the clients, and therefore we provided the middle-tier façade that orchestrates the services for the user and maintains the integrity between the Metadata and Storage services.

We developed flexible yet secure group-based authorization mechanisms. The system is flexible because it allows the users to create groups on-the-fly (a sort of “mini virtual organizations”) without contacting the system administrator and to tune the metadata schemas to their needs. The security is achieved by augmenting the user credentials required by the back-end service owner (i.e., the Storage) by a chain of authorization assertions added by the repository services: GMAS, Metadata service and Replica Locator service. To make the complete system work, the trust relationship between the constituent repository services must be established, that is, they must honor each other as the authorization authorities. Our solution builds on and extends those of GT4 Community Authorization Services.

Finally, the metadata service introduces a very flexible system for storing and querying the metadata. Each metadata record conforms to a schema. The schemas define the variable length vector of attributes that can be easily modified by introducing new versions without intervention of the system administrator. In addition, the user may organize the metadata entries into a tree hierarchy of indefinite depth that simplifies identification of data resources and the assignment of permission flags. In spite of the apparent complexity in metadata organization, the user may employ the full power of SQL to search the data objects of interest.

The system has been deployed at Mississippi State University and it being tested by sharing data (the captured motion data used by researchers in ergonomics) with our corporate partners. In many cases proprietary data are being shared between the corporations and researchers at MSU under a nondisclosure agreement.

## REFERENCES

- 
- [1] A. Rajasekar, M. Wan, R. Moore, W. Schroeder, G. Kremenek, A. Jagatheesan, C. Cowart, B. Zhu, Sheau-Yen Chen, R. Olschanowsky, “Storage Request Broker – Managing Distributed Data in the Grid,” Computer Society of India Journal, Special Issue on SAN, Vol. 33, No. 4, pp. 42-54 Oct 2003
  - [2] S. Weibel, J. Kunze, C. Lagoze and M. Wolf, “Dublin Core Metadata for Resource Discovery,” [Online document] Sept 1998, [2006 Apr 19], Available at HTTP: <http://www.ietf.org/rfc/rfc2413.txt>
  - [3] Sun Developer Network, “Core J2EE Patterns – Business Delegate,” [Online document] 2001, [2006 Apr 19]. Available at HTTP: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html>
  - [4] The Globus Project, “Overview of the Grid Security Infrastructure,” [Online document] 2001, [2006 Apr 19]. Available at HTTP: <http://www.globus.org/security/overview.html>
  - [5] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke, “A Community Authorization Service for Group Collaboration,” in Proc. of IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, Monterey, California, June 2002
  - [6] OASIS Security Services TC, “Security Assertion Markup Language (SAML),” [Online document] Nov 2002, [2006 Apr 19], Available at HTTP: <http://www.oasis-open.org/committees/security/>
  - [7] Sun Developer Network, “Core J2EE Patterns – Business Delegate,” [Online document] 2002, [2006 Apr 19], Available at HTTP: <http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html>