
# IVOA VOStore
# Version 0.17

## IVOA Working Draft
## 2005-05-05

**This version:**

**0.17 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOStore0.17.pdf**

**Previous versions:**

**0.15 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOStore0.14.pdf**

**0.13 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOStore0.13.pdf**

**Editors:**

Dave Morris, William O'Mullane, Guy Rixon., Mathew Graham, Ani Thakar

**Authors:**

IVOA Web and Grid Services Working group

**Please send comments to: webservices@ivoa.net**

## Abstract

This document describes the VOStore concept

## Status of this document

This is a Working Draft. There are no prior released versions of this document.

*This is an IVOA Working Draft for review by IVOA members and other interested parties. It is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use IVOA Working Drafts as reference materials or to cite them as other than "work in progress." A list of current IVOA Recommendations and other technical documents can be found at http://www.ivoa.net/docs/.*

## Acknowledgments

This work is based on discussions at various IVOA meetings and continuing emails on the mailing list.

Contents

# 1 Introduction

The initial and strong driver for VOSpace remains the integration of various technologies which allow users to store data through a web based service. The two storage services in the Virtual Observatory realm which we would particularly like to integrate are MYDB[1] and MYSpace[7]. In addition, the Storage Resource Broker[9][10] system is already used in a number of other grid architectures. Integrating this into VOSpace would enable us to use SRB as a storage mechanism within the virtual observatory, and to inter-operate with other grid systems already using SRB.

Large scale analyses will require access through high-performance interfaces[8]. Should we also be considering LDAP[3] or WS-Transfer [11].

# 2 VOSpace and VOStore

In order to support high level storage systems such as MYDB[1], MYSpace[7] and Storage Resource Broker[9] as well as lower level systems such as HTTP web servers, FTP and GridFTP servers, the VOSpace concept has been separated into two layers, VOSpace and VOStore.

The VOSpace layer will provide the higher level abstractions, integrating multiple storeage systems into a global virtual file system.

The VOStore layer will define a common interface that all of the storage systems must implement, enabling a VOSpace service to move data between storage locations without having to handle multiple different interfaces.

## 2.1 Goals of VOStore

The main goal of VOStore is to provide a uniform interface to existing or new data storage locations. Figure 1 below depicts the VOStore interface being implemented on three different systems, which currently exist in the Virtual Observatory realm, MYDB, MySpace and SRB. The notion is to keep the interface simple so that it should be easy to implement beside existing interfaces. Each of these systems has the capabilities to accept and transmit files – however each does it in a slightly different way. VOStore would define a common interface which should be relatively easy to implement on such systems hopefully it would simply be a ‘facade’[1] in terms of programming patterns. The particular transport mechanism for the data may be brokered between the stores e.g. if both support gridftp that could be used.

The different VOStore providers would then, of course, be put in the registry where they could be looked up. We would refer to a Store by some logical name, specifically an IVOA Identifier[5]. Access to the store would require authentication, nominally we would use the Single Sign On[6] approach emerging in the Web and Grid services Working group. This would certainly imply the use WS-Security.

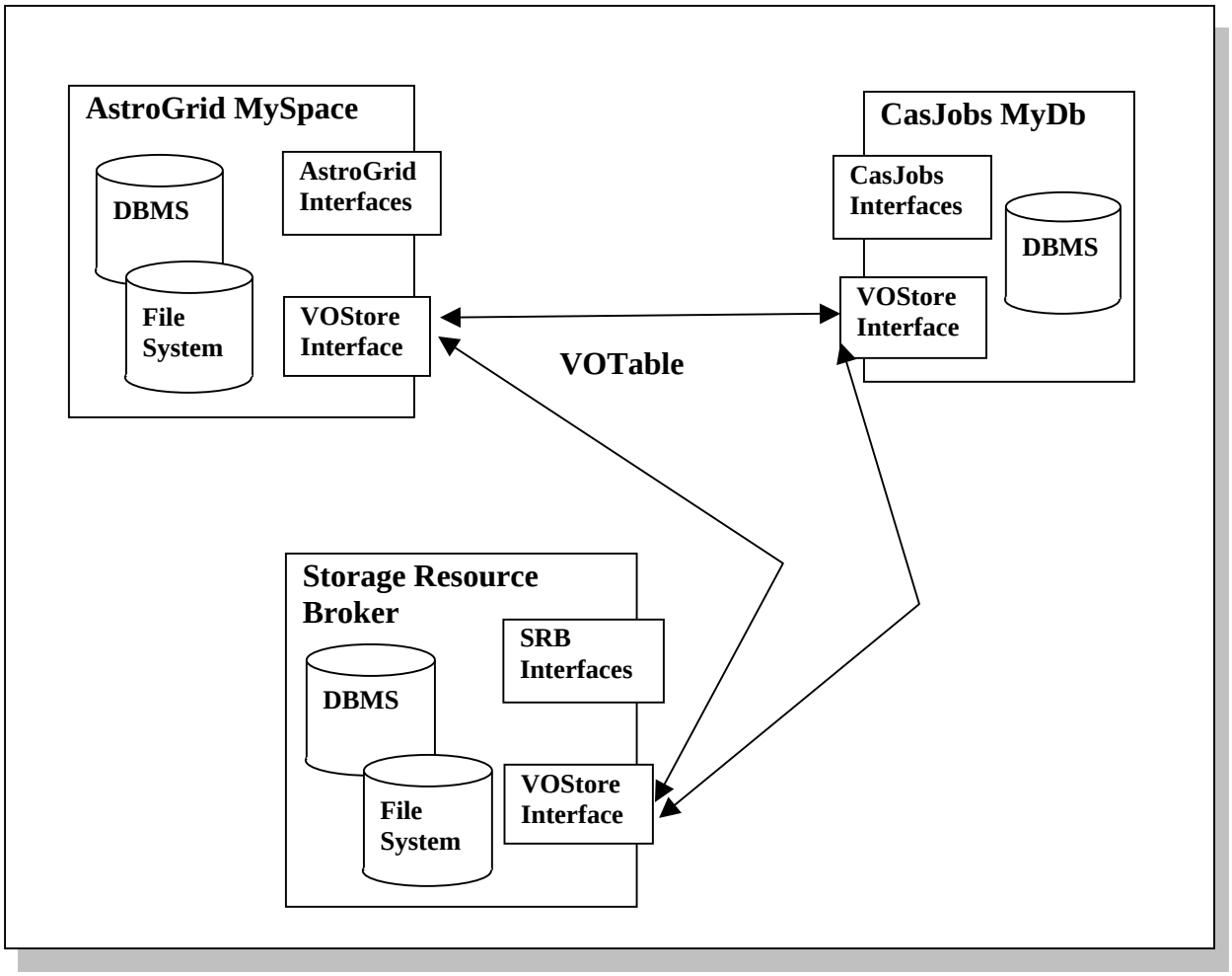


**Figure 1.** Example VOStore interfaces

It is becoming clear that more sophisticated access is needed to databases than simply get and put of a table. There are other IVOA interfaces for interacting with databases and it seems this is the correct place for handling queries. On the other hand one may want a subset of some data. In the MYDB model of CasJobs this is handled by two interfaces, first the user makes the subset in the MYDB , then the entire table may be requested as a file. So we could envision a particular query system allowing “Select x into vostore:y.z from tab where x > 10” but the store still only needs get and put to support this, the service supporting the query will have to do the hard work of formulating that get or put..

## 2.2 VOSpace

The best analogy for VOSpace, and the one we are currently working toward, is that of a global file system. The VOSpace would manage access to a number of VOStores and turn them into a global file store. The VOSpace would handle authentication and support multiple access protocols. In this way VOStore may remain somewhat simpler allowing more complex interactions to be handled by a VOSpace portal. This also means the higher layer is isolated from the peculiarities of the lower layer implementations.

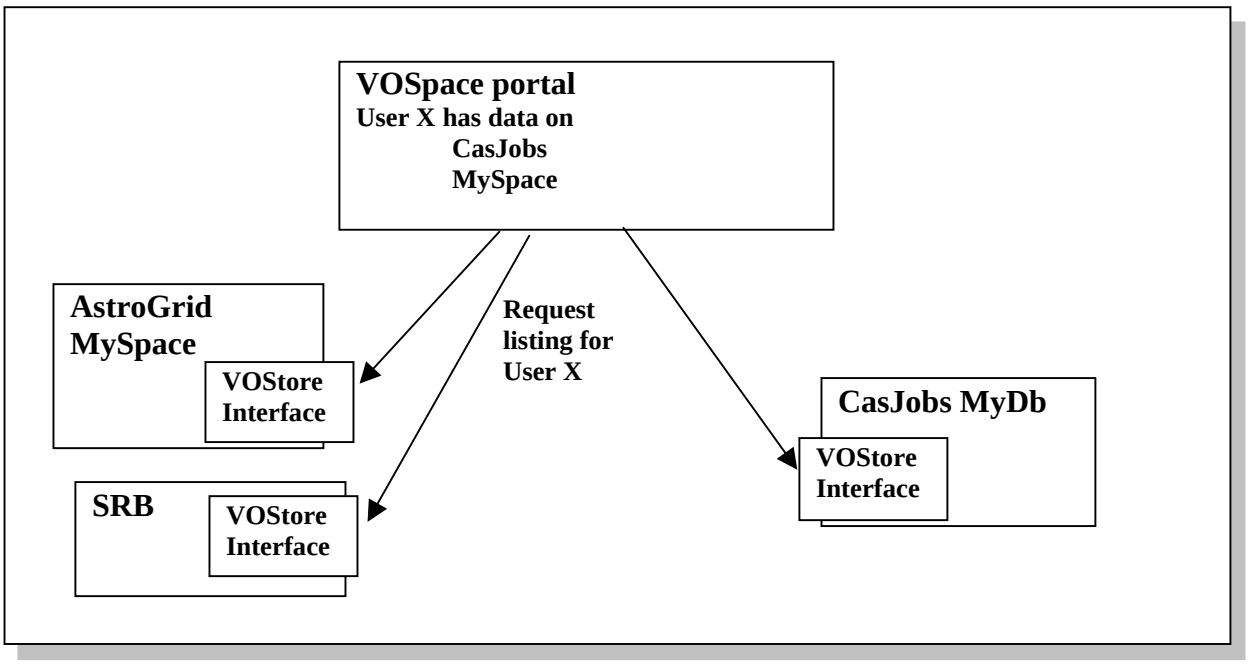


**Figure 2.** Example of VOStore portal concept. The portal queries 3 resources looked up from the registry the user has data on only two of those.

For a model for how this could be used consider an asynchronous version of SIA. Following the query, a client issues a staging request to the service to generate a number of images. By default the service would generate the images and place them in a local VOStore. Either messaging or polling could be used to notify the client when the images are ready for retrieval. Alternatively, the client (via the staging request) could direct the service to place the generated images in a remote VOStore. This could be the client's VOStore, close to where analysis will take place, or possibly a third-party VOStore, e.g., for a complex workflow.

Hence VOSpace may be seen as the federation/distributed directory service layer on top of this. The users can view their holdings across multiple VOStores and move and copy things across. It may be viewed as a single authentication directory service. There may be an associated registry-like function which is updated via harvesting the VOStores. The locations of VOStores should be obtained from the Registry.

This document does not define or specify VOSpace - this is a complex topic and requires its own specification. VOSpace is mentioned here only to provide some slightly broader context for VOStore.

## 2.3 Security

Security is seen as orthogonal to VOStore. The issues of single sign on and use authentication need to be solved for many systems in the VO and all should do it in the same way. The manner of doing the security has no impact on the interface defined in this document. Experiments are being conducted in java and Csharp using WS-Security which confirm this view.

There is an issue with some interfaces below (e.g. import export) implying a secure access to an implied endpoint like a URL. How that security is managed is a matter for the implementer of the particular service. If a VOStore implementation returns me a plain URL with no security to pick up a file that is a matter for that service – it should not return a URL for someone else's file. If the protocol used is WebDav or GridFtp this implies a level of security for picking up the object. We do not wish to tie down all protocols here and define what they mean. We do suggest a minimum of SOAP-Attachment which would imply the security is covered by the message the object is attached to.

# 3 VOStore

**VOS-1** VOStore shall implement the mandatory interface methods defined in the VO Support Services specification[2].

**VOS-2** VOStore shall be defined in terms of a SOAP Service with a WSDL for the precise definition of the interface.

**VOS-3** VOStore shall support as a minimum VOTable for tabular transmission and acceptance of data.

**VOS-4** VOStore shall support the "formats" interface method. This shall return the list of formats supported by the service. The Formats shall be in the form of strings containing mime types. Simple file based systems that do not interpret the data can declare that they accept 'ANY' and return the data in the original format.

**VOS-5** VOStore shall support the "transports" interface method. This shall return the list of transports supported by the service. These will take the form of strings. All services shall support SOAP-ATTACHMENT.

**VOS-6** VOStore shall support the "Get" interface method. This method shall take as arguments the identifier of the container on the server and the required format (as listed in VOS-3). In response, the service will reply with the properties of the container and the contents of the container as a SOAP attachment. If the requested format is not supported this may throw an exception. This shall return a VOResponse document or a document derived from that. This will contain a message and status information.

**VOS-7** VOStore shall support the "Put" interface method. This method shall take as arguments the upload format (as listed in VOS-1), an optional set of properties and the data to store as a SOAP attachment. In response, the service will transfer the data from

the SOAP message into a new container and reply with the identifier for the new container and an updated set of properties. If the data format is not supported this may throw an exception. This shall return a VOReponse document.

**VOS-8** VOStore shall support the "List" interface method to list resources per user and for the entire server. This shall return a set of VOStoreDescriptors – a descriptor shall contain the owner, modification date and Identifier of the objects as well as a PropertyPair set to allow arbitrary Name Value pairs to be returned.

**VOS-9** VOStore shall support the "Delete" interface method. This method shall take the identifier of the container to be deleted.

**VOS-10** VOStore should support the "importInit" interface method. This method shall take as arguments the transfer protocol to use (as listed in VOS-5), the data format (as listed in VOS-3), and an optional set of properties. In response, the VOStore will create a new container, and reply with the identifier for the container and a URL that enables the client to transfer the data into the container. If the data format or transfer protocol are not supported, this may throw an exception.

**VOS-11** VOStore should support the "importData" interface method. This method shall take as arguments a URL pointing to the current location of the data, the transfer protocol to use (as listed in VOS-5), the data format (as listed in VOS-3), a number of retries (default=1), and an optional set of properties. In response, the VOStore will transfer the data into a new container, and reply with the identifier for the container and an updated set of properties. If the data format or transfer protocol are not supported, this may throw an exception.

**VOS-12** VOStore should support the "exportInit" interface method. This method shall take as arguments the identifier of the container, the transfer protocol to use (as listed in VOS-5) and the data format (as listed in VOS-3). In response the VOStore will reply with the current properties for the container and a URL that enables the client to access the data. If the data format or transfer protocol are not supported, this may throw an exception.

**VOS-13** VOStore should support the "exportData" interface method. This method shall take as arguments the identifier of the container, a URL indicating where to send the data to, the transfer protocol to use (as listed in VOS-5), a number of retries (default=1), and the required format (one of those listed in VOS-3). If the data format or transfer protocol are not supported, this may throw an exception. In response, the VOStore will transfer the contents of the container to the specified location.

**VOS-14** VOStore shall support a "status" interface method if it implements any of the import/export methods. This interface shall take as input the container Identifier concerned and should return the status of the import/export – shall return a VOResponse where the message is an indication of completeness of the task. The status of the response would relate to the container rather than this call.

## 3.1 VOStore identifiers

In order to support a wide range of different service implementations and clients, containers within a store should be identified using unique identifiers allocated by the store.

When data is imported into a VOStore service, the service creates a container for the data, and returns a unique identifier for that container. All future access to the data is done with reference to the identifier for that container.

It is up to the client to keep hold of the identifier in order to be able to request further actions on the data in that container.

Unlike filenames, once an identifier has been allocated, it cannot be re-used at a later time.

The proposed format for a VOStore identifier is to append a server generated identifier to the IVO registry identifier for the service using the URI fragment notation, 'uri:identifier#'fragment.

For example, if the IVO registry identifier for a VoStore service is

'ivo://authority/resource'

then an identifier for a container within that service would be

'ivo://authority/resource#container'

This allows different VOStore service implementations to use different mechanisms for generating unique identifiers for objects within the service.

For example, a database implementation of a VOStore service may use a simple integer allocated by the database to generate sequential identifiers.

'ivo://authority/resource#0000

'ivo://authority/resource#0001

A file based implementation of a VOStore may use a different mechanism for generating unique identifiers.

'ivo://authority/resource#a53948.1033bcd63aa.7fe3

'ivo://authority/resource#a53948.1033bcd63aa.7fdf

Basing the VOStore identifier on the IVO identifier of the service allows service implementations to use fairly simple mechanisms to generate the identifiers, while still ensuring that that the identifiers are globally unique. This also enables a client to locate and access data within a distributed VOSpace system, by resolving the service identifier of the VoStore via the global registry.

## 3.2 VOStore properties

A VOStore service needs to provide a mechanism for storing descriptive metadata for each object.

In order to make the system as flexible as possible, the metadata properties are handled as a map of name value pairs for each container.

The VOStore specification may define the minimum set of properties for an object, but 3rd party services can add any number of additional properties to the set. As far as the VOStore service is concerned, the additional properties are treated as pairs of strings.

This enables 3rd party service to define and use their own properties, without requiring the VOStore system to understand or interpret them.

In order to allow different organizations to add their own properties without causing name conflicts, the recommended naming convention for properties is to use the organization domain name in reverse as a prefix for property names.

Properties defined by IVOA as part of the VOStore specification would start with 'net.ivoa.vostore'

e.g.

net.ivoa.vostore.createdate

net.ivoa.vostore.mimetype

etc

Additional properties defined by AstroGrid FileStore services would start with 'org.astrogrid.filestore'

e.g.

org.astrogrid.filestore.community

org.astrogrid.filestore.jobident

## 3.3 Import and export

The core function of a storage service is to enable clients to import and export data to and from the service. To support these actions, a VOStore service provides the following four import and export methods.

For simple access to small data objects, a VOStore service needs to be able to transport data in the SOAP message using the SOAP attachments mechanism.

For medium sized data sets, a VOStore service needs to be able to transport data using a binary transport protocol such as HTTP, FTP and GridFTP.

For very large data sets, a VOStore service needs to be able to transfer data asynchronously, using SOAP messages to initiate the transfer and a callback mechanism to update a client when the transfer completes.

In addition, binary protocols such as HTTP and FTP can transfer data in two directions. The client can send the data, in a PUT transfer, or recipient to request the data using a GET transfer.

To support all of these options, the VOStore specification defines four interfaces to support synchronous and asynchronous GET and PUT and transfers.

ImportInit and Import data provide interfaces for importing data into a VOStore.

ExportInit and ExportData provide interfaces for exporting data from a VOStore.

### 3.3.1 ImportInit

This interface is used when a client has some data that it wants to send a VOStore. In response the VOStore replies with a URL that enables the client to send data to the store.

The client calls ImportInit, and specifies the data format (as listed in VOS-3), the transfer protocol it wants to use (as listed in VOS-5), and any additional properties it wants to associate with the data. The client can also supply an identifier for a callback service to notify when the transfer is completed.

In response, the VOStore service creates a new (empty) container for the data, and replies with the identifier for the container and a URL that enables the client to transfer data into the container.

The client can then open a connection to the URL and send the data to the container using the chosen protocol, e.g. HTTP-PUT.

If the client supplied a callback address in the original request, the VOStore service will call the callback service when the data transfer is completed with an updated set of properties. These should include information about the data received such as the size and some form of checksum. This enables a higher-level component such as a VOSpace service, to check that the data arrived intact.

The client should not assume anything about the URL supplied by the VOStore service beyond the fact that it enables the client to send some data to the container. In particular, the client should not assume that the URL can be re-used to send additional data to the same container, or that the URL can be passed on to a 3$^{rd}$ party who will perform the transfer on their behalf.

Restricting the use of the URL allows a VOStore service to encode transaction and security information in the URL, enabling the VOStore service to restrict who is allowed to transfer data into a container.

### 3.3.2 ImportData

This interface is used when a client wants the VOStore service to fetch some data from a remote location. In this case, the client provides the URL to the VOStore and in response the VOStore uses the URL to import the data.

The client calls ImportData, and specifies a URL pointing to the current location of the data, the transfer protocol it wants the service to use (as listed in VOS-5), the current data format (as listed in VOS-3), a flag to indicate if the transfer should be performed synchronously or asynchronously and any additional properties it wants to associate with

the data. The client can also supply an identifier of a callback listener that should be notified when the transfer is completed.

If the transfer is to be done synchronously, the VOStore service will create a new container for the data and use the URL specified in the request to transfer the data into the container.

When the transfer is completed, the VOStore service will reply with the identifier for the new container and an updated set of properties. These should include information about the data received such as the size and some for of checksum.

If the data is to be transferred asynchronously, the VOStore service will create a new (empty) container for the data, and schedule the data transfer to occur after the SOAP call completes. The VOStore service then completes the SOAP call, and replies with the identifier of the new container, and an updated set of properties.

Once the initial SOAP call has completed, the VOStore service will then use the URL specified in the request to transfer the data into the container.

If the client supplied a callback address in the original request, the VOStore service will call the callback service with an updated set of properties when the data transfer has been completed. These should include information about the data received such as the size and some for of checksum. This enables a higher-level component such as a VOSpace service, to check that the data arrived intact.

### 3.3.3 ExportInit

This interface is used when a client wants access to data stored in a VOStore service. The client provides the container identifier, and in response the VOStore replies with a URL that enables the client to access the data.

The client calls exportInit and specifies the identifier of the container, the required format (as listed in VOS-3) and the transfer protocol it wants to use (as listed in VOS-5). The VOStore service replies with the current properties for the container and a URL that will enable the client to access the data.

The client can then use the URL to transfer the data from the store.

The client should not assume anything about the URL supplied by the VOStore service beyond the fact that it enables the client to access the data in the container. In particular, the client should not assume that the URL can be used more than once, or that the URL can be passed on to a 3$^{rd}$ party who will perform the transfer on their behalf.

Restricting use of the access URL allows a VOStore service to encode transaction and security information in the URL, enabling the VOStore service to restrict who can access the data in the container.

### 3.3.4 ExportData

This interface is used when a client wants the VOStore to transfer data from a container to a remote location. The client provides the container identifier and the destination URL, in response the VOStore sends the data to the target URL.

The client calls exportData and specifies the identifier of the container, the required format (as listed in VOS-3), the transfer protocol it wants the VOStore to use (as listed in VOS-5), a flag to indicate if the transfer should be done synchronously or asynchronously, and a URL indicating where it wants the data to be sent. The client can also supply an identifier for a callback listener that should be notified when the transfer is completed.

If the transfer is to be done synchronously, the VOStore will open a connection to the destination URL and transfer the contents of the container. When the transfer is completed, the VOStore service will reply with the current container properties.

If the transfer is to be done asynchronously, the the VOStore will schedule the transfer to occur after the SOAP call has completed, and then reply with the current set of properties. After the SOAP call has completed the VOStore service will open a connection to the destination URL and transfer the contents of the container.

If the client supplied a callback address in the original request, the VOStore service will call the callback service with an updated set of properties when the data transfer has been completed. These should include current information about the data such as the size and some form of checksum. This enables a higher-level component such as a VOSpace service to check that the data arrived intact.

# 4 State Information

The proposed VOStore interface to storage repositories will interact with state information maintained within each storage repository. The state information managed in each storage repository includes:

- naming convention for users (distinguished user name space)
- naming convention for files and tables (logical file name space)
- access controls (constraints between the distinguished user name space and the logical file name space)
- file properties (size, creation date, owner, mime-type)
- IVOA identifier for the storage repository
- IVOA identifier for each file/table (Global Unique ID)

The state information may also include:

- descriptive metadata
- replication metadata

Which state information should the VOSpace federation level manage? The answer depends upon the degree of synchronization and consistency that VOSpace is intended to support across the multiple VOStore storage repositories. There are at least two global name spaces that are of interest:

- Distinguished name space for users. This is needed so that a single sign-on authentication environment can be created for access to all VOStores. The authentication system will make it possible for individuals to restrict access to

their private data, and system administrators to track storage utilization by individual. A very important requirement to avoid security problems is the ability to restrict write access to authorized persons.

One approach is to use GSI, and create a VO Certificate Authority to manage public key certificates. A user would be able to access public data and data that had been stored under his/her user account. Each VOStore would then need to establish user accounts for each name registered in the VO Certificate Authority.

- Distinguished name space for files. If VOSpace supports movement of files between VOStores, then VOSpace needs a global unique identifier for each file that can be used to track location. A GUID would allow VOSpace to query multiple VOStores, identify the location of replicas, and then provide access to the closest copy.

  If VOSpace tracks file replicas, additional state information will be needed to identify when changes have been made to a file. A "dirty" flag can be used to denote when a file has been modified, along with a modification date. VOSpace would then periodically query the VOStores to identify those files that have been modified since the last check, and synchronize replicas. A third piece of state information is needed to implement a write lock on a local file. This would prevent synchronization from occurring until after the write lock was released.

  The GUID, "dirty" flag, modification date, and synchronization locks could be stored within each VOStore as metadata associated with each file. If the VOStores are not capable of doing this, then VOSpace will need to manage a database that contains the state information for all files that have a GUID. This approach is similar to that used by the CNRI Handle system.

Such a system would be able to federate access to multiple VOStores, support data movement and data replication, and provide single sign-on authentication to access-controlled data. This approach can be characterized as a federation that builds upon a central repository managed by VOSpace. The central repository maintains state information needed to support the VOSpace user interfaces.

Given the need to manage state information that cannot be pushed down to the VOStores, a second approach can be implemented to support write access. VOSpace can be tasked with managing state information associated with user access to private data. A VOSpace account ID can be established in each VOStore. Users would issue a write request to VOSpace, which would then write the data into the VOSpace account within the VOStore. VOSpace would maintain a distinguished name space for users, maintain a distinguished name space for the files, maintain access controls on the data, and manage the state information for each file. Interactions with each VOStore would be through the VOStore interface. The state information that would be maintained by VOSpace would include:

- naming convention for users (distinguished user name space)
- naming convention for files and tables (logical file name space)
- access controls (constraints between the distinguished user name space and the logical file name space)

- file properties (size, creation date, owner, mime-type)
- file location within a directory hierarchy
- IVOA identifier for the storage repository
- IVOA identifier for each file/table (Global Unique ID)
- descriptive metadata
- replication metadata
- dirty flag, modification date, synchronization lock

The advantages are that the VOStore environments could be simple file systems or sophisticated data grids. The VOSpace environment would be able to maintain privacy for users without having to set up user accounts at each VOStore location. VOSpace would be able to maintain consistent state information about the files, and support replication between VOStores. If all files with GUIDs are registered into the VOSpace environment, then VOSpace could support descriptive metadata for browsing and discovery. The services that are installed on top of VOSpace would be decoupled from the underlying storage systems.

This could be implemented today by writing a VOStore driver for the Storage Resource Broker. This would enable the SRB to access and manipulate data within any of the VOStore storage repositories. The approach should also be feasible with Astrogrid MySpace and CASjobs MyDB. All that is needed is the ability to support schema extension within the associated database, and the registration of the information associated with users with private data. In effect, each of the proposed VOStores could be turned into a VOSpace that accesses data stored within another VOStore through the standard VOStore interface.

In the high-energy physics community, the equivalent VOStore interface is the “Storage Resource Manager” interface developed at Lawrence Berkeley National Laboratory. This interface is oriented toward asynchronous get/put of files. A WSDL version is being developed (SRM version 3). A description of version 2.1 is available at http://www.nesc.ac.uk/events/GGF10-DA/programme/papers/SRMInterfaceSpecification.pdf

For improved performance, it would be beneficial to support cross-registration of users and files between VOSpace instances. A user would be identified by their home VOSpace instance, and authentication would be done by a request back to the home VOSpace. (This is similar to the Shibboleth model.) A federated VOSpace environment would then be feasible, with each user able to access the system from their preferred VOSpace instance. Each VOSpace could impose their structural view on the data with their preferred descriptive metadata. Similar federation between data grids has been shown to work on a global scale with the SRB.

# 5 References

[1] Gamma E. et Al. Design Patterns ; Addison Wesley 1995.

[2] IVOA Grid & Web Services Working Group; VO Support Services; http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOSupportInterfaces-0.2.pdf

[3] Lightweight Directory Access protocol; http://www.activexperts.com/activmonitor/functions/ldap/rfc1777/

[4] O'Mullane W. et. Al., Batch Query System with Interactive Local Storage for SDSS and the VO ; ADASS XII 2003.

[5] Plante R. et Al.. IVOA Identifiers, http://www.ivoa.net/Documents/latest/IDs.html

[6] Rixon G., Single Sign On, http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/SSO-msg-protocol.html

[7] Walton N.A. et. Al., AstroGrid: Initial Deployment of the UK's Virtual Observatory; ADASS XII 2003.

[8] Moore, R., "Operations for Access, Management, and Transport at Remote Sites," Global Grid Forum, December 2003.

[9] Rajasekar, A.,M. Wan, R. Moore, "mySRB and SRB, Components of a Data Grid", 11th High Performance Distributed Computing conference, Edinburgh, Scotland, July 2002.

[10] Storage Resource Broker, http://www.sdsc.edu/srb/

[11] WS Transfer; http://xml.coverpages.org/WS-transfer200409.pdf

# Appendix A: SRB data and metadata access commands

A VOStore interface can provide multiple access mechanisms. The proposed WSDL service is useful for retrieval of small amounts of data. For large-scale analyses, additional interfaces are useful for bulk data access (retrieval of tens of thousands of files), parallel data access (use of parallel I/O streams to move large files), and data subsetting. The additional data movement commands may need to support interactions with firewalls and require both server-initiated and client-initiated parallel I/O stream invocation.

A VOStore can manage a collection, with operations defined for collection access, collection management, and data transport. The set of commands currently implemented in the SRB data grid are listed below. The VOStore could provide equivalent functionality – it would be interesting to decide the minimum set.

SRB has existing interfaces that support access through web services, web browsers, Java, Python, Perl, DSpace, OAI, shell commands, C library, Windows browsers, OAI, etc.

- Sannotate - provides a facility for inserting, deleting, updating, and accessing annotations on data objects
- Sappend - appends a local file or a SRB object to a target SRB object.
- Sattrs - lists the queriable MCAT attributes.

- Sbkupsrb - synchronizes copies of an SRB object across replicas
- Sbload - imports in bulk one or more local files and/or directories into SRB space
- Sbregister - registers in bulk one or more local files and/or directories into SRB space.
- Sbunload - exports in bulk a collection or a SRB container to local file system
- Scat - concatenate and display files read from SRB space
- Scd - change working SRB collection
- Schdefres - change default storage resource for the current session at the terminal.
- Schhost - change default connection host for the current session at the terminal.
- Schksum - Checksum SRB data files and/or collections.
- Schmod - changes permission modes and ownership for objects and collections in SRB space
- Scp - copies a srbObj or srbCollection in SRB space
- SdelValue - Deletes a SRB entry (user, physical resource, location)
- Senv - Displays SRB environmental file content
- Serror - describes SRB errors.
- Sexit - Ends a SRB session
- Sget - exports one or more objects from SRB space into the local file system
- SgetColl - display information about SRB data objects.
- SgetD - display information about SRB data objects.
- SgetR - display information about SRB resource(s).
- SgetT - display information about SRB tickets
- SgetU - display information about SRB user(s).
- Shelp - Displays a list of all available SRB S-commands
- Singestuser - adds an account to a SRB grid.
- Sinit - initializes SRB client environment.
- Sln - make links to srbObjects or to srbCollections.
- Sls - display objects and sub-collections in a SRB collection
- Slscont - list your SRB containers
- Smeta - modifies metadata information about SRB data objects.
- Smkcont - creates a new SRB container
- Smkdir - creates a new SRB collection
- SmodColl - modifies metadata information about a SRB collection.

- SmodD - modifies metadata information about SRB data objects.
- SmodifyUser - Modify user account settings.
- SmodR - modifies metadata information about a SRB resource.
- Smv - changes the collection for objects in SRB space
- Spasswd - changes password of current user
- Spcommand - Proxy command operation. Request a remote SRB server to execute arbitrary commands on behalf of client.
- Sphymove - moves an SRB object to a new resource
- Spullmeta - accesses metadata in bulk from a SRB.
- Spushmeta - ingests metadata in bulk into a SRB.
- Sput - imports one or more local files and/or directories into SRB space.
- Spwd - displays current working SRB collection
- Sregister - registers one or more files into SRB space.
- Sregisterlocation - register a new location (computer)
- Sregisterresource - register a new physical resource
- Sreplcont - Replicate a container copy.
- Sreplicate - replicates an SRB object
- Srm - Remove files from SRB space
- Srmcont - Remove a SRB container
- Srmdir - deletes a SRB collection
- Srmticket - remove a ticket
- Srmtrash - Purge files and collections in the "trash" directories.
- Srsync - Synchronize the data between a local copy (local file system) and the copy stored in SRB or between two SRB copies.
- Ssh - Invokes the SRB shell.
- Ssyncd - synchronizes copies of an SRB object
- Ssyncont - Synchronizes the "permanent" copy of the container with the "cache" copies.
- Stcat - display files read from SRB space for a ticketuser
- Sticket - issue tickets for access to SRB objects and collections.
- Stls - display objects and sub-collections in SRB collection for a given ticket
- Stoken - display information about SRB native types (data types, resource types, user types, domain, location, action, access constraints, zones, resource class)

- Sufmeta - modifies attribute-value-unit triplets of user-defined metadata information about SRB data objects.
- Szone - modifies metadata information about SRB data objects.