International

Virtual

Observatory

Alliance

VOSpace service specification

Version 1.0
IVOA Proposed Recommendation 2007 June 26

This version:

http://www.ivoa.net/Documents/PR/GWS/VOSpace-PR-1.0.odt

Latest version:

http://www.ivoa.net/Documents/latest/VOSpace.html

Previous version(s):

WD 1.0 http://www.ivoa.net/Documents/WD/GWS/VOSpace-20070304.pdf
0.21 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/vospace-0.21.doc
0.20 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/vospace-0.20.doc
0.19 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/vospace-0.19.doc
0.18 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOStore0.18.pdf
0.17 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices/VOStore0.17.pdf
0.15 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices /VOStore0.14.pdf
0.13 http://www.ivoa.net/internal/IVOA/IvoaGridAndWebServices /VOStore0.13.pdf

Author(s):

Matthew Graham (editor for this version)
Paul Harrison
Dave Morris
Guy Rixon

Abstract

VOSpace is a SOAP interface for access to data stores.

This version applies the VOSpace concept to flat, unconnected data spaces.

Future versions of the specification will add extensions to support a hierarchical structure and links between the individual space services.

Status of this document

This is an IVOA Proposed Recommendation made available for public review. The first release of this document (Version 1.0) was 2007 June 26. The first release of the predecessor Working Draft (Version 1.0) was 2007 March 4.

This is an IVOA Proposed Recommendation made available for public review. It is appropriate to reference this document only as a recommended standard that is under review and which may be changed before it is accept as a full recommendation.

A list of current IVOA Recommendations and other technical documents can be found at http://www.ivoa.net/Documents/.

Acknowledgements

This document derives from discussions among the Grid and Web Services working group of the IVOA. It is particularly informed by prototypes built by Matthew Graham (Caltech/NVO), Paul Harrison (ESO/EuroVO) and David Morris (Cambridge, Astrogrid).

Contents

Abstract....1
Status of this document....2
Acknowledgements....2
Contents....2
Introduction....4
VOSpace identifiers....6
VOSpace data model....8
  Nodes and node types....9
  Properties....9
    Property values....10
    Property identifiers....10
    Property descriptions....10
      UI display name....11
      Property editors....11
    Standard properties....11
  Views....11
    Example use cases....12
      Simple file store....12
      Database store....12
    View identifiers....13
    View descriptions....13
      UI display name....14
      MIME types....14
  Protocols....14
    Protocol identifiers....14
    Protocol descriptions....15
      UI display name....15
    Standard protocols....15

Custom protocols 15
SRB gateway 15
Local NFS transfers 16
Transfers 17
Synchronous transfers 17
Asynchronous transfers 17
Access control 18
Web service operations 19
Service meta data 19
getProtocols 19
Parameters 19
Returns 19
Faults 19
getViews 19
Parameters 19
Returns 19
Faults 20
getProperties 20
Parameters 20
Returns 20
Faults 20
Creating and manipulating data nodes 20
createNode 20
Parameters 20
Returns 21
Faults 21
deleteNode 21
Parameters 21
Returns 21
Faults 21
listNodes 21
Parameters 21
Returns 23
Faults 23
moveNode 23
Parameters 23
Returns 23
Faults 23
copyNode 24
Parameters 24
Returns 24
Faults 24
Accessing meta data 24
getNode 24
Parameters 24
Returns 25
Faults 25
setNode 25
Parameters 25
Returns 25

Faults....25
pushDataToVoSpace....25
Parameters....26
Returns....26
Faults....26
pullDataToVoSpace....26
Parameters....27
Returns....27
Faults....27
Notes....27
pullDataFromVoSpace....27
Parameters....27
Returns....28
Faults....28
Notes....28
pushDataFromVoSpace....28
Parameters....28
Returns....28
Faults....29
Notes....29
Fault arguments....29
InternalFault....29
PermissionDenied....29
InvalidURI....29
NodeNotFound....29
DuplicateNode....29
InvalidToken....29
InvalidArgument....29
TypeNotSupported....30
ViewNotSupported....30
InvalidData....30
References....31

Introduction

VOSpace is an interface standard for data stores. It specifies how VO agents and applications can use network attached data stores to persist and exchange data in a standard way.

A VOSpace web service is an access point for a distributed storage network. Through this access point, a client can :

- add or delete data objects
- manipulate meta data for the data objects
- obtain URIs through which the content of the data objects can be accessed

VOSpace does not define how the data is stored or transferred, only the control messages to gain access. Thus, the VOSpace interface can readily be added to an existing storage system.

When we speak of “a VOSpace”, we mean the arrangement of data accessible through one particular VOSpace service. A VOSpace node represents a data object within a

VOSpace service.

Nodes in VOSpace have unique identifiers expressed as URIs in the '`vos://`' scheme, as defined below.

In this version of the standard, each VOSpace service provides a single, flat set of data objects, similar to a service described by the earlier VOStore standard; this version of the VOSpace specification supersedes VOStore.

Future versions of the VOSpace specification may provide support for a hierarchical arrangement of objects within a space, and may provide support for VOSpace services to be linked such that a client can navigate them as one global space.

Services implementing the current version of the specification can be linked in as leaf nodes of this global space without needing to change.

VOSpace identifiers

The identifier for a node in VOSpace shall be a URI with the scheme `vos://`.

Such a URI shall have the following parts with the meanings and encoding rules defined in RFC2396 [2].

- scheme
- naming authority
- path
- (optional) query
- (optional) fragment identifier

The naming authority for a VOSpace node shall be the VOSpace service through which the node was created. The authority part of the URI shall be constructed from the IVO registry identifier [2] for that service by deleting the `ivo://` prefix and changing all forward-slash characters('/') in the resource key to exclamation marks ('!').

This is an example of a possible VOSpace identifier.

```
vos://org.astrogrid.cam!vospace!container-6/siap-out-1.vot?foo=bar#baz
```

- The URI scheme is `vos://`

Using a separate URI scheme for VOSpace identifiers enables clients to distinguish between IVO registry identifiers and VOSpace identifiers.

- `org.astrogrid.cam!vospace!container-6`

is the authority part of the URI, corresponding to the IVO registry identifier

- `ivo://org.astrogrid.cam/vospace/container-6`

This is the IVO registry identifier of the VOSpace service that contains the node.

- `/siap-out-1.vot` is the URI path

Slashes in the path imply a hierarchical arrangement of data, as is normal with URIs. Since the current version of the specification does not support data hierarchies, an identifier for a node in a current service must have one slash at the start of the path and no other slashes.

- `?foo=baz` is a query string and thus is something to which the VOSpace service is supposed to respond

No queries of this nature are defined in the current version of the specification, but the query string system is reserved for use in later versions of the VOSpace specification.

- `#baz` is a fragment identifier. Its meaning attaches to the data returned from the VOSpace node, not to the node itself.

The fragment identifier should be interpreted by the client, not by the VOSpace service; the service shall ignore any fragment identifiers.

A VOSpace identifier is globally unique, and identifies one specific node in a specific VOSpace service.

A client should use the following procedure to resolve access to a VOSpace node from a VOSpace identifier:

- Extract the authority part of the VOSpace URI
- Convert the authority back to the IVO registry identifier of the VOSpace service by changing any '!' characters to '/' and adding the `ivo://` prefix
- Resolve the IVO registry identifier to an endpoint for the VOSpace service using the IVO resource registry
- Access the node via the endpoint using one of the web service methods defined in this standard

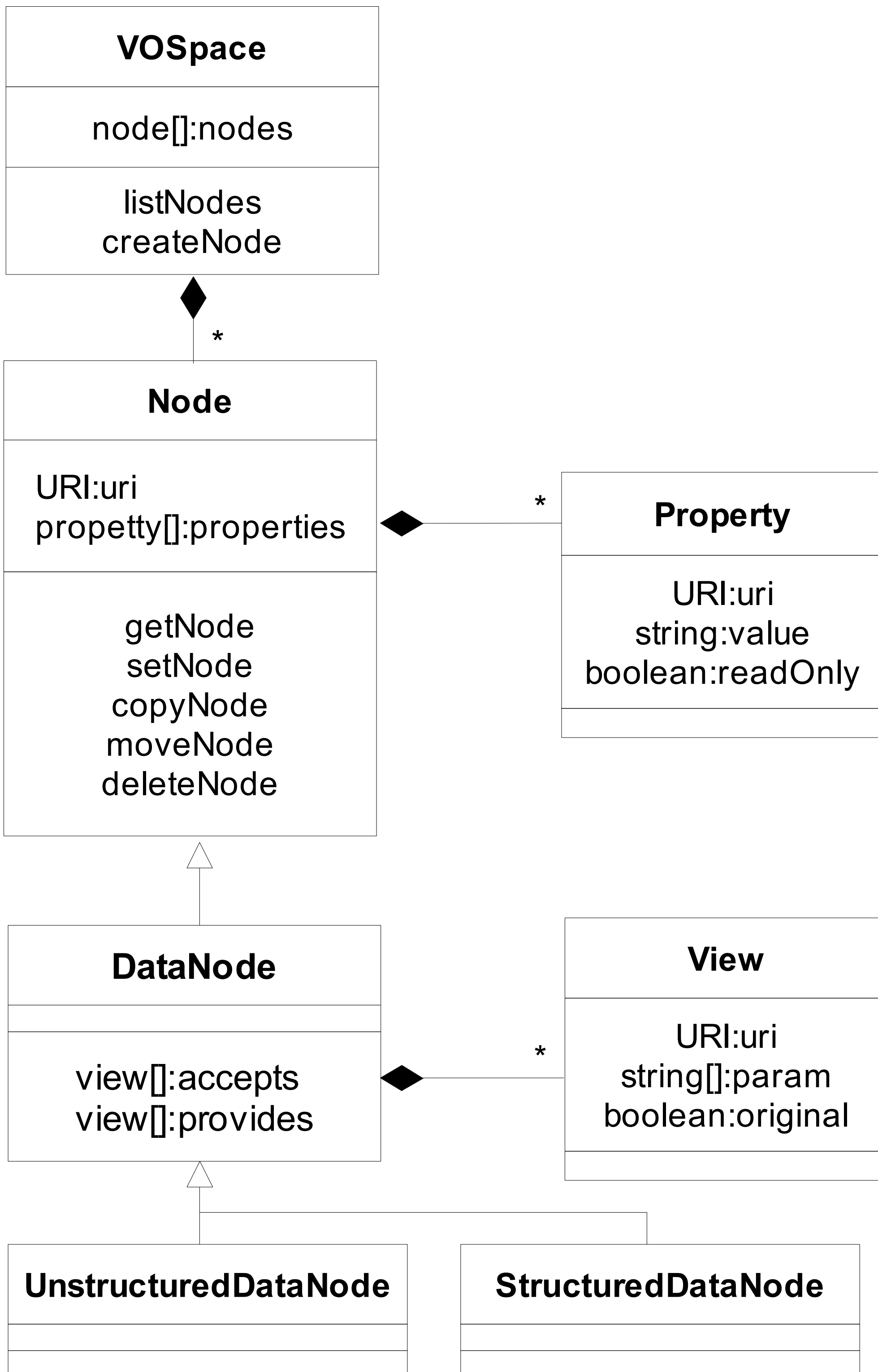
Given the example identifier

`vos://org.astrogrid.cam!vospace!container-6/siap-out-1.vot?foo=bar#baz`

processing the URI to resolve the VOSpace service would involve :

- Extract the authority part of the VOSpace URI
  - `org.astrogrid.cam!vospace!container-6`
- Convert the authority back to the IVO registry identifier of the VOSpace service by changing any '!' characters to '/'
  - `org.astrogrid.cam/vospace/container-6`
- Adding the *`ivo://`* prefix
  - `ivo://org.astrogrid.cam/vospace/container-6`
- Using the IVO registry to resolve the VOSpace service endpoint from the IVO identifier

VOSpace data model



Nodes and node types

The type of a VOSpace node determines how the VOSpace service stores and interprets the node data.

The types are arranged in a hierarchy, with more detailed types inheriting the structure of more generic types.

The following types are defined:

- *Node* is the most basic type
- *DataNode* describes a data item stored in the VOSpace
- *UnstructuredDataNode* describes a data item for which the VOSpace does not understand the data format

When data is stored and retrieved from an *UnstructuredDataNode* , the bit pattern read back shall be identical to that written

- *StructuredDataNode* describes a data item for which the space understands the format and may make transformations that preserve the meaning of the data.

When data is stored and retrieved from a *StructuredDataNode*, the bit pattern returned may be different to the original. For example, storing tabular data from a VOTable file will preserve the tabular data, but any comments in the original XML file may be lost.

A *Node* has the following elements:

- *uri* : the `vos://` identifier for the node , URI-encoded according to RFC2396 [2]
- *properties* : a set of meta data properties for the node

A *DataNode* has the following elements:

- *accepts* : a list of the views (data formats) that the node can accept
- *provides* : a list of the views (data formats) that the node can provide
- *busy* : a boolean flag to indicate that the data associated with the node cannot be accessed

The *busy* flag is used to indicate that an internal operation is in progress, and the node data is not available.

In the current version of the specification, all nodes are either structured or unstructured data nodes.

Future versions of the specification may introduce new types of nodes.

The set of node types defined by this standard is closed; new types may be introduced only via new versions of the standard.

To comply with the standard, a client or service must be able to parse XML representations of all the node types defined in the current specification.

Note - This does not require all services to support all of the *Node* types, just that it can process an XML request containing any of the types. If the service receives a request for a type that it does not support, the service should return a *TypeNotSupported* fault. The service must not throw an XML parser error if it receives a request for a type that it does not support.

Properties

Properties are simple string based meta data properties associated with a node.

Individual *Properties* should contain simple short string values, not large blocks of information. If a system needs to attach a large amount of meta data to a node, then it should either use multiple small *Properties,* or a single *Property* containing a URI or URL pointing to an external resource that contains the additional meta data.

A *Property* has the following elements:

- *uri* : the *Property* identifier
- *value* : the string value of the *Property*
- *readOnly* : a boolean flag to indicate that the *Property* cannot be changed by the client

Properties may be set by the client or the service

Property values

Unless they have special meaning to the service or client, *Properties* are treated as simple string values.

Some *Properties* may have meaning to the service. others may have meaning only to one specific type of client. A service implementation does not need to understand the meaning of all the *Properties* of a node, any *Properties* that it does not understand can simply be stored as text strings.

Property identifiers

Every new type of *Property* requires a unique URI to identify the *Property* and its meaning.

The rules for the *Property* identifiers are similar to the rules for namespace URIs in XML schema. The only restriction is that it must be a valid (unique) URI.

- An XML schema namespace identifier can be just a simple URN, e.g. `urn:my-namespace`
- Within the IVOA, the convention for namespace identifiers is to use a HTTP URL pointing to the namespace schema or a resource describing it

The current VOSpace schema defines *Property* identifiers as *anyURI*. The only restriction is that it must be a valid (unique) URI.

- A *Property* URI can be a simple URN, e.g. `urn:my-property`

This may be sufficient for testing and development on a private system, but it is not scalable for use on a public service.

For a production system, any new *Properties* should have unique URIs that can be resolved into to a description of the *Property.*

Ideally, these should be IVO registry URIs that point to a description registered in the IVO registry :

- `ivo://my-registry/vospace/properties/my-property`

Using an IVO registry URI to identify *Properties* has two main advantages :

- IVO registry URIs are by their nature unique, which makes it easy to ensure that different teams do not accidentally use the same URI
- If the IVO registry URI points to a description registered in the IVO registry, this provides a mechanism to discover what the *Property* means

Property descriptions

If the URI for a particular *Property* is resolvable, i.e. an IVO registry identifier or a HTTP URL, then it should point to an XML resource that describes the *Property.*

A *Property* description should describe the data type and meaning of a *Property*.

A *PropertyDescription* should have the following members :

- *uri* : the formal URI of the *Property*
- *DisplayName* : A display name for the *Property*
- *Description* : A text block describing the meaning and validation rules of the *Property*

A *PropertyDescription* may have the following optional members :

- *UCD* : the Universal Content Descriptor (in the UCD1+ scheme) for the *Property*
- *Unit* : the unit of measurement of the *Property*

The information in a *Property* description can be used to generate a UI for displaying and modifying the different types of *Properties.*

Note that at the time of writing, the schema for registering *PropertyDescriptions* in the IVO registry has not been finalized.

UI display name

If a client is unable to resolve a *Property* identifier into a description, then it may just display the identifier as a text string :

- `urn:modified-date`

If the client can resolve the *Property* identifier into a description, then the client may use the information in the description to display a human readable name and description of the *Property :*

- *Last modification date of the node data*

Property editors

If the client is unable to resolve a *Property* identifier into a description, or does not understand the type information defined in the description, then the client may treat the *Property* value as a simple text string.

If the client can resolve the *Property* identifier into a description, then the client may use the information in the description to display an appropriate editing tool for the *Property.*

In the current version of the specification the rules for editing *Properties* are as follows :

- A service may impose validation rules on the values of specific types of *Properties*
- If a client attempts to set a *Property* to an invalid value, then the service may reject

the change
- Where possible, the validation rules for a type of *Property* should be defined in the *Property* description

Future versions of the VOSpace specification may extend the *PropertyDescription* to include more specific machine readable validation rules for a *Property* type.

Note that at the time of writing, the schema for registering validation rules in *PropertyDescriptions* has not been finalized.

Standard properties

The VOSpace team intend to register *Property* URIs and *PropertyDescriptions* for the core set of *Properties*.

However, this is not intended to be a closed list, different implementations are free to define and use their own *Properties*.

Views

A *View* describes the data formats and contents available for importing or exporting data to or from a VOSpace node.

The meta data for each VOSpace *Node* contains two lists of *Views.*

- *accepts* is a list of *Views* that the service can accept for importing data into the *Node*
- *provides* is a list of *Views* that the service can provide for exporting data from *Node*

A *View* has the following members :

- *uri* : the *View* identifier
- *original* : an optional boolean flag to indicate that the *View* preserves the original bit pattern of the data
- *param* : a set of name-value pairs that can be used to specify additional arguments for the *View*

Example use cases

Simple file store

A simple VOSpace system that stores data as a binary files can just return the contents of the original file. The client supplies a *View* identifier when it imports the data, and the service uses this information to describe the data to other clients.

A file based system can use the special case identifier `'ivo://net.ivoa.vospace/views/any'` to indicate that it will accept any data format or *View* for a *Node.*

For example :

- A client imports a file into the service, specifying a *View* to describe the file contents
- The service stores the data as a binary file and keeps a record of the *View*
- The service can then use the *View* supplied by the client to describe the data to

other clients

This type of service is not required to understand the imported data, or to verify that it contents match the *View,* it treats all data as binary files.

Database store

A VOSpace system that stores data in database tables would need to be able to understand the data format of an imported file in order to parse the data and store it correctly. This means that the service can only accept a specific set of *Views,* or data formats, for importing data into the *Node*.

In order to tell the client what input data formats it can accept, the service publishes a list of specific *Views* in the *accepts* list for each *Node.*

On the output side, a database system would not be able to provide access to the original input file. The contents of file would have been transferred into the database table and then discarded. The system has to generate the output results from the contents of the database table.

In order to support this, the service needs to be able to tell the client what *Views* of the data are available.

A database system may offer access to the table contents as either VOTable or FITS files, it may also offer zip or tar.gz compressed versions of these. In which case the system needs to be able to express nested file formats such as 'zip containing VOTable' and 'tar.gz containing FITS'.

A service may also offer subsets of the data. For example, a work flow system may only want to look at the table headers to decide what steps are required to process the data. If the table contains a large quantity of data, then downloading the whole contents just to look at the header information is inefficient. To make this easier, a database system may offer a 'meta data only' *View* of the table, returning a VOTable or FITS file containing just the meta data headers and no rows.

So our example service may want to offer the following *Views* of a database table :

- Table contents as FITS
- Table contents as VOTable

- Table contents as zip containing FITS
- Table contents as zip containing VOTable

- Table contents as tar.gz containing FITS
- Table contents as tar.gz containing VOTable

- Table meta data as FITS
- Table meta data as VOTable

The service would publish this information as a list of *Views* in the *provides* section of the meta data for each *Node.*

The VOSpace specification does not mandate what *Views* a service must provide. The VOSpace specification is intended to provide a flexible mechanism enabling services to describe a variety of different *Views* of data. It is up to the service implementation to decide what *Views* of the data it can accept and provide.

View identifiers

Every new type of *View* requires a unique URI to identify the *View* and its content.

The rules for the *View* identifiers are similar to the rules for namespace URIs in XML schema. The only restriction is that it must be a valid (unique) URI.

- An XML schema namespace identifier can be just a simple URN, e.g. `urn:my-namespace`
- Within the IVOA, the convention for namespace identifiers is to use a HTTP URL pointing to the namespace schema, or a resource describing it

The current VOSpace schema defines *View* identifiers as *anyURI*. The only restriction is that it must be a valid (unique) URI.

- A *View* URI can be a simple URN, e.g. `urn:my-view`

This may be sufficient for testing and development on a private system, but it is not scalable for use on a public service.

For a production system, any new *Views* should have unique URIs that can be resolved into to a description of the *View.*

Ideally, these should be IVO registry URIs that point to a description registered in the IVO registry :

- `ivo://my-registry/vospace/views/my-view`

Using an IVO registry URI to identify *Views* has two main advantages :

- IVO registry URIs are by their nature unique, which makes it easy to ensure that different teams do not accidentally use the same URI
- If the IVO registry URI points to a description registered in the IVO registry, this provides a mechanism to discover what the *View* contains

View descriptions

If the URI for a particular *View* is resolvable, i.e. an IVO registry identifier or a HTTP URL, then it should point to an XML resource that describes the *View.*

A *ViewDescription* should describe the data format and/or content of the view.

A *ViewDescription* should have the following members :

- *uri* : the formal URI of the *View*
- *DisplayName* : A simple text display name of the *View*
- *Description* : Text block describing the data format and content of the *View*

A *ViewDescription* may have the following optional members :

- *MimeType* : the standard MIME type of the *View,* if applicable

However, at the time of writing, the schema for registering *ViewDescriptions* in the IVO registry has not been finalized.

UI display name

If a client is unable to resolve a *View* identifier into a description, then it may just display

the identifier as a text string :

- *Download as* `urn:table.meta.fits`

If the client can resolve the *View* identifier into a description, then the client may use the information in the description to display a human readable name and description of the *View* :

- *Download table meta data as FITS header*

MIME types

If a VOSpace service provides HTTP access to the data contained in a *Node*, then if the *ViewDescription* contains a *MimeType* field, this should be included in the appropriate header field of the HTTP response.

Protocols

A *Protocol* describes the parameters required to perform a data transfer using a particular protocol.

A *Protocol* has the following members :

- *uri* : the *Protocol* identifier
- *endpoint* : the endpoint URL to use for the data transfer
- *param* : A list of name-value pairs that specify any additional arguments required for the transfer

Protocol identifiers

Every new type of *Protocol* requires a unique URI to identify the *Protocol* and how to use it.

The rules for the *Protocol* identifiers are similar to the rules for namespace URIs in XML schema. The only restriction is that it must be a valid (unique) URI

- An XML schema namespace identifier can be just a simple URN, e.g. `urn:my-namespace`
- Within the IVOA, the convention for namespace identifiers is to use a HTTP URL pointing to the namespace schema, or a resource describing it

The current VOSpace schema defines *Protocol* identifiers as *anyURI*. The only restriction is that it must be a valid (unique) URI.

- A *Protocol* URI can be a simple URN, e.g. `urn:my-protocol`

This may be sufficient for testing and development on a private system, but it is not scalable for use on a public service.

For a production system, any new *Protocols* should have unique URIs that can be resolved into to a description of the *Protocol.*

Ideally, these should be IVO registry URIs that point to a description registered in the IVO registry :

- `ivo://my-registry/vospace/protocols/my-protocol`

Using an IVO registry URI to identify *Protocols* has two main advantages :

- IVO registry URIs are by their nature unique, which makes it easy to ensure that different teams do not accidentally use the same URI
- If the IVO registry URI points to a description registered in the IVO registry, this provides a mechanism to discover how to use the *Protocol*

Protocol descriptions

If the URI for a particular *Protocol* is resolvable, i.e. an IVO registry identifier or a HTTP URL, then it should point to an XML resource that describes the *Protocol.*

A *ProtocolDescription* should describe the underlying transport protocol, and how it should be used in this context.

A *ProtocolDescription* should have the following members :

- *uri* : the formal URI of the *Protocol*
- *DisplayName* : A simple display name of the *Protocol*
- *Description* : Text block describing describing the underlying transport protocol, and how it should be used in this context

However, at the time of writing, the schema for registering *ProtocolDescriptions* in the IVO registry has not been finalized.

UI display name

If a client is unable to resolve a *Protocol* identifier into a description, then it may just display the identifier as a text string :

- *Download using* `urn:my-protocol`

If the client can resolve the *Protocol* identifier into a description, then the client may use the information in the description to display a human readable name and description of the *Protocol :*

- *Download using standard HTTP GET*

Standard protocols

The VOSpace team intend to register *Protocol* URIs and *ProtocolDescriptions* for the core set of standard transport protocols.

e.g.

- Standard HTTP get and put
- Standard FTP get and put

However, this is not intended to be a closed list, different implementations are free to define and use their own transfer *Protocols.*

Custom protocols

There are several use cases where a specific VOSpace implementation may want to define and use a custom VOSpace transfer *Protocol,* either extending an existing *Protocol,* or defining a new one.

SRB gateway

One example would be a VOSpace service that was integrated with a SRB system. In order to enable the service to use the native SRB transport protocol to transfer data, the service providers would need to register a new *ProtocolDescription* to represent the SRB transport protocol.

The *ProtocolDescription* would refer to the technical specification for the SRB transport protocol, and define how it should be used to transfer data to and from the VOSpace service.

Clients that do not understand the SRB transport protocol would not recognize the URI for the SRB *Protocol,* and would ignore *Transfer* options offered using this *Protocol.*

Clients that were able to understand the SRB transport protocol would recognize the URI for the SRB *Protocol,* and could use the '`srb://..`' endpoint address in a *Protocol* option to transfer data using the SRB transport protocol.

Enabling different groups to define, register and use their own custom *Protocols* in this way means that support for new transport protocols can be added to VOSpace systems without requiring changes to the core VOSpace specification.

In this particular example, it is expected that one group within the IVOA will work with the SRB team at SDSC to define and register the *Protocol* URI and *ProtocolDescription* for using the SRB protocol to transfer data to and from VOSpace systems.

Other implementations that plan to use the SRB transport protocol in the same way could use the same *Protocol* URI and *ProtocolDescription* to describe data transfers using the SRB transport protocol.

The two implementations would then be able use the common *Protocol* URI to negotiate data transfers using the SRB transport protocol.

Local NFS transfers

Another example of a custom *Protocol* use case would to transfer data using the local NFS file system within an institute.

If an institute has one or more VOSpace services co-located with a number of data processing applications, all located within the same local network, then it would be inefficient to use HTTP get and put to transfer the data between the services if they could all access the same local file system.

In this case, the local system administrators could register a custom *ProtocolDescription* which described how to transfer data using their local NFS file system.

- `ivo://my.institute/vospace/protocols/internal-nfs`

Data transfers using this *Protocol* would be done using `file://` URLs pointing to locations within the local NFS file system :

- `file:///net/host/path/file`

These URLs would only have meaning to services and applications located within the local network, and would not be usable from outside the network.

Services and applications located within the local network would be configured to recognize the custom *Protocol* URI, and to use local file system operations to move files within the NFS file system.

Services and applications located outside local network would not recognize the custom *Protocol* URI and so would not attempt to use the internal file URLs to transfer data.

Note that in this example the custom *Protocol* URI and the associated *ProtocolDescription* refer to data transfers using file URLs **within a specific local NFS file system.**

If a different institute wanted to use a similar system to transfer data within their own local network, then they would have to register their own custom *Protocol* URI and associated *ProtocolDescription.*

The two different *Protocol* URIs and *ProtocolDescriptions* describe how to use the same underlying transport protocol (NFS) in different contexts.

Enabling different groups to define, register and use their own custom *Protocols* in this way means that systems can be configured to use the best available transport protocols for transferring data, without conflicting with other systems who may be using similar a transport protocol in a different context.

Transfers

A *Transfer* describes the details of a data transfer to or from a space.

A *Transfer* has the following members :

- *view* : A *View* specifying the requested *View*
  - For the transfer to be valid, the specified *View* must match one of those listed in the *accepts* or *provides* list of the *Node*
- *protocol* : one or more a *Protocols* specifying the transfer protocols to use
  - A *Transfer* may contain more than one *Protocol* element with different *Protocol* URIs
  - A *Transfer* may contain more than one *Protocol* element with the same *Protocol* URI with different endpoints

Synchronous transfers

Two of the VOSpace transfer methods are synchronous - the service performs the data transfer during the scope of the SOAP call.

In these methods, the client constructs a *Transfer* request containing details of the *Node* and *View* and one or more *Protocol* elements with valid endpoint addresses.

The service may ignore *Protocols* with URIs that it does not recognize.

If the server is unable to handle any of the requested *Protocols* in a *Transfer* request, then it responds with a fault.

The order of the *Protocols* in the request indicates the order of preference that the client

would like the server to use However, this is only a suggestion, and the server is free to use its own preferences to select which *Protocol* it uses first.

The service selects the first *Protocol* it wants to use from the list and attempts to transfer the data using the *Protocol* endpoint.

If the first attempt fails, the server may choose another *Protocol* from the list and re-try the transfer using that *Protocol* endpoint.

The server may attempt to transfer the data using any or all of the *Protocols* in the list until either, the data transfer succeeds, or there are no more *Protocol* options left.

The server is only allowed to use each *Protocol* option once. This allows a data source to issue one time URLs for a *Transfer,* and cancel each URL once it has been used.

Once one of the *Protocol* options succeeds the transfer is considered to be completed, and the server is not allowed to use any of the remaining *Protocol* options. This allows a data source to issue a set of one time URLs for a transfer, and to cancel any unused URLs once the transfer has been completed.

Some *Protocols* may require the service to call a callback address when a data transfer completes. This behavior is specific to the *Protocol,* and should be defined in the *ProtocolDescription.*

If none of the *Protocol* options succeed, then the transfer is considered to have failed, and the service returns an exception containing details of the *Protocol* options it tried.

Asynchronous transfers

Two of the VOSpace transfer methods are asynchronous - an external actor performs the data transfer outside the scope of the SOAP call.

In these methods, the client sends a template *Transfer* request to the server.

The *Transfer* request should contain details of the *Node* and *View* and one or more *Protocol* elements without endpoint addresses.

In effect, the client is sending a list of *Protocols* that it (the client) wants to use for the transfer.

The service may ignore *Protocols* with URIs that it does not recognize.

The service selects the *Protocols* from the request that it is capable of handling, and builds a *Transfer* response containing the selected *Protocol* elements filling in valid endpoint addresses for each of them.

The order of the *Protocol* elements in the request indicates the order of preference that the client would like to use. However, this is only a suggestion, and the server is free to use its own preferences when generating the list of *Protocols* in the response.

In effect, the server is responding with the subset of the requested *Protocols* that it (the server) is prepared to offer.

If the server is unable to accept any of the requested *Protocols,* then it responds with a fault.

On receipt of the response, the client can use the list of *Protocols* itself, or pass them on to another agent to perform the data transfer on its behalf.

The agent may ignore *Protocols* URIs that it does not recognize.

The agent selects the first *Protocol* it wants to use from the list and attempts to transfer the data using the *Protocol* endpoint.

If the first attempt fails, the agent may choose another *Protocol* from the list and re-try the transfer using that *Protocol* endpoint.

The agent may attempt to transfer the data using any or all of the *Protocols* in the list until either, the data transfer succeeds, or there are no more *Protocol* options left.

The agent is only allowed to use each *Protocol* option once. This allows a data source to issue one time URLs for a *Transfer,* and cancel each URL once it has been used.

Once one of the *Protocol* options succeeds the transfer is considered to be completed, an the agent is not allowed to use any of the remaining unused *Protocol* options. This allows a data source to issue a set of one time URLs for a transfer, and to cancel any unused URLs once the transfer has been completed.

Some *Protocols* may require the agent to call a callback address when a data transfer completes. This behavior is specific to the *Protocol,* and should be defined in the *ProtocolDescription.*

If none of the *Protocol* options succeed, then the transfer is considered to have failed.

Access control

The access control policy for a VOSpace is defined by the implementor of that space according to the use cases for which the implementation is intended.

A human-readable description of the implemented access policy must be declared in the registry meta data for the space.

These are the most probable access policies :

- No access control is asserted. Any client can create, read, write and delete nodes anonymously
- No authorization is required, but clients must authenticate an identity (for logging purposes) in each request to the space
- Clients may not create or change nodes (i.e. the contents of the space are fixed by the implementation or set by some interface other than VOSpace), but any client can read any node without authentication
- Nodes are considered to be owned by the user who created them. Only the owner can operate on a node

No operations to modify the access policy (e.g. to set permissions on an individual node) are included in this standard. Later versions may add such operations.

Where the access policy requires authentication of callers, the VOSpace service shall support one of the authentication methods defined in the IVOA Single Sign On profile.

Web service operations

A VOSpace-1.0 service shall be a SOAP service with the following operations.

Service meta data

getProtocols

Get a list of the transfer *Protocols* supported by the space service.

Parameters

- none

Returns

- *accepts* : A list of *Protocols* that the service can accept
  - In this context 'accepting a protocol' means that the service can act as a client for the protocol
  - e.g. 'accepting http-get' means the service can read data from an external http web server
- *provides* : A list of *Protocols* that the service can provide
  - In this context 'providing a protocol' means that the service can act as a server for the protocol
  - e.g. 'providing http-get' means the service can act as a http web server

Faults

- The service shall throw an *InternalFault* exception if the operation fails

getViews

Get a list of the *Views* and data formats supported by the space service.

Parameters

- none

Returns

- *accepts* : A list of *Views* that the service can accept
  - In this context 'accepting a view' means that the service can receive input data in this format
  - A simple file based system may use the reserved *View* URI `ivo://net.ivoa.vospace/views/any` to indicate that it can accept data in any format
- *provides* : A list of *Views* that the service can provide
  - In this context 'providing a view' means that the service can produce output data in this format
  - A simple file based system may use the reserved *View* URI `ivo://net.ivoa.vospace/views/any` to indicate that it can provide data in any format

Faults

- The service shall throw an *InternalFault* exception if the operation fails

getProperties

Parameters

- none

Returns

- *accepts* : A list of identifiers for the *Properties* that the service accepts and understands
- *provides* : A list of identifiers for the *Properties* that the service provides
- *contains* : A list of identifiers for all the *Properties* currently used by *Nodes* within the service

Faults

- The service shall throw an *InternalFault* exception if the operation fails

Creating and manipulating data nodes

createNode

Create a new node at a specified location.

Parameters

- *node* : A template *Node* for the node to be created

A valid *uri* attribute is required. If the reserved URI `vos://null` is used the service will replace it with a new unique service-generated URI.

If the *Node* `xsi:type` is not specified then a generic node of type *Node* is implied.

The permitted values of `xsi:type` are :

- `vos:Node`
- `vos:DataNode`
- `vos:UnstructuredDataNode`
- `vos:StructuredDataNode`

When creating a new *Node* the service may substitute a valid subtype, i.e. If `xsi:type` is set to `vos:DataNode` then this may be implemented as a *DataNode, StructuredDataNode* or an *UnstructuredDataNode.*

The properties of the new *Node* can be set by adding *Properties* to the template. Attempting to set a *Property* that the service considers to be 'readonly' will cause a *PermissionDenied* exception.

The *accepts* and *provides* lists of *Views* for the *Node* cannot be set using this method.

Returns

- *node* : details of the new *Node*

The *accepts* list of *Views* for the *Node* will be filled in by the service based on service capabilities.

The *provides* list of *Views* for the *Node* may not be filled in until some data has been imported into the *Node*.

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *DuplicateNode* exception if a *Node* already exists with the same URI
- The service shall throw an *InvalidURI* exception if the requested URI is invalid
- The service shall throw a *TypeNotSupported* exception if the type specified in `xsi:type` is not supported
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation

deleteNode

Delete a node.

Parameters

- *target* : the URI of an existing *Node*

Returns

- *void*

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw a *NodeNotFound exception* if the target node does not exist

listNodes

List nodes in the space.

In order to support large numbers of nodes, this method uses a continuation token to enable the list of results to be split across more than response.

Parameters

- *token* : An optional continuation token from a previous request
  - No token indicates a request for a new list

The server may impose a limited lifetime on the continuation token. If a token has expired,

the server will throw an exception, and the client will have to make a new request.

- *limit* : An optional limit indicating the maximum number of results in the response
  - No limit indicates a request for an unpaged list. However the server may still impose its own limit on the size of an individual response, splitting the results into more than one page if required

- *detail* : The level of detail in the returned listing
  - `min` : The list contains the minimum detail for each *Node* with all optional parts removed
    - e.g. `<node uri="vos://service/name"/>`
  - `max` : The list contains the maximum detail for each *Node,* including any `xsi:type` specific extensions
  - `properties` : The list contains a basic *node* element with a list of *properties* for each *Node* with no `xsi:type` specific extensions.

- *nodes* : A list containing zero or more *Nodes* identifying the target URIs to be listed

Only the *uri* of the *Nodes* in the request list are used, the service will ignore any other elements or attributes. This method does not perform a search based on the *Node Properties* or other attributes.

The *Node* URIs in the request list may contain a '*' wild card character in the *name* part of the URI (the remaining text following the last '/' character).

A single request may include more than one target *Node* containing a wild card. For example, the following request lists all *Nodes* with names that that match either '`*.xml`' or '*.txt'

```
<listNodes>
    <nodes>
        <node uri="vos://service/*.xml"/>
        <node uri="vos://service/*.txt"/>
    </nodes>
</listNodes>
```

Note that the wild card substitution for the '`*`' is a simple 'zero or more of any characters' match.

So a request for '`*.txt`' will match *Nodes* with the the following names :

- `.txt`
- `frog.txt`
- `.txtinfo`
- `frog.txtinfo`

This method does not support regular expression matches.

An empty list of target *Nodes* list implies a full listing of the space.

A request with an empty list of target *Nodes* :

```
<listNodes>
    <nodes/>
</listNodes>
```

is equivalent to a request with a single target *Node* URI of '*' :

```
<listNodes>
    <nodes>
        <node uri="vos://service/*"/>
    </nodes>
</listNodes>
```

Returns

- *token* : An optional continuation token, indicating that the list is incomplete
  - The client may use this token to request the next block of *Nodes* in the sequence
  - No token indicates that the list is complete.
- *limit* : An optional limit which must be present if a limit parameter was used in the request
  - If present, the value is the value from the original request and not any limit imposed by the service
- *nodes* : A list of the *Nodes* matching the requested *Node* URIs

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw a *NodeNotFound* exception if a specific target *Node* does not exist
  - This does not apply if the target *Node* URI contains a wild card
- The service shall throw an *InvalidToken* exception if it does not recognize the continuation token
- The service shall throw an *InvalidToken* exception if the continuation token has expired

moveNode

Move a node within a VOSpace service.

Note that this does not cover moving data between two separate VOSpace services.

Moving nodes between separate VOSpace services should be handled by the client, using the import, export and delete methods.

Parameters

- *source* : The URI of an existing *Node*
- *destination* : A template describing the new *Node.*

If the *destination uri* is set to the reserved URI `vos://null` then the service will generate a new unique URI for the Node.

The *Properties* from the *source Node* will be combined with the *Properties* from the *destination* to create the new *Node.*

The *Node* type cannot be changed using this method.

The value of the `xsi:type` attribute on the *destination* will be ignored and the new *Node* will inherit the same type as the original.

The *accepts* and *provides* lists of *Views* for the new *Node* cannot be set using this method.

Returns

- *node* : Details of the *Node* in its new location

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw a *NodeNotFound* exception if the source *Node* does not exist
- The service shall throw a *DuplicateNode* exception if a *Node* already exists at the destination
- The service shall throw an *InvalidURI* exception if a specified URI is invalid
- The service shall throw an *InvalidArgument* exception if a specified value is invalid

copyNode

Copy a node within a VOSpace service.

Note that this does not cover copying data between two separate VOSpace services.

Copying nodes between separate VOSpace services should be handled by the client, using the import and export methods.

Parameters

- *source* : The URI of an existing *Node*
- *destination* : A template *Node* describing the new *Node*

If the *destination uri* is set to the reserved URI `vos://null` then the service will generate a new unique URI for the *Node.*

The *Properties* from the *source Node* will be combined with the *Properties* from request to create the new *Node.*

The *Node* type cannot be changed using this method.

The value of the `xsi:type` attribute on the *destination* will be ignored and the new *Node* will inherit the same type as the original.

The *accepts* and *provides* lists of *Views* for the new *Node* cannot be set using this method.

Returns

- *node* : Details of the new *Node*

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw a *NodeNotFound* exception if the source *Node* does not exist
- The service shall throw a *DuplicateNode* exception if a *Node* already exists at the destination
- The service shall throw an *InvalidURI* exception if a specified URI is invalid
- The service shall throw an *InvalidArgument* exception if a specified value is invalid

Accessing meta data

getNode

Get the details for a specific *Node.*

Parameters

- *target* : The URI of an existing *Node*

Returns

- *node* : Details of the *Node*

The full expanded record for the node is returned, including any `xsi:type` specific extensions.

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw a *NodeNotFound* exception if the target *Node* does not exist

setNode

Set the property values for a specific node.

Parameters

- *node* : A *Node* containing a the *Node uri* and a list of the *Properties* to set

This will add or update the node properties including any `xsi:type` specific elements.

The operation is the union of existing values and new ones.

- An empty value sets the value to blank.

- To delete a *Property,* set the `xsi:nill` attribute to **`true`**

This method cannot be used to modify the *Node* type.

This method cannot be used to modify the *accepts* or *provides* list of *Views* for the *Node*.

Returns

- *node* : Details of the *Node*

The full expanded record for the node is returned, including any `xsi:type` specific extensions.

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the requests attempts to modify a *readonly Property*
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw a *NodeNotFound* exception if the target *Node* does not exist
- The service shall throw an *InvalidArgument* exception if a specified *property* value is invalid

pushDataToVoSpace

Request a list of URLs to send data to a VOSpace node.

This method asks the server for a list of one or more URLs that the client can use to send data to.

The data transfer is initiated by the client, after it has received the response from the VOSpace service.

The primary use case for this method is client that wants to send some data directly to a VOSpace service.

Parameters

- *destination* : A description of the target *Node*

A valid *uri* attribute is required.

If a *Node* already exists at the target URI, then the data will be imported into the existing *Node* and the *Node properties* will be updated with the *Properties* in the request.

If there is no *Node* at the destination URI, then the service will create a new *Node* using the *uri,* `xsi:type` and *properties* supplied in the request (see *createNode* for details).

- *transfer* : A template for the data *Transfer*

The *Transfer* template contains details of the *View* and a list of the *Protocols* that the client wants to use.

The list of *Protocols* should not contain *endpoint* addresses, the service will supply the *endpoint* addresses in the response.

The service will ignore any of the requested protocols that it does not understand or is unable to support.

Returns

- *destination* : Updated details of the destination *Node*
- *transfer* : Updated details of the data *Transfer*

The service selects which of the requested *Protocols* it is willing to provide and fills in the operational details for each one.

The service response should not include any *Protocols* which it is unable to support.

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service will throw a *TypeNotSupported* exception if it it is unable to create a new *Node* of the requested type
- The service shall throw a *ViewNotSupported* exception if a *StructuredDataNode* is requested with no *View*
- The service shall throw a *ViewNotSupported* exception if the service does not support the the requested *View*
- The service shall throw a *ProtocolNotSupported* exception if it supports none of the requested *Protocols*
- The service shall throw an *InvalidURI* exception if the *destination* URI is invalid
- The service shall throw an *InvalidArgument* exception if a *View* parameter is invalid
- The service shall throw an *InvalidArgument* exception if a *Protocol* parameter is invalid

pullDataToVoSpace

Import data into a VOSpace node.

This method asks the server to fetch data from a remote location.

The data transfer is initiated by the VOSpace service and transferred direct into the target *Node.*

The data source can be another VOSpace service, or a standard HTTP or FTP server.

The primary use case for this method is transferring data from one server or service to another.

Parameters

- *destination* : A description of the target *Node*

A valid *uri* attribute is required.

If a *Node* already exists at the target URI, then the data will be imported into the existing *Node* and the *Node properties* will be updated with the *Properties* in the request.

If there is no *Node* at the destination URI, then the service will create a new *Node* using

the *uri,* `xsi:type` and *properties* supplied in the request (see *createNode* for details).

- *transfer* : Details of the *Transfer*

The *Transfer* details should include the *View* and a list of one or more *Protocols* with valid *endpoints* and parameters.

Returns

- *destination* : Updated details of the destination *Node*

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service will throw a *TypeNotSupported* exception if it is unable to create a new *Node* of the requested type
- The service shall throw a *ViewNotSupported* exception if a *StructuredDataNode* is requested with no *View*
- The service shall throw a *ViewNotSupported* exception if the service does not support the requested *View*
- The service shall throw a *ProtocolNotSupported* exception if it supports none of the requested *Protocols*
- The service shall throw an *InvalidURI* exception if the destination URI is invalid
- The service shall throw an *InvalidArgument* exception if a *View* parameter is invalid
- The service shall throw an *InvalidArgument* exception if a *Protocol* parameter is invalid
- The service shall throw a *TransferFailed* exception if the data transfer does not complete
- The service shall throw an *InvalidData* exception if the data does not match the *View*

Notes

In VOSpace version 1.0, the transfer is synchronous, and the SOAP call does not return until the transfer has been completed.

If the *Transfer* request contains more than one *Protocol* option, then the service may fail over to use one or more of the options if the first one fails. The service should try each *Protocol* option in turn until one succeeds or all have been tried.

pullDataFromVoSpace

Request set of URLs that the client can read data from.

The client requests access to the data in a *Node,* and the server responds with a set of URLs that the client can read the data from.

Parameters

- *source* : The URI of an existing *DataNode*

- *transfer* : A template for the *Transfer*

The template for the *Transfer* should contain details of the *View* and a list of the *Protocols* that the client would like to use.

The list of *Protocols* should not contain *endpoint* addresses, the service will supply the *endpoint* addresses in the response.

The service will ignore any of the requested protocols that it does not understand or is unable to support.

Returns

- *transfer* : Updated details of the data *Transfer*

The service selects which of the requested *Protocols* it is willing to provide and fills in the operational details for each one.

The service response should not include any *Protocols* which it is unable to support.

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw an *InvalidURI* exception if the *source* URI is invalid
- The service shall throw a *NodeNotFound* exception if the source *Node* does not exist.
- The service shall throw a *ProtocolNotSupported* exception if it none of the requested *Protocols* are supported
- The service shall throw a *ViewNotSupported* exception if it does not support the requested *View*
- The service shall throw an *InvalidArgument* exception if a *View* parameter is invalid
- The service shall throw an *InvalidArgument* exception if *Protocols* a parameter is invalid

Notes

The any endpoint URLs supplied in the response should be considered as 'one shot' URLs. A VOSpace service connected to a standard web server may return the public URL for the data. However, a different implementation may create a unique single use URL specifically for this transfer.

pushDataFromVoSpace

Ask the server to send data to a remote location.

The client supplies a list of URLs and asks the server to send the data to the remote location.

The transfer is initiated by the server, and the data is transferred direct from the server to the remote location.

Parameters

- *source* : The URI of an existing *DataNode*
- *transfer* : Details of the *Transfer*

The *Transfer* details should include the *View* and a list of one or more *Protocols* with valid *endpoint* and *parameters*.

Returns

- void

Faults

- The service shall throw an *InternalFault* exception if the operation fails
- The service shall throw a *PermissionDenied* exception if the user does not have permissions to perform the operation
- The service shall throw an *InvalidURI* exception if the *source* URI is invalid
- The service shall throw a *NodeNotFound* exception if the source *Node* does not exist
- The service shall throw a *ProtocolNotSupported* exception if it supports none of the requested *Protocols*
- The service shall throw a *ViewNotSupported* exception if it does not support the requested *View*
- The service shall throw an *InvalidArgument* exception if a *Protocol* parameter
- The service shall throw an *InvalidArgument* exception if a *Protocols* parameter is invalid
- The service shall throw a *TransferFailed* exception if the data transfer does not complete

Notes

In VOSpace version 1.0, the transfer is synchronous, and the SOAP call does not return until the transfer has been completed.

If the *Transfer* request contains more than one *Protocol* then the service may fail over to use one or more of the options if the first one fails. The service should try each *Protocol* option in turn until one succeeds or all have been tried.

Fault arguments

InternalFault

This is thrown with a description of the cause of the fault.

PermissionDenied

This is thrown with no arguments.

InvalidURI

This is thrown with details of the invalid URI.

NodeNotFound

This is thrown with the URI of the missing *Node.*

DuplicateNode

This is thrown with the URI of the duplicate *Node.*

InvalidToken

This is thrown with the invalid token.

InvalidArgument

This is thrown with a description of the invalid argument, including the View or Protocol URI and the name and value of the parameter that caused the fault.

TypeNotSupported

This is thrown with the *QName* of the unsupported type.

ViewNotSupported

This is thrown with the uri of the *View.*

InvalidData

This is thrown with any error message that the data parser produced.

References

[1] T. Berners-Lee, R. Fielding, U. Irvine, L. Masinter, Uniform Resource Identifiers (URI): Generic Syntax, http://www.faqs.org/rfcs/rfc2396.html

[2] R. Plante, T. Linde, R. Williams, & K. Noddle, IVOA Identifiers, http://www.ivoa.net/Documents/latest/IDs.html