



International

Virtual

Observatory

Alliance

VOSpace service specification

Version 1.10

IVOA Working Draft 2008 March 11

This version:

<http://www.ivoa.net/Documents/WD/GWS/VOSpace-20080311.doc>

Latest version:

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

Previous version(s):

REC 1.02 <http://www.ivoa.net/Documents/REC/GWS/VOSpace-20080124.pdf>

Author(s):

Matthew Graham
Dave Morris
Guy Rixon

Abstract

VOSpace is the IVOA interface to distributed storage. This version extends the existing VOSpace 1.0 specification to support containers, links between individual VOSpace instances, third party APIs, and a find mechanism.

Status of This Document

This is an IVOA Working Draft. The first release of this document was 2008 March 11.

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](http://www.ivoa.net/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) and David Morris (Cambridge, Astrogrid).

This document has been developed with support from the National Science Foundation’s Information Technology Research Program under Cooperative Agreement AST0122449 with the John Hopkins University, from the UK Science and Technology Facilities Council (STFC), and from the European Commission’s Sixth Framework Program via the Optical Infrared Coordination Network (OPTICON).

Conformance related definitions

The words “MUST”, “SHALL”, “SHOULD”, “MAY”, “RECOMMENDED”, and “OPTIONAL” (in upper or lower case) used in this document are to be interpreted as described in IETF standard, RFC 2119 [RFC 2119].

The **Virtual Observatory (VO)** is a general term for a collection of federated resources that can be used to conduct astronomical research, education, and outreach. The **International Virtual Observatory Alliance (IVOA)** is a global collaboration of separately funded projects to develop standards and infrastructure that enable VO applications. The International Virtual Observatory (IVO) application is an application that takes advantage of IVOA standards and infrastructure to provide some VO service.

Contents

| | | |
|-------|------------------------|---|
| 1 | Introduction | 3 |
| 1.1 | Document roadmap | 3 |
| 2 | VOSpace data model | 4 |
| 2.1 | ContainerNode | 5 |
| 2.1.1 | Container identifiers | 5 |
| 2.1.2 | Inheritable properties | 5 |
| 2.1.3 | Container views | 5 |
| 2.2 | LinkNode | 6 |
| 2.3 | Capabilities | 6 |
| 2.3.1 | Example use cases | 7 |

| | | |
|-------|--------------------------------------|----|
| 2.3.2 | Capability identifiers | 7 |
| 2.3.3 | Capability descriptions | 8 |
| 2.3.4 | UI display name | 8 |
| 2.3.5 | Standard capabilities | 8 |
| 3 | Web service operations | 9 |
| 3.1 | Service metadata | 9 |
| 3.1.1 | getProtocols | 9 |
| 3.1.2 | getViews | 9 |
| 3.1.3 | getProperties | 9 |
| 3.1.4 | getCapabilities | 9 |
| 3.2 | Creating and manipulating data nodes | 9 |
| 3.2.1 | createNode | 9 |
| 3.2.2 | deleteNode | 10 |
| 3.2.3 | listNodes | 10 |
| 3.2.4 | findNodes | 11 |
| 3.2.5 | moveNode | 12 |
| 3.2.6 | copyNode | 13 |
| 3.3 | Accessing metadata | 14 |
| 3.3.1 | getNode | 14 |
| 3.3.2 | setNode | 14 |
| 3.4 | Transferring data | 15 |
| 3.4.1 | pushToVoSpace | 15 |
| 3.4.2 | pullToVoSpace | 15 |
| 3.4.3 | pullFromVoSpace | 16 |
| 3.4.4 | pushFromVoSpace | 16 |
| 3.5 | Fault arguments | 17 |
| 3.5.1 | ContainerNotFoundFault | 17 |
| 3.5.2 | LinkFoundFault | 17 |
| | References | 17 |

1 Introduction

VOSpace is the IVOA interface to distributed storage. VOSpace 1.0 [VOSpace] defined a flat, unconnected data space. VOSpace 1.1 builds on top of this and introduces the following new functionality:

- containers - this allows the grouping of data in a hierarchical fashion
- links - this allows the federation of distinct VOSpace services
- third party APIs - this allows data objects and collections to be exposed through other interfaces
- find - this offers a more extensive search capability than is provided by list with wildcard support

1.1 Document roadmap

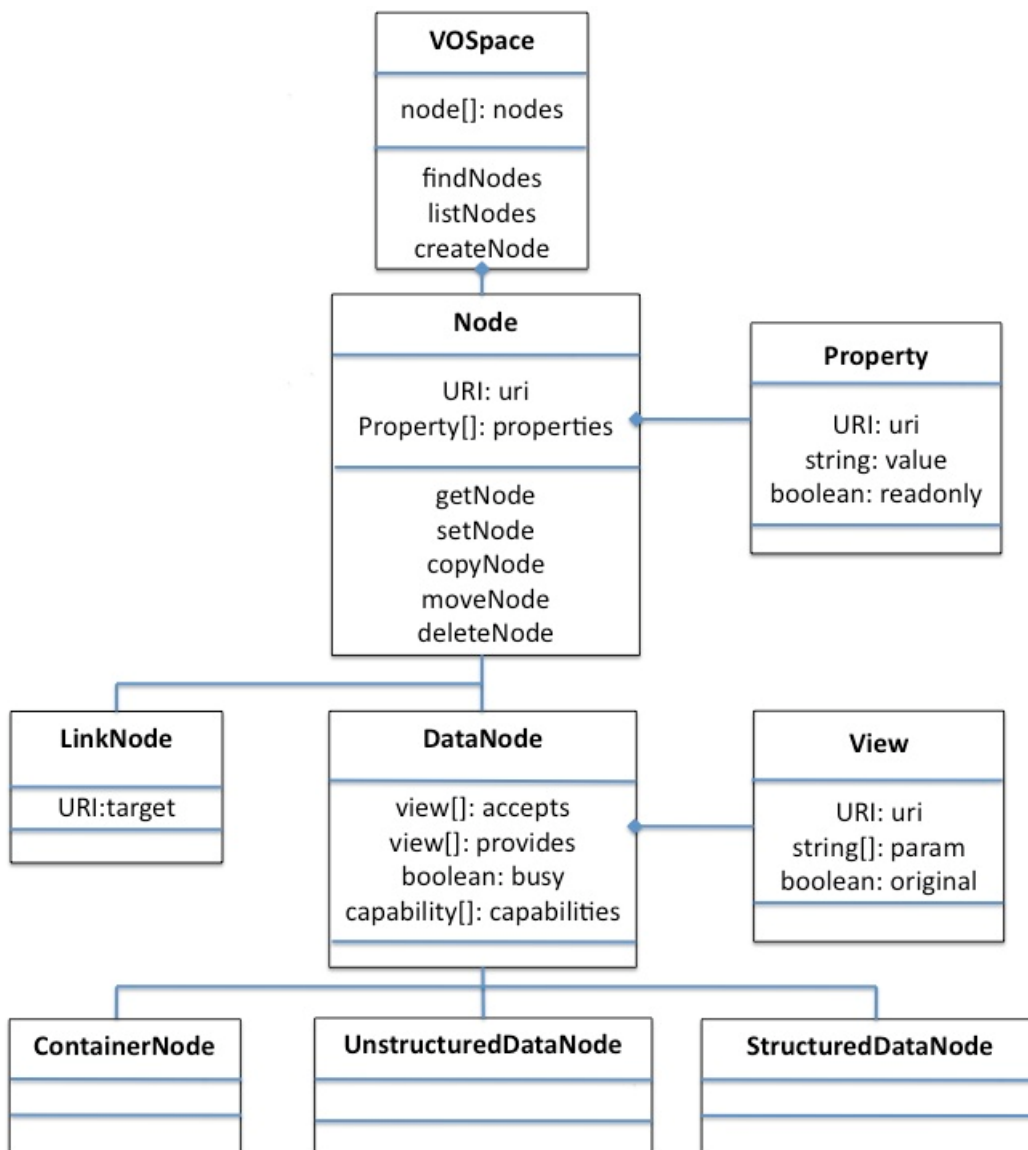
The rest of this document is structured as follows:

In Section 2, we present the updated data model for VOSpace 1.1.

In Section 3, we detail new operations and changes to existing operations with reference to VOSpace 1.0.

2 VOSpace data model

VOSpace 1.1 extends the VOSpace 1.0 data model by introducing two new node types: *ContainerNode* and *LinkNode*. A new element, capabilities, is also added to the *DataNode* node type.



[DISCUSSION POINT: Should capabilities only exist on the *ContainerNode*?]

2.1 ContainerNode

ContainerNode describes a data item that can contain other data items. These can be of any type including other *ContainerNodes*. A *ContainerNode* has no data bytes associated with it directly but only with its contents - in a tree representation, a *ContainerNode* is a branch whereas data objects are leaves.

ContainerNode extends *DataNode* and so has the following elements:

- *uri*: the `vos://` identifier for the node, URI-encoded according to RFC2396 [2].
- *properties*: a set of metadata properties for the node.
- *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 or its children cannot be accessed.

[DISCUSSION POINT: capabilities should be in this list as well since we are adding to the standard definition of *DataNode*].

The *busy* flag is used to indicate that an internal operation, such as the service implementation unpacking an archive format, is in progress and so none of the node data are available.

2.1.1 Container identifiers

Slashes in the URI path imply a hierarchical arrangement of data: the data object identified by `vos://nvo.caltech!vospace/tables/myTable1` is within the container identified by `vos://nvo.caltech!vospace/tables`. In fact, all ancestors in the hierarchy will be resolvable containers back to the root node of the space (this precludes any system of implied hierarchy in the naming scheme for nodes with ancestors that are just logical entities and cannot be reified, e.g. the Amazon S3 system).

2.1.2 Inheritable properties

Properties on a *ContainerNode* may be designated as inheritable and will propagate to children nodes of the container if they are specified in the *accepts* or *provides* list for this node.

[DISCUSSION POINT: If a property is also declared on a child, which value takes priority? How are properties registered as inheritable?]

2.1.3 Container views

For VOSpace 1.1, a view is the data representation (format) of the file that is transferred. If the view is an archive format (tar, zip, etc.) then the space will

provide access to the archive contents as children nodes of the container. Whether or not the space actually unpacks the archive is implementation dependent but the service will behave as though it has done so. For example, a client wishes to upload a tar file containing several images to a VOSpace service. If he associates it with (uploads it to) a *Structured/UnstructuredDataNode* then it will be treated as a blob and its contents will not be available. However, if he uses a *ContainerNode* with an accepts view of "tar" then the image files within the tar file will be represented as children nodes of the *ContainerNode* and accessible like any other data object within the space.

[DISCUSSION POINT: What are the names of the children nodes? Are these Structured/UnstructuredDataNodes? What is the default? How is this set?]

[DISCUSSION POINT: How does the service identify what it considers to be archive formats?]

If a *provides* view is an archive format (tar, zip, etc.) then the space will package the container and all its children nodes in the specified format.

2.2 *LinkNode*

LinkNode describes a node that points to another node. These can be of any type including other *LinkNodes*. A *LinkNode* has no data bytes associated with it.

LinkNode extends *Node* and so has the following elements associated with it:

- *uri*: the `vos://` identifier for the node, URI-encoded according to RFC2396 [REF]
- *properties*: a set of metadata properties for the node. The properties do not propagate to the target of the *LinkNode*. One use case is to enable third-party annotations to be associated with a data object but without the data object itself getting cluttered with unnecessary metadata. In this case, the client creates a *LinkNode* pointing to the data object in question and then adds the annotations as properties of the *LinkNode*.
- *target*: the identifier, URI-encoded according to RFC2396, for the data object to which the *LinkNode* points.

2.3 *Capabilities*

A *Capability* is a third-party interface to a data object. It enables data access using other non-VOSpace methods.

A *Capability* has the following members:

- *uri*: the *Capability* identifier
- *endpoint*: the endpoint URL to use for the third-party interface

[DISCUSSION POINT: Should there be any more members to a *Capability*, e.g. param to specify additional arguments that might be required for access?]

2.3.1 Example use cases

A *ContainerNode* contains image files and has a DAL SIAP capability so that the images in the container can also be accessed using a SIAP service. In this way, a user could create a container in VOSpace, drop some images into it and then query the set of images using the SIAP interface.

Another example is a *DataNode* with an iRODS capability so that the data replication for this data object can be handled using the iRODS service API located at the specified endpoint.

2.3.2 Capability identifiers

Every new type of *Capability* requires a unique URI to identify the *Capability*.

The rules for the *Capability* 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 *Capability* identifiers as anyURI [TBD]. The only restriction is that it must be a valid (unique) URI.

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

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 *Capabilities* should have unique URIs that can be resolved into a description of the *Capability*.

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

- `ivo://my-registry/vospace/capabilities#my-capability`

Using an IVO registry URI to identify *Capabilities* 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 *Capability*.

2.3.3 Capability descriptions

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

A *CapabilityDescription* should describe the third-party interface and how it should be used in this context.

A *CapabilityDescription* should have the following members:

- *uri*: the formal URI of the *Capability*
- *DisplayName*: a simple display name of the *Capability*.
- *Description*: a text block describing the third-party interface and how it should be used in this context.

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

2.3.4 UI display name

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

- Access data using `urn:edu.sdsc.irods`

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

- Access data using iRODS

2.3.5 Standard capabilities

The VOSpace team intend to register *Capability* URIs and *CapabilityDescriptions* for the core set of *Capabilities*, e.g.

- Cone Search
- SIAP
- SSAP
- TAP

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

3 Web service operations

A VOSpace 1.1 service shall be a SOAP service with the following operations:

3.1 Service metadata

3.1.1 getProtocols

This is unchanged from VOSpace 1.0 (Sec 5.1.1).

3.1.2 getViews

This is unchanged from VOSpace 1.0 (Sec 5.1.2).

3.1.3 getProperties

This is unchanged from VOSpace 1.0 (Sec 5.1.3).

[DISCUSSION POINT: Is this true - do we want to denote inheritable properties in some fashion?]

3.1.4 getCapabilities

[DISCUSSION POINT: Do we want this operation?]

3.2 Creating and manipulating data nodes

3.2.1 createNode

Create a new node at a specified location.

3.2.1.1 Parameters

This is the same as VOSpace 1.0 (Sec 5.2.1.1) except that:

- the permitted values of `xsi:type` are:
 - `vos:Node`
 - `vos:DataNode`
 - `vos:UnstructuredDataNode`
 - `vos:StructuredDataNode`
 - `vos:ContainerNode`
 - `vos:LinkNode`

`.auto` replaces `vos://null` as the reserved URI to indicate an auto-generated URI for the destination, i.e. `vos://service/path/.auto` will cause a new unique URI for the node within `vos://service/path` to be generated.

The capabilities list for the *Node* cannot be set using this method.

3.2.1.2 Returns

This is the same as VOSpace 1.0 (Sec 5.2.1.2) except that:

- the *capabilities* list for the *Node* may not be filled in until some data has been imported into the *Node*.

3.2.1.3 Faults

This is the same as VOSpace 1.0 (Sec 5.2.1.3) except that:

- * The service shall throw a *LinkFound* exception if the parent path includes a link.

- * The service shall throw a *LinkFound* exception if the parent node is a link.

- * The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*

[DISCUSSION POINT: Do we need both a *LinkFound* and a *ContainerNotFound* exception or does the latter work for both cases?]

3.2.2 deleteNode

Delete a node.

When the target is a *ContainerNode*, all its children (the contents of the container) will also be deleted.

3.2.2.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.2.2.1).

3.2.2.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.2.2.2).

3.2.2.3 Faults

This is the same as VOSpace 1.0 (Sec 5.2.2.3) except that:

- The service shall throw a *LinkFound* exception if the parent path includes a link.
- The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*.

3.2.3 listNodes

List nodes in a space.

When a target URI is a *ContainerNode*, only direct (first generation) children of the node will be listed.

3.2.3.1 Parameters

This is the same as VOSpace 1.0 (Sec 5.2.3.1) except that:

- Wild cards can only be used in the final part of the URL path: for example, `a/b/c/*.txt` is allowed by `a/*/*c/*.txt` is not.

3.2.3.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.2.3.2).

3.2.3.3 Faults

This is unchanged from VOSpace 1.0 (Sec 5.2.3.3).

3.2.4 findNodes

Find nodes whose properties match the specified values.

3.2.4.1 Parameters

- *token*: An optional continuation token from a previous request
 - No token indicates a request for a new find operation.

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 requests in the response
 - No limit indicates a request for an unpagged response. 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 response
 - *min*: The response contains the minimum detail for each *Node* with all optional parts removed - the node type should be returned
 - e.g. `<node uri="vos://service/name" xsi:type="Node"/>`
 - *max*: The response contains the maximum detail for each *Node*, including any `xsi:type` specific extensions
 - *properties*: The response contains a basic node element with a list of properties for each *Node* with no `xsi:type` specific extensions.
- *matches*: A list of match elements identifying the properties and values to match against and whether these should applied in conjunction (and) or disjunction (or).

The *match* element has a *uri* attribute to identify the property to which it is applying. The regular expression against which the property values are to be matched is then specified as the value of the match element:

```
<match uri="..."> regex </match>
```

The *match* elements can be combined in conjunction and/or disjunction by specifying them as subelements of `<or>` and `<and>` respectively. For example, the predicate "(property1 and property2) or property3" would be specified as:

```
<or>
  <and>
    <match uri="property1"> regex </match>
    <match uri="property2"> regex </match>
  </and>
  <match uri="property3"> regex </match>
</or>
```

[DISCUSSION POINT: Are wildcards allowed in the property URIs - find me all nodes where any property matches this regular expression?]

An empty list of `<matches>` implies a full listing of the space.

3.2.4.2 Returns

- *token*: An optional continuation token, indicating that the response 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 properties

3.2.4.3 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 *PropertyNotFound* exception if a particular property is specified and does not exist in the space
 - This does not apply if wildcards are allowed in the property URIs
- 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

3.2.5 moveNode

Move a node within a VOSpace service.

When the source is a *ContainerNode*, all its children (the contents of the container) will also be moved to the new destination.

When the destination is an existing *ContainerNode*, the source will be placed under it (i.e. within the container).

3.2.5.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.2.4.1).

3.2.5.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.2.4.2).

3.2.5.3 Faults

This is the same as VOSpace 1.0 (Sec 5.2.4.3) except that:

- The service shall throw a *DuplicateNode* exception if a *Node* already exists at the destination unless it is a *ContainerNode*.
- The service shall throw a *LinkFound* exception if the target path includes a link.
- The service shall throw a *LinkFound* exception if the target node is a link.
- The service shall throw a *LinkFound* exception if the parent path includes a link.
- The service shall throw a *LinkFound* exception if the parent node is a link.
- The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*
- The service shall throw an *InvalidArgument* exception if the source is a *ContainerNode* and the destination is not.
- The service shall throw a *ContainerNotFound* exception if the target node is a *ContainerNode* and does not exist.

3.2.6 copyNode

Copy a node with a VOSpace service.

When the source is a *ContainerNode*, all its children (the full contents of the container) get copied, i.e. this is a deep recursive copy.

When the destination is an existing *ContainerNode*, the copy will be placed under it (i.e. within the container).

3.2.6.1 Parameters

This is the same as VOSpace 1.0 (Sec 5.2.5.1) except that:

- `.auto` replaces `vos://null` as the reserved URI to indicate an auto-generated URI for the destination, i.e. `vos://service/path/.auto` will

cause a new unique URI for the node within `vos://service/path` to be generated.

3.2.6.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.2.5.2).

3.2.6.3 Faults

This is the same as VOSpace 1.0 (Sec 5.2.5.3) except that:

- The service shall throw a *DuplicateNode* exception if a *Node* already exists at the destination unless it is a *ContainerNode*.
- The service shall throw a *LinkFound* exception if the target path includes a link.
- The service shall throw a *LinkFound* exception if the target node is a link.
- The service shall throw a *LinkFound* exception if the parent path includes a link.
- The service shall throw a *LinkFound* exception if the parent node is a link.
- The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*
- The service shall throw an *InvalidArgument* exception if the source is a *ContainerNode* and the destination is not.
- The service shall throw a *ContainerNotFound* exception if the target node is a *ContainerNode* and does not exist.

3.3 Accessing metadata

3.3.1 getNode

Get the details for a specific Node.

3.3.1.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.3.1.1).

3.3.1.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.3.1.2).

3.3.1.3 Faults

This is the same as VOSpace 1.0 (Sec 5.3.1.3) except that:

- The service shall throw a *LinkFound* exception if the target path includes a link.

3.3.2 setNode

Set the property values for a specific node.

Changes to inheritable properties on *ContainerNodes* will propagate to children nodes of the container where applicable.

3.3.2.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.3.2.1).

3.3.2.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.3.2.2).

3.3.2.3 Faults

This is the same as VOSpace 1.0 (Sec 5.3.2.3) except that:

- The service will throw a *LinkFound* exception if the target path includes a link.

3.4 Transferring data

3.4.1 pushToVoSpace

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

3.4.1.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.4.1.1) except that:

[DISCUSSION POINT: If a *Node* already exists at the target URI and it is a *ContainerNode*, should it be overwritten by the target *Node* or should the target *Node* become a child of the *ContainerNode*? This also applies to *pullToVoSpace*.]

3.4.1.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.4.1.2).

3.4.1.3 Faults

This is the same as VOSpace 1.0 (Sec 5.4.1.3) except that:

- The service shall throw a *LinkFound* exception if the target path includes a link.
- The service shall throw a *LinkFound* exception if the target node is a link.
- The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*.

3.4.2 pullToVoSpace

Import data into a VOSpace node.

3.4.2.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.4.2.1).

3.4.2.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.4.2.2).

3.4.2.3 Faults

This is the same as VOSpace 1.0 (Sec 5.4.2.3) except that:

- The service shall throw a *LinkFound* exception if the target path includes a link.
- The service shall throw a *LinkFound* exception if the target node is a link.
- The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*.

3.4.3 pullFromVoSpace

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

3.4.3.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.4.3.1).

3.4.3.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.4.3.2).

3.4.3.3 Faults

This is the same as VOSpace 1.0 (Sec 5.4.3.3) except that:

- The service shall throw a *LinkFound* exception if the target path includes a link.
- The service shall throw a *LinkFound* exception if the target node is a link.
- The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*.

3.4.4 pushFromVoSpace

Ask the server to send data to a remote location.

3.4.4.1 Parameters

This is unchanged from VOSpace 1.0 (Sec 5.4.4.1).

3.4.4.2 Returns

This is unchanged from VOSpace 1.0 (Sec 5.4.4.2).

3.4.4.3 Faults

This is the same as VOSpace 1.0 (Sec 5.4.4.3) except that:

- The service shall throw a *LinkFound* exception if the target path includes a link.
- The service shall throw a *LinkFound* exception if the target node is a link.
- The service shall throw a *ContainerNotFound* exception if the parent path is not composed solely of *ContainerNodes*.

3.5 *Fault arguments*

This is the same as VOSpace 1.0 [Sec 5.5] with the addition of:

3.5.1 *ContainerNotFoundFault*

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

3.5.2 *LinkFoundFault*

This is thrown with the URI of the found *LinkNode*.

4 *Appendix: Machine readable definitions*

4.1 *WSDL*

A working draft version of the VOSpace 1.1 WSDL can be found on the IVOA VOSpace page at: <http://www.ivoa.net/twiki/bin/view/IVOA/VOSpaceHome>.

4.2 *Message schema*

A working draft version of the XML message schema for VOSpace 1.1 can be found on the IVOA VOSpace page at:

<http://www.ivoa.net/twiki/bin/view/IVOA/VOSpaceHome>.

References

[VOSpace] Matthew Graham, Paul Harrison, Dave Morris, Guy Rixon, VOSpace service specification v1.02, IVOA Recommendation 2007 October 01, <http://www.ivoa.net/Documents/latest/VOSpace.html>