GWD-R
Category: Recommendations

Manuel Pereira – IBM Almaden Research Center
Osamu Tatebe - Grid Technology Research Center
Leo Luan, IBM Almaden Research Center
Ted Anderson, IBM Almaden Research Center
Jane Xu, IBM Systems and Technology Group

GFS-WG

September 2004

Virtual Filesystem Directory Service Specification

Status of This Memo

This memo provides information to the Grid community about file system directory services. It does not define any standards or technical recommendations. Distribution is unlimited.

Copyright Notice

Abstract

This document describes the specification a file system directory service, which will be one of the essential services for Grid file systems or virtual file systems in the Grid environment. It manages the namespace of federated and virtualized data from file system resources, access control mechanisms, and meta-data management. This document proposes a set of operations needed to be supported by file system directory services. For scalable, large-scale and distributed file system directory management, this document also proposes two types of federation of file system directory services.

Contents

1. Introduction

Data in the Grid can be of any format and be stored in any type of storage system. There can be many hundreds of petabytes of data in grids, among which a very large percentage is stored in files. A standard mechanism to describe and organize file-based data is essential for facilitating access to this large amount of data. The Grid File System Working Group (GFS-WG) was established in GGF data area to standardize a mechanism to address this need by providing a Grid File System (GFS) or virtual file system in the Grid environment.

Two major deliverables of the WG are (1) architecture of Grid File System Services and (2) specification of Virtual Filesystem Directory Services. File system directory services will manage the namespace of federated and virtualized data from file system resources, access control mechanisms, and meta-data management [1]. It will provide features such as (a) virtualized hierarchical namespaces for files or potentially other types of data (such as live data feeds), (b) efficient and transparent file sharing, and (c) ability to describe and manage file-system and application-specific metadata.

This document intends to present the Virtual Filesystem Directory Service specification. It proposes a set of operations needed to be supported by file system directory services. For scalable, large-scale and distributed file system directory management, it also proposes two types of federation of the file system directory services.

The overall architecture of the Grid File System will be specified later in GFS-WG, which provides infrastructure of virtual file systems facilitating federation and sharing of virtualized data from file systems in the Grid environment by using Virtual Filesystem Directory Services.

2. Virtual Filesystem Directory Services

The Virtual Filesystem Directory Service, which will henceforth be referred to as VFDS, enables construction of a uniform, global, hierarchical namespace.[1] This directory service enables federation of individual files as well as filesystem trees that are exported by a variety of distributed file systems. It is comprised of two fundamental namespace components: *virtual directories* and *junctions*. These two essential namespace components, also referred to as *entries*, are employed to federate existing files and filesystem trees and construct a uniform hierarchy.

A virtual directory is a VFDS entry that is represented as a filesystem directory, however does not have any corresponding position in any physical filesystem; hence it is *virtual*. A virtual directory, therefore, is purely a namespace entity that functions in much the same way as a conventional filesystem directory by maintaining a list of subentries, which thereby demonstrate a hierarchical relationship.

A junction is a VFDS entry that interconnects an existing file or filesystem tree into the global namespace. It functions in much the same way as traditional distributed file system mount points, with the unique property of maintaining uniform namespace representation while facilitating logical as well as physical referencing to external resources. Junctions are therefore categorized into four basic types: *physical junctions*, *logical junctions*, *referrals*, and *aliases*. A physical junction maintains a URL pointer to the physical location of a filesystem tree or single file; an example URL might look something like: gsiftp://gridftp.abc.com/filesetX. In contrast, a logical junction contains a URL that points to a secondary service, like a Replica Location Service (RLS), for logical-to-physical resolution given a logical name; an example URL might look something like:

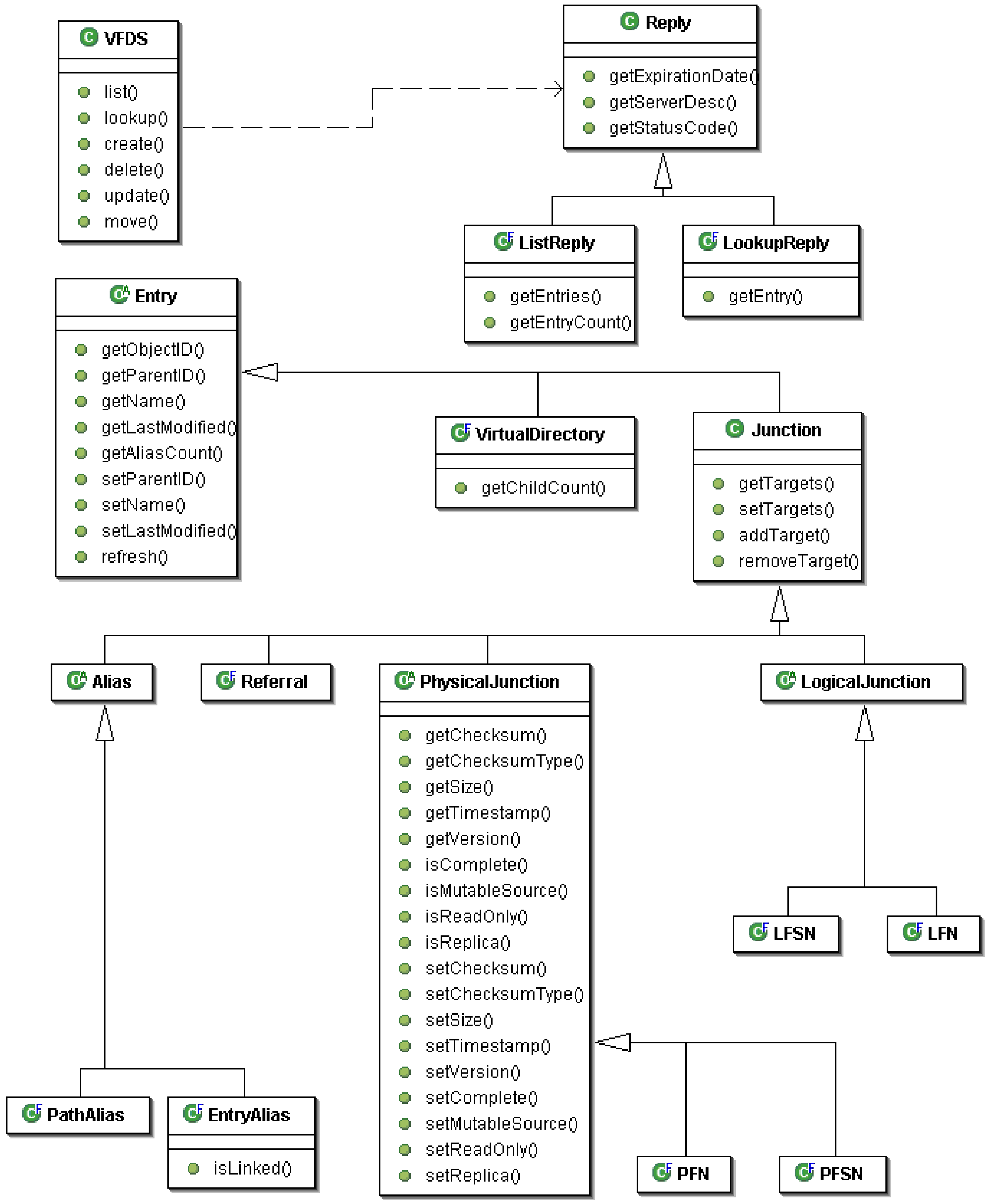
rls://rls.abc.com/logical_name_xyz. Referrals are junctions that link to other VFDS instances, thereby facilitating such features as federation of independent domains of control, scalability of a single domain of control, availability of redundant service instances that may or may not be geographically distributed, etc; an example referral URL might look something like: vfds://vfds.abc.com/. An alias entry is a junction that references another entry within the same service instance, behaving in much the same way as a convention symbolic-link; an example alias URL might look something like: “alias:/target/path” for path aliases or “alias:012345” for entry aliases. In all cases, junctions are capable of maintaining a list of references (URL pointers) per entry, that is a single junction my render several available URLs, each of which represent replicas or copies of the same data or service instance.

The following sections explore the objects and interface definitions that exemplify the operations of VFDS. This material is not comprehensive, is subject to change, and does not examine the internal procedures of the interface. Error and status codes are not specified at this time.

2.1 Virtual Filesystem Directory Service Objects

VFDS employs an object oriented service interface that consists of two chief objects: *Reply* and *Entry*. The service port type corresponds to the principal service class, that is the *VFDS* class, which defines the interface methods or port type definition in WSDL. These methods package all reply messages in *Reply* objects. Therefore, all interface methods return a *Reply* object to the requester in response to both mutator and accessor operations. The *Reply* object encapsulates the following basic members: *status/error code*, *time-to-live expiration*, *array of Entry objects* (if applicable), and *textual server description*. The *Reply* object therefore not only discloses information consequent from the request made, but also delivers all *Entry* objects corresponding to the response if the request was made to an accessor method. The *Entry* object simply represents the VFDS entry maintained by the directory service.

Following is an object diagram of the VFDS objects represented in this document:



As illustrated here, and previously mentioned, an *Entry* object can be represented as either a virtual directory or a junction. This is shown in the second row of objects in the object diagram as a subclass relationship. Notice that the *Junction* class, however, is an abstract class and is further extended by its subclasses, which represent the various types of junctions previously mentioned. Finally, notice that both the physical and logical classes are abstract and also extended to be represented by concrete classes that denote file or filesystem junctions, which are physical or logical respectively. Incidentally, these classes are as follows: PFN=Physical File Name, PFSN=Physical File System Name, LFSN=Logical File System Name, and LFN=Logical File Name.

2.2 Operations of the Virtual Filesystem Directory Service

The basic operations of VFDS are:

1) Lookup operation, governed by access control permissions, to convert a globally unique path name to the corresponding VFDS entry.
2) Creation, removal, move/rename, and update operations for all entry types.
3) Operations for managing attributes or status of an entry.

As revealed by the VFDS object diagram, the VFDS service definition specifically contains the following operations:

Operation: `create`

Description:

Creates a new VFDS entry in the directory service database that corresponds to the entry object provided and positions the new entry at the absolute path specified by the first parameter, or optionally by the OID specified in the third parameter.

This method is designed to process all defined types of entry objects, so the entry parameter should reflect the appropriate runtime cast of the abstract Entry class.

If the optional *oid* parameter is used, then the *path* argument is treated as a relative path; relative to the VFDS entry denoted by the *oid* parameter. If both an absolute path and a valid OID are provided, then an exception is thrown (absolute paths are identified by a leading slash, ie. "/gfs/ggf.org/vfds").

(similar to a `mkdir` and `mount` in a Unix shell).

Signature:

```
public Reply create( String path, Entry entry, [String oid] ) throws
ProtocolException;
```

Parameters:

path: Absolute path, or relative path if *oid* is used, of the parent of the object to create.

entry: A valid Entry object populated with all of the member values necessary to represent the entry to create. *Please note: the oid member of this object will be ignored, as the service will assign a new OID during the create operation.*

oid: *[Optional]* Object ID of the immediate parent object, provided *path* is null, otherwise it acts as a reference directory used in conjunction with the relative path value of *path*. A practical use for this parameter is to specify the object ID of the parent object and set *path* to null.

Returns:

Reply object that contains the status of the requested operation

Example:

```
try {
  /* Construct a new Virtual Directory object */
  VirtualDirectory d = new VirtualDirectory( "New Folder" );

  /* Create a new directory entry record using
   * the newly constructed object */
  vfds.create( "/gfs/ggf.org/dir1/", d );

} catch ( Exception e ) {
  // Handle
}
```

WSDL:

```
<operation name='create' parameterOrder='path entry'>
    <input message='tns:VFDS_create__Request_Soap'/>
    <output message='tns:VFDS_create_Response_Soap'/>
    <fault name='ProtocolException1' message='tns:ProtocolException1'/>
</operation>
```

Operation: `update`

Description:

Updates an existing VFDS entry in the directory service database that corresponds to the entry object represented by *original*. The values of the entry record are overwritten to reflect the values contained in the *updated* entry object.

When updating an existing VFDS entry, the *updated* Entry object must represent the complete set of values to be stored. The *updated* Entry object does not symbolize the modifications to be applied to the original record, but rather embodies the complete set of values designated to replace all previously stored values in the record. Effectively this method can be used to conveniently accomplish multiple field modifications in a single transaction.

This method is designed to process all defined types of entry objects, so the entry parameter should reflect the appropriate runtime cast of the abstract Entry class.

Signature:

```
public Reply update( Entry original, Entry updated ) throws
ProtocolException;
```

Parameters:

original: A valid Entry object that represents the existing VFDS entry record in the directory service database to be updated. *Please note: the oid member of this object must accurately signify the record to be updated; member values for this object should not be modified.*

updated: A fully populated Entry object that comprehensively represents the VFDS entry record to be stored. *Please note: any modification of the oid member of this object will be ignored, as the oid member of a VFDS entry is immutable.*

Returns:

Reply object that contains the status of the requested operation

Note:

The following members of *updated* are ignored by this method:

- Parent ID
- Object ID
- AliasCount
- ChildCount *(for Virtual Directories only)*

Example:

```
VFDS vfds = new VFDS();
LookupReply r = vfds.lookup( "/some/path" );

try {
  Junction orig    = (Junction)r.getEntry();
  Junction updated = (Junction)orig.clone();

  /* Add a new target value */
  updated.addTarget( "gsiftp://abc.com/fileset2" );

  /* Update directory service entry */
  vfds.update( orig, updated );

} catch ( Exception e ) {
  // Handle
}
```

WSDL:
```
        <operation name='update' parameterOrder='original updated'>
            <input message='tns:VFDS_update__Request_Soap'/>
            <output message='tns:VFDS_update_Response_Soap'/>
            <fault name='ProtocolException1' message='tns:ProtocolException1'/>
        </operation>
```

Operation: `delete`

Description:

Deletes the VFDS entry specified by *path* and/or *oid*. If *path* and *oid* are null an exception is thrown.

If path represents a virtual directory, all sub-entries must be deleted before it can be deleted.

If the optional *oid* parameter is used, then the *path* argument is treated as a relative path; relative to the VFDS entry denoted by the *oid* parameter. If *path* is null, then the VFDS entry identified by *oid* will be deleted.

(similar to a `rm` in a Unix shell).

Signature:

```
public Reply delete( String path, [String oid] ) throws
ProtocolException;
```

Parameters:

path: Absolute path, or relative path if *oid* is used, for the object to delete.
oid: [Optional] Object ID of a reference directory

Returns:

Reply object that contains the status of the requested operation

Example:

```
    try {
      /* Delete an existing directory entry record using
       * an absolute pathname */
      vfds.delete( "/gfs/ggf.org/vfds/oldentry" );

    } catch ( Exception e ) {
      // Handle
    }
```

WSDL:

```
        <operation name='delete' parameterOrder='path'>
            <input message='tns:VFDS_delete__Request_Soap'/>
            <output message='tns:VFDS_delete_Response_Soap'/>
            <fault name='ProtocolException' message='tns:ProtocolException'/>
        </operation>
```

Operation: `list`

Description:

Returns a Reply object that represents the directory listing of the specified path that symbolizes a virtual directory; if *path* and *oid* are null then an exception is thrown.

If *path* represents a virtual directory, its contents will be returned; otherwise an exception is thrown.

If the optional *oid* parameter is used, then the *path* argument is treated as a relative path; relative to the VFDS entry denoted by the *oid* parameter. If *path* is null, then the VFDS entry identified by *oid* will be listed (see example).

(similar to a `ls -al` in a Unix shell).

Signature:

```
public ListReply list( String path, [String oid] ) throws
ProtocolException;
```

Parameters:

path: The full path, or relative path if *oid* is used, to the virtual directory to be listed.
oid: [Optional] Object ID of a reference virtual directory (signifies the virtual directory to list if *path* is null).

Returns:

A ListReply object containing directory listing (array of Entry objects).

Example:

```
    try {
      /* Get current working directory */
      VirtualDirectory cwd = (VirtualDirectory)
                             vfds.lookup( "/gfs/ggf.org/vfds/" ).getEntry();

      /* List the current working directory (cwd) */
      ListReply r = vfds.list( null, cwd.getObjectID() );
      ...
    } catch ( Exception e ) {
      // Handle
    }
```

WSDL:

```
        <operation name='list' parameterOrder='path'>
            <input message='tns:VFDS_list__Request_Soap'/>
            <output message='tns:VFDS_list_Response_Soap'/>
            <fault name='ProtocolException3' message='tns:ProtocolException3'/>
        </operation>
```

Operation:	`lookup`

Description:

Returns a Reply object that contains an Entry object, which represents the VFDS entry and all associated attributes of the specified path; if *path* and *oid* are null then an exception is thrown.

If the path denotes a virtual directory, then only the Entry object that represents that VFDS virtual directory is returned; not its sub-entries. (see list() for a listing of sub-entries)

If the optional *oid* parameter is used, then the *path* argument is treated as a relative path; relative to the VFDS entry denoted by the *oid* parameter. If *path* is null, then the VFDS entry identified by *oid* will be looked-up.

(similar to a `stat()` call).

Signature:

```
public LookupReply lookup( String path, [String oid] ) throws
ProtocolException;
```

Parameters:

path: The full path, or relative path if *oid* is used, to the entry whose attributes are desired.
oid: [Optional] Object ID of a reference virtual directory (signifies the entry to lookup if *path* is null).

Returns:

A LookupReply object containing the Entry object requested.

Example:

```
try {
  /* Lookup a specific entry */
  LookupReply r = vfds.lookup( "/gfs/ggf.org/vfds/entry1" );

  Entry e = r.getEntry();
  if ( e instanceof VirtualDirectory ) {
    // Cast to a virtual directory object
  } else if ( e instanceof Referral ) {
    // Cast to a referral object
  } else if ( e instanceof PFN ) {
    // Cast to a Physical File Name object
  } else if ( e instanceof PFSN ) {
    // Cast to a Physical Filesystem Name object
  } else if ( e instanceof LFN ) {
    // Cast to a Logical File Name object
  } else if ( e instanceof LFSN ) {
    // Cast to a Logical Filesystem Name object
  } else if ( e instanceof EntryAlias ) {
    // Cast to an entry alias object
  } else if ( e instanceof PathAlias ) {
    // Cast to a path alias object
  }
} catch ( Exception e ) {
  // Handle
}
```

WSDL:

```
        <operation name='lookup' parameterOrder='path'>
            <input message='tns:VFDS_lookup__Request_Soap'/>
            <output message='tns:VFDS_lookup_Response_Soap'/>
            <fault name='ProtocolException4' message='tns:ProtocolException4'/>
        </operation>
```

Operation:	`move`

Description:

Moves or renames the VFDS entry denoted by *src* to *dest*.

At least one source (*src* and/or *srcOID*) and one destination (*dest* and/or *destOID*) parameter is required.

(similar to a `mv` in a Unix shell)

Signature:

```
public Reply move( String src, String dest, [String srcOID], [String
destOID] ) throws ProtocolException;
```

Parameters:

src:	Absolute path, or relative path if *srcOID* is used, to source/original VFDS entry.
dest:	Absolute path, or relative path if *destOID* is used, to destination for the VFDS entry denoted by src.
srcOID:	[Optional] Object ID of a reference source entry (signifies the source entry if *src* is null).
destOID:	[Optional] Object ID of a reference destination entry (signifies the destination entry if *dest* is null).

Returns:

Reply object that contains the status of the requested operation

Example:

```
try {
  /* Lookup parent dir */
  Entry e = vfds.lookup( "/gfs/ggf.org/vfds/" ).getEntry();

  /* Rename "name1" to "new-name1" */
  Reply r = vfds.move( "name1", "new-name1",
                       e.getObjectID(), e.getObjectID() );
} catch ( Exception e ) {
  // Handle
}
```

WSDL:

```
<operation name='move' parameterOrder='src dest'>
    <input message='tns:VFDS_move__Request_Soap'/>
    <output message='tns:VFDS_move_Response_Soap'/>
    <fault name='ProtocolException2' message='tns:ProtocolException2'/>
</operation>
```

2.3 Description of WSDL Complex Types

The objects used as components of the communication interface defined by WSDL, also referred to as *complex types*, are the two chief service interface objects previously enumerated: *Reply* and *Entry*. These two objects serve as integral components in the task of communicating and representing the VFDS service.

2.3.1 Reply Object

The following describes the interface definition for the VFDS *Reply* object:

Method:	`getExpirationDate`

Description:
Returns the expiration timestamp of this Reply, which is the time at which the Reply should be considered stale. This property is analogous to time-to-live (TTL).

Signature:
```
public Date getExpirationDate();
```

Returns:
The expiration date of this Reply, if available and appropriate; otherwise a null value is used to indicate this reply is a point-in-time reply.

WSDL:
```
<xsd:complexType name="Reply">
  ...
  <xsd:sequence>
    <xsd:element name="expirationDate" type="xsd:dateTime" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getStatusCode`

Description:
Returns the status or error code; otherwise referred to as a reply code. Every reply from the VFDS service will contain some status or error code and is readily available via this method. A value of zero '0' indicates success, while a value other than zero may correspond to an error.

Signature:
```
public int getStatusCode();
```

Returns:
Status/error code

WSDL:
```
<xsd:complexType name="Reply">
  ...
  <xsd:sequence>
    <xsd:element name="statusCode" type="xsd:int"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getServerDesc`

Description:
Returns a simple character string that represents the text that VFDS server implementations use to identify themselves to VFDS service users for usage tracking, version identification, etc.

Signature:
```
public String getServerDesc();
```

Returns:
VFDS server implementation description

WSDL:
```
<xsd:complexType name="Reply">
  ...
  <xsd:sequence>
    <xsd:element name="serverDesc" type="xsd:string" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

2.3.1.1 Reply Subclass: LookupReply

The VFDS *LookupReply* class is a concrete class that is a direct subclass of *Reply*. It appends one accessor method to the structural definition of *Reply* for the purpose of enabling access to enclosed content:

Method:	`getEntry`

Description:
Returns the Entry object representing the result of a lookup operation. This method will return null if there were no Entry objects returned by the remote operation.

Signature:
```
public Entry getEntry();
```

Returns:
The Entry object contained within this LookupReply object

WSDL:
```
<xsd:complexType name="Reply">
  ...
  <xsd:sequence>
    <xsd:element name="entry" type="tns:Entry" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

2.3.1.2 Reply Subclass: ListReply

The VFDS *ListReply* class is a concrete class that is a direct subclass of *Reply*. It appends two accessor methods to the structural definition of *Reply* for the purpose of enabling access to enclosed content:

Method:	`getEntryCount`

Description:
Returns the number of entries contained within this ListReply object.

Signature:
```
public int getEntryCount();
```

Returns:
The number of entries contained within this ListReply object

WSDL:
```
<xsd:complexType name="Reply">
  ...
  <xsd:sequence>
    <xsd:element name="entryCount" type="xsd:int"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getEntries`

Description:
Returns an array of Entry objects representing the result of a list operation. This method will return null if there were no Entry objects returned by the remote operation.

Signature:
```
public Entry[] getEntries();
```

Returns:
The array of Entry objects contained within this ListReply object

WSDL:
```
<xsd:complexType name="Reply">
  ...
  <xsd:sequence>
    <xsd:element name="entries" type="tns:ArrayOfEntry" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

2.3.2 Entry Object

All valid VFDS *Entry* objects are characterized by the following abstract class definition, which is to say that the base *Entry* class is abstract and serves as a structural definition for its various subclass heirs.

The following describes the interface definition for the VFDS *Entry* object:

Method: `getLastModified`

Description:
Returns the last modified timestamp of the VFDS entry represented by this object. This timestamp reflects the last time the corresponding VFDS entry was modified within the namespace repository and does not reflect the last modified timestamp from any physical target.

Last modified timestamps may be set by an administrator, however in practice the last modified value of an entry is updated automatically and thereby denotes the last time the directory service database record was modified.

Signature:
```
public Date getLastModified();
```

Returns:
Modification timestamp of the VFDS entry this object represents

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="lastModified" type="xsd:dateTime" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `getName`

Description:
Returns the name of the VFDS entry this object represents.

Signature:
```
public String getName();
```

Returns:
Name of the represented VFDS entry

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `getObjectID`

Description:
Returns the session specific object ID that uniquely identifies the VFDS entry this object represents.

Signature:
```
public String getObjectID();
```

Returns:
Object ID of the represented VFDS entry

Note:
The Object ID member is a read-only member and once a directory service entry is created, the Object ID of the entry cannot be changed.

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="oid" type="xsd:string" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `getParentID`

Description:
Returns the session specific object ID of the parent object that uniquely identifies the parent VFDS entry of the VFDS entry this object represents.

Signature:
```
public String getParentID();
```

Returns:
Parent object ID of the represented VFDS entry

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="pid" type="xsd:string" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `getAliasCount`

Description:
Returns the number of aliases this VFDS entry has associated with it. An alias is a VFDS entry that symbolizes another VFDS entry via a different pathname (see 2.3.2.2.2 Junction Subclass: Alias).

Signature:
```
public int getAliasCount();
```

Returns:
Number of the associated aliases

Note:
The alias count member is a read-only attribute and cannot be set by any method.

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="aliasCount" type="xsd:ArrayOfInt" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `setLastModified`

Description:
Sets the last modified timestamp of the VFDS entry represented by this object. This timestamp reflects the last time the corresponding VFDS entry was modified within the namespace repository and does not reflect the last modified timestamp from any physical target.

Last modified timestamps may be set by an administrator, however in practice the last modified value of an entry is updated automatically and thereby denotes the last time the directory service database record was modified.

Signature:
```
public void setLastModified( Date timestamp );
```

Parameters:
timestamp: The timestamp to be used to represent the last modified time of the namespace entry

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="lastModified" type="xsd:dateTime" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`setName`

Description:
Sets the name of the VFDS entry this object represents.

Signature:
```
public void setName( String name );
```

Parameters:
name: The name to be assigned to the entry represented by this object.

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`setParentID`

Description:
Sets the session specific object ID of the parent object that uniquely identifies the parent VFDS entry of the VFDS entry this object represents.

Signature:
```
public void setParentID( String pid );
```

Parameters:
pid: The Object ID of the parent directory.

WSDL:
```
<xsd:complexType name="Entry">
  ...
  <xsd:sequence>
    <xsd:element name="pid" type="xsd:string" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`refresh`

Description:
Refreshes, or re-synchronizes, the member values of the current in memory object with the values stored in the directory service database.

Signature:
```
public void refresh();
```

2.3.2.1 Entry Subclass: VirtualDirectory

The VFDS *VirtualDirectory* class is a concrete class that is a direct subclass of *Entry*. It appends one method to the structural definition of *Entry*:

Method:	`getChildCount`

Description:
Returns the number of immediate subentries related to this virtual directory in the directory hierarchy.

Signature:
```
public int getChildCount();
```

Returns:
Number of immediate subentries related to this VirtualDirectory

Note:
The child count member is a read-only attribute and cannot be set by any method.

WSDL:
```
<xsd:complexType name="VirtualDirectory">
  ...
  <xsd:sequence>
    <xsd:element name="childCount" type="xsd:int"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

2.3.2.2 Entry Subclass: Junction

The VFDS *Junction* class is an abstract class that is a direct subclass of *Entry*. It appends one method to the structural definition of *Entry*:

Method:	`getTargets`

Description:
Returns at least one String, typically a URL, that identifies the target resource for the VFDS junction represented by this object. The URL syntax must comply with RFC 1738.

Signature:
```
public String[] getTargets();
```

Returns:
An array of URLs or end point references that refer to target resources denoted by this junction

WSDL:
```
<xsd:complexType name="Junction">
  ...
  <xsd:sequence>
    <xsd:element name="targetURL" type="xsd:ArrayOfString" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`setTargets`

Description:
Sets the target String array, typically in the form of URLs, that identifies all available target resources for the VFDS junction represented by this object. Any given junction may have more than one target resource, identical in content but different in reference or location.

The URL syntax must comply with RFC 1738.

Signature:
```
public void setTargets( String[] targets );
```

Parameters:
targets: A String array of target URLs

WSDL:
```
<xsd:complexType name="Junction">
  ...
  <xsd:sequence>
    <xsd:element name="targetURL" type="xsd:ArrayOfString" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	addTarget

Description:

Adds a single target URL to the list of registered targets of the current Junction object. This method automatically increases the internal array of targets and assigns the *target* parameter to the end of the array.

The URL syntax must comply with RFC 1738.

Signature:

```
public void addTarget( String target );
```

Parameters:

target:	A String representing a target resource, typically in the form of a URL, to be added to the list of targets for a given junction

WSDL:

```
<xsd:complexType name="Junction">
  ...
  <xsd:sequence>
    <xsd:element name="targetURL" type="xsd:ArrayOfString" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	removeTarget

Description:

Removes a single target URL from the list of registered targets of the current Junction object. This method automatically removes the *target* parameter from the list of existing targets and resizes the internal array.

Notice that this method does not in any way affect the physical target, but rather “dereferences” the physical target by removing the reference to it from the junction’s list of target resources.

The URL syntax must comply with RFC 1738.

Signature:

```
public void removeTarget( String target );
```

Parameters:

target:	A String representing a target resource, typically in the form of a URL, to be removed from the list of targets for a given junction. *Please note: the target parameter is case sensitive and must match exactly before the target will be removed from the list.*

WSDL:

```
<xsd:complexType name="Junction">
  ...
  <xsd:sequence>
    <xsd:element name="targetURL" type="xsd:ArrayOfString" nillable="false"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

2.3.2.2.1 Junction Subclass: Referral

The VFDS *Referral* class is an abstract class that is a direct subclass of *Junction*. A *Referral* is a VFDS entry that refers to another VFDS service instance. This enables delegate VFDS service instances with the potential of separation of administrative ownership and control.

2.3.2.2.2 Junction Subclass: Alias

The VFDS *Alias* class is an abstract class that is a direct subclass of *Junction*. An *Alias* is a VFDS entry that symbolizes another VFDS entry. It is effectively an “alias” or alternate name for the entry being symbolized, and may have a different path and/or name that uniquely identifies it in the global namespace.

2.3.2.2.2.1 Junction Subclass: EntryAlias

The VFDS *EntryAlias* class is a concrete class that is a direct subclass of *Alias*. An *EntryAlias* persistently maintains its referential relationship with the VFDS entry it is aliasing by object ID. Since entry aliases point to target entries by their respective object IDs, the targeted VFDS entry may be moved and/or renamed without dereferencing the alias entry.

Entry aliases truly symbolize the VFDS entry that they target by simply presenting an alternate name to the same entry. Provided that the user creating the alias entry has the appropriate authorization, the establishment of the alias entry will transactionally increment an alias/link count attribute of the targeted entry. If the user creating the alias entry does not have adequate permission to alter the alias count attribute of the target entry, then the alias is created without a corresponding “alias count” reference.

Method:	`isLinked`

Description:

Identifies whether or not this EntryAlias is considered “linked”, which if true denotes a special relationship between the targeted VFDS entry object and the alias object in that the target object includes this alias object in its alias count.

Similar to Unix style “hard-links”, a “linked” EntryAlias enables the original entry to be deleted without losing access to its content. An entry will remain accessible as long as there is at least one “linked” EntryAlias reference to it; even if the original entry has been deleted. If the EntryAlias object, for whatever reason, is not “linked” to its target then the target entry will not have “knowledge” of this reference and consequently act as a Unix style “soft-link”; this implies that a non-linked EntryAlias can become an orphaned object, pointing to an extinct target.

Signature:
```
public boolean isLinked();
```

Returns:

True if the target VFDS entry’s alias count attribute was incremented; false if it was not.

Note:

The *linked* member is a read-only attribute and cannot be set by any method.

WSDL:
```
  <xsd:complexType name="EntryAlias">
    ...
    <xsd:sequence>
      <xsd:element name="linked" type="xsd:boolean"/>
      ...
    </xsd:sequence>
  </xsd:complexType>
```

2.3.2.2.2.2 Junction Subclass: PathAlias

The VFDS *PathAlias* class is a concrete class that is a direct subclass of *Alias*. A *PathAlias* refers to a target VFDS entry by pathname. If the VFDS entry object that represents the target pathname is moved, renamed, or deleted, the *PathAlias* object reference will become invalid.

2.3.2.2.3 Junction Subclass: LogicalJunction

The VFDS *LogicalJunction* class is an abstract class that is a direct subclass of *Junction*. A *LogicalJunction* is a VFDS entry that junctions a resource by its unique logical name, which implies the necessity of a Logical-to-Physical resolver in order to obtain a physical location of the data represented; an example resolver might be something like a *Replica Location Service*.

2.3.2.2.4 Junction Subclass: PhysicalJunction

The VFDS *PhysicalJunction* class is an abstract class that is a direct subclass of *Junction*. It appends four methods to the structural definition of *Junction*:

Method:	`isComplete`

Description:
Identifies whether or not the file or filesystem source targeted by this VFDS entry is complete. In the case of files, a value of true connotes all of the file content is embodied in the file; for filesets (filesystem subtrees) this identifies whether or not the fileset is complete in terms of number of files participating and the coherency of these files.

Signature:
```
public boolean isComplete();
```

Returns:
True if the data source is complete or “all there”; false if it is a partial data source.

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="complete" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`isMutableSource`

Description:
Identifies whether or not the file or filesystem source targeted by this VFDS entry can change.

Signature:
```
public boolean isMutableSource();
```

Returns:
True if the data source is mutable; false if it is not.

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="mutableSource" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `isReadOnly`

Description:
Identifies whether or not a local copy of the data should be locally read-only.

Signature:
```
public boolean isReadOnly();
```

Returns:
True if the local copy of the data should be read-only; false if the local copy allows writes.

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="readOnly" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `isReplica`

Description:
Identifies whether or not the file or filesystem source targeted by this VFDS entry is a replica copy.

Signature:
```
public boolean isReplica();
```

Returns:
True if the data source is a replica copy; false if it is an original data source.

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="replica" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getChecksum`

Description:
Returns a string representation of the actual checksum corresponding to the physical file or fileset symbolized by this PhysicalJunction object.

Signature:
```
public String getChecksum();
```

Returns:
The checksum of the file or fileset

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="checksum" type="xsd:string" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getChecksumType`

Description:
Returns a string representation of the checksum type or algorithm used to produce the checksum (see getChecksum()).

Signature:
```
public String getChecksumType();
```

Returns:
The algorithm employed to produce the checksum
(example checksum types may include: MD5, CRC-32, RSA-MD4, DES)

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="checksumType" type="xsd:string" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getSize`

Description:

This represents the physical size of the targeted data source. If the target data is in the form of a file (implying a PFN) then this value discloses the size of the file in bytes. If the target data is in the form of a fileset (implying a PFSN) then this value discloses the summation size of all the contained files.

Signature:

```
public long getSize();
```

Returns:

Number of bytes used by the targeted data if available

WSDL:

```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="size" type="xsd:long"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getTimestamp`

Description:

This represents a replica or fileset’s point-in-time timestamp corresponding to the time at which the source snapshot was made.

Signature:

```
public Date getTimestamp();
```

Returns:

The point-in-time timestamp of the targeted data

WSDL:

```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="timestamp" type="xsd:dateTime" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	`getVersion`

Description:
This represents a version number of the targeted data if available.

Signature:
```
public float getVersion();
```

Returns:
Version number of the targeted data if available

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="version" type="xsd:float"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method:	set`Complete`

Description:
Set the *complete* state for the object representing the physical resource (see *isComplete()*).

Signature:
```
public void setComplete( boolean flag );
```

Parameters:
value: True if the corresponding physical resource is complete

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="complete" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: setMutableSource

Description:
Sets whether or not the file or filesystem source targeted by this VFDS entry can change.

Signature:
```
public void setMutableSource( boolean flag );
```

Parameters:

value: True if the corresponding physical resource can change

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="mutableSource" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: setReadOnly

Description:
Sets whether or not a local copy of the data should be locally read-only.

Signature:
```
public void setReadOnly( boolean flag );
```

Parameters:

value: True if the corresponding physical resource only exports read access. *Please notice: this doesn't necessarily indicate the source copy of data is read-only, but rather identifies whether the source copy allows writes from foreign replicas.*

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="readOnly" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `setReplica`

Description:
Sets whether or not the file or filesystem source targeted by this VFDS entry is a replica copy.

Signature:
```
public void setReplica( boolean flag );
```

Parameters:
value: True if the corresponding physical resource is a replica copy and not the “authoritative” source.

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="replica" type="xsd:boolean"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `setChecksum`

Description:
Sets the string that represents the actual checksum corresponding to the physical file or fileset symbolized by this PhysicalJunction object. See also *setChecksumType()*.

Signature:
```
public void getChecksum( String checksum );
```

Parameters:
checksum: The checksum value that represents the checksum of the physical resource.

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="checksum" type="xsd:string" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `setChecksumType`

Description:
Sets the string that represents the checksum type or algorithm used to produce the checksum (see *setChecksum()*).

Signature:
```
public void setChecksumType( String checksumType );
```

Parameters:
checksum: The checksum type or algorithm

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="checksumType" type="xsd:string" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: `setSize`

Description:
Sets the value representing the physical size of the targeted data source. If the target data is in the form of a file (implying a PFN) then this value discloses the size of the file in bytes. If the target data is in the form of a fileset (implying a PFSN) then this value discloses the summation size of all the contained files. See also *getSize()*.

Signature:
```
public void setSize( long size );
```

Parameters:
size: The physical size of the targeted data source (number of bytes)

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="size" type="xsd:long"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: setTimestamp

Description:
Sets the point-in-time timestamp corresponding to the time at which the targeted source snapshot was made, if such a value is appropriate and available.

Signature:
```
public void setTimestamp( Date date );
```

Parameters:
date: The point-in-time timestamp of the corresponding physical data source

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="timestamp" type="xsd:dateTime" nillable="true"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

Method: setVersion

Description:
Sets the version number of the targeted data, if such a value is appropriate and available.

Signature:
```
public void setVersion( float version );
```

Parameters:
version: The version number of the corresponding physical data source

WSDL:
```
<xsd:complexType name="PhysicalJunction">
  ...
  <xsd:sequence>
    <xsd:element name="version" type="xsd:float"/>
    ...
  </xsd:sequence>
</xsd:complexType>
```

2.4 Operation Sequence of VFDS

This section will briefly outline the operational sequence of communicating and interacting with VFDS.

2.4.1 List entries of a virtual directory

To retrieve a list of directory entries for a given virtual directory in the global namespace hierarchy, the *list* operation is utilized. The client making such a request would simply perform a *list* service operation against the VFDS service, whose returning messages will be constructed into a *Reply* object. The requesting client then retrieves the list of all subentries, via the *getEntries* method, immediately related to the virtual directory specified by the path used in the *list* operation. At this point the requesting client can process any number (or all) of the returned *Entry* objects. If an unexpected exception occurs, the status code available via the *Reply* object can be checked.

In summary, the following operations are necessary to retrieve a directory listing in VFDS:
1) Invoke the *list* operation of the VFDS service, specifying the globally unique path of the virtual directory to list.
2) Retrieve all *Entry* objects from the *Reply* object by calling the *getEntries* method.
3) Enumerate the *Entry* objects delivered via the *Reply* object.

2.4.2 Create a virtual directory

To create a virtual directory in the global namespace hierarchy, the *create* operation is utilized. The client placing such a request would simply perform a *create* service operation against the VFDS service, specifying the global path and providing an instantiated *VirtualDirectory* object as the *Entry* parameter. Once the service completes this operation, its returning messages will be constructed into a *Reply* object. The requesting client then evaluates the status code available via the *getStatusCode* method of the *Reply* object. If an unexpected exception occurs, the status code available via the *Reply* object can be checked.

In summary, the following operations are necessary to retrieve a directory listing in VFDS:
1) Construct a *VirtualDirectory* object with the appropriate property values assigned (name, modification time, etc.).
2) Invoke the *create* operation of the VFDS service, specifying the globally unique path of the virtual directory to create and providing the previously instantiated *VirtualDirectory* object from step 1 as the *Entry* parameter.
3) Evaluate the status code returned by the service via the *getStatusCode* method of the *Reply* object.

3. Federation of Virtual Filesystem Directory Services

A global namespace service directly implies the employment of a multitude of namespace servers by virtue of geographical distribution, segregated domains of ownership and control, scalability, and redundancy/availability. A principal goal of a global namespace service is to provide a location independent view of consistent access paths to data. Since these access paths are represented by hierarchal path names, symbolizing a globally unique identifier of data, it is a natural extension of the design to postulate an architecture that federates multiple namespace servers in a hierarchical fashion. Similar to the well established DNS model, VFDS servers can be interlinked by referrals whilst providing a seamless and transparent view of the namespace. Once several instances of the namespace service are interlinked, the most obvious challenge is related to path name resolution when dealing with paths that cross referral boundaries. There are two fundamental approaches to resolving path names that span multiple namespace domains or service instances: service referrals and delegated resolution.

3.1 Service Referrals

The most straightforward and arguably the most secure and truly scalable approach to resolving path names that span multiple domains or service instances is to place the onus of handling VFDS referrals on the VFDS client. In this approach, the namespace server would simply return a VFDS referral to the VFDS client when a junction to another namespace server is encountered. The client implementing the VFDS API is then responsible for continuing the task of resolving the original path name by connecting to the namespace server indicated by the VFDS referral and querying the newly connected server for further (relative) path name resolution.

One clear advantage of this approach is the direct management of namespace service connections, which implies authentication and authorization control per connection, rather than accessing a referred namespace server via proxied security. Additionally, this approach promotes distributed work load balancing; instead of requiring VFDS servers to handle namespace requests for both locally managed namespace and remotely managed namespace via proxy.

3.2 Delegated Resolution

Another possible approach to resolving path names that span multiple domains or service instances is to empower the VFDS server to delegate queries to other VFDS servers for complete resolution of any given path. Although this approach is demonstrated in DNS, it should be noted that the security requirements are quire different. Since DNS generally operates in a public read-only manner without authentication and authorization per DNS server, it is not too unreasonable to endorse such an approach. VFDS, however, facilitates the possibility of requiring authentication per service instance and enforcing access control per entry. Nevertheless, an approach that allows for the possibility of delegated resolution should be considered as at least an optional mode of operation; incidentally DNS is capable of both approaches.

4. Considerations

There are several issues to consider, with respect to VFDS, that have not been explored in this document.

- Security – The topic of security as a whole is not discussed in this specification document. Security is recognized as a substantial are of interest and will require further investigation.
- Replication of VFDS databases – To enhance fault tolerance and reliability, replication of namespace service data is indispensable. The consistency model required by VFDS needs to be investigated.
- Backup – Backup of VFDS data may be required.
- Discussion of access control lists (ACLs) within VFDS, their purpose, scope, representation, and enforcement. If access permissions defined by physical filesystems are to be represented within VFDS then significant consideration must be taken with respect to consistency problems between access permissions of a virtual file and the corresponding file data.

? Removal or modification of a file data without notification to the file system directory services.

? Consistency problems between file data replicas.

? Interoperability issue with NFSv4 and CIFS.

5. Summary and conclusion

This document intended to describe the specification of the Virtual Filesystem Directory Service, which will be one of the essential services for the realization of a Grid File System. It manages the namespace of federated and virtualized data from file system resources, access control mechanisms, and meta-data management.

This document proposed a set of operations needed to be supported by VFDS. Additionally, it proposed two approaches to federation of VFDS service instances for scalable, large-scale and distributed namespace management.

Further detailed discussion for specification and evaluation by implementing file system directory services are needed with respect to security, performance, consistency, scalability, and reliability. The evaluation needs to consider functionality of a client library, especially, with and without client attribute cache.

6. Appendix: Sample WSDL for VFDS

The following is a sample Web Services Description Language (WSDL) file intended to describe VFDS in terms of a web service using a standard XML format.

```
<service name='VFDS'>
    <port name='VFDS' binding='tns:VFDS'>
        <soap:address location='http://vfds.almaden.ibm.com:6060/VFDS/'/>
    </port>
</service>
<portType name='VFDS'>
    <operation name='create' parameterOrder='path entry'>
        <input message='tns:VFDS_create__Request_Soap'/>
        <output message='tns:VFDS_create_Response_Soap'/>
        <fault name='ProtocolException1' message='tns:ProtocolException1'/>
    </operation>
    <operation name='delete' parameterOrder='path'>
        <input message='tns:VFDS_delete__Request_Soap'/>
        <output message='tns:VFDS_delete_Response_Soap'/>
        <fault name='ProtocolException' message='tns:ProtocolException'/>
    </operation>
    <operation name='list' parameterOrder='path'>
        <input message='tns:VFDS_list__Request_Soap'/>
        <output message='tns:VFDS_list_Response_Soap'/>
        <fault name='ProtocolException3' message='tns:ProtocolException3'/>
    </operation>
    <operation name='lookup' parameterOrder='path'>
        <input message='tns:VFDS_lookup__Request_Soap'/>
        <output message='tns:VFDS_lookup_Response_Soap'/>
        <fault name='ProtocolException4' message='tns:ProtocolException4'/>
    </operation>
    <operation name='move' parameterOrder='src dest'>
        <input message='tns:VFDS_move__Request_Soap'/>
        <output message='tns:VFDS_move_Response_Soap'/>
```

```
            <fault name='ProtocolException2' message='tns:ProtocolException2'/>
        </operation>
        <operation name='update' parameterOrder='original updated'>
            <input message='tns:VFDS_update__Request_Soap'/>
            <output message='tns:VFDS_update_Response_Soap'/>
            <fault name='ProtocolException1' message='tns:ProtocolException1'/>
        </operation>
    </portType>
    <binding name='VFDS' type='tns:VFDS'>
        <soap:binding transport='http://schemas.xmlsoap.org/soap/http' style='document'/>
        <operation name='create'>
            <map:java-operation name='create' signature='xxx'>
                <map:fault name='ProtocolException1' java-
type='com.ibm.vfds.ProtocolException'/>
            </map:java-operation>
            <soap:operation
               soapAction='http://systinet.com/wsdl/com/ibm/vfds/VFDS#create?xxx'
style='document'/>
            <input>
                <soap:body parts='path entry' use='literal'/>
            </input>
            <output>
                <soap:body parts='response' use='literal'/>
            </output>
            <fault name='ProtocolException1'>
                <soap:fault name='ProtocolException1' use='literal'/>
            </fault>
        </operation>
        <operation name='delete'>
            <map:java-operation name='delete' signature='xxx'>
                <map:fault name='ProtocolException' java-
type='com.ibm.vfds.ProtocolException'/>
            </map:java-operation>
            <soap:operation
               soapAction='http://vfds.almaden.ibm.com/wsdl/com/ibm/vfds/VFDS#delete?xxx'
               style='document'/>
            <input>
                <soap:body parts='path' use='literal'/>
            </input>
            <output>
                <soap:body parts='response' use='literal'/>
            </output>
            <fault name='ProtocolException'>
                <soap:fault name='ProtocolException' use='literal'/>
            </fault>
        </operation>
        <operation name='list'>
            <map:java-operation name='list' signature='xxx'>
                <map:fault name='ProtocolException3' java-
type='com.ibm.vfds.ProtocolException'/>
            </map:java-operation>
            <soap:operation
soapAction='http://vfds.almaden.ibm.com/wsdl/com/ibm/vfds/VFDS#list?xxx'
               style='document'/>
            <input>
                <soap:body parts='path' use='literal'/>
            </input>
            <output>
                <soap:body parts='response' use='literal'/>
            </output>
            <fault name='ProtocolException3'>
                <soap:fault name='ProtocolException3' use='literal'/>
            </fault>
        </operation>
        <operation name='lookup'>
            <map:java-operation name='lookup' signature='xxx'>
```

```
                <map:fault name='ProtocolException4' java-
type='com.ibm.vfds.ProtocolException'/>
            </map:java-operation>
            <soap:operation
               soapAction='http://vfds.almaden.ibm.com/wsdl/com/ibm/vfds/VFDS#lookup?xxx'
               style='document'/>
            <input>
                <soap:body parts='path' use='literal'/>
            </input>
            <output>
                <soap:body parts='response' use='literal'/>
            </output>
            <fault name='ProtocolException4'>
                <soap:fault name='ProtocolException4' use='literal'/>
            </fault>
        </operation>
        <operation name='move'>
            <map:java-operation name='move' signature='xxx'>
                <map:fault name='ProtocolException2' java-
type='com.ibm.vfds.ProtocolException'/>
            </map:java-operation>
            <soap:operation
soapAction='http://vfds.almaden.ibm.com/wsdl/com/ibm/vfds/VFDS#move?xxx'
               style='document'/>
            <input>
                <soap:body parts='src dest' use='literal'/>
            </input>
            <output>
                <soap:body parts='response' use='literal'/>
            </output>
            <fault name='ProtocolException2'>
                <soap:fault name='ProtocolException2' use='literal'/>
            </fault>
        </operation>
        <operation name='update'>
            <map:java-operation name='update' signature='xxx'>
                <map:fault name='ProtocolException2' java-
type='com.ibm.vfds.ProtocolException'/>
            </map:java-operation>
            <soap:operation
               soapAction='http://vfds.almaden.ibm.com/wsdl/com/ibm/vfds/VFDS#update?xxx'
               style='document'/>
            <input>
                <soap:body parts='original updated' use='literal'/>
            </input>
            <output>
                <soap:body parts='response' use='literal'/>
            </output>
            <fault name='ProtocolException2'>
                <soap:fault name='ProtocolException2' use='literal'/>
            </fault>
        </operation>
    </binding>
```

Author Information

Osamu Tatebe
Grid Technology Research Center, AIST
1-1-1 Umezono, Tsukuba
Ibaraki 3058568 Japan
o.tatebe@aist.go.jp

Manuel Pereira, Leo Luan, Ted Anderson
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120, USA
mpereira@us.ibm.com
leoluan@us.ibm.com
ota@us.ibm.com

Jane Xu
IBM Systems and Technology Group
5600 Cottle Road
San Jose, CA 95193, USA
jxu@us.ibm.com

Intellectual Property Statement

Full Copyright Notice

References

[1] Leo Luan and Ted Anderson, “Grid Namespace for Files”, GGF working draft, GGF8, 2003 https://forge.gridforum.org/projects/gfs-wg/document/Grid_Namespace_for_Files/en/1
[2] S. Shepler, et al., “Network File System (NFS) version 4 Protocol”, RFC3530, 2003