X/Open Preliminary Specification

DCE 1.1: Distributed File Service Specification

X/Open Company Ltd.

X/Open Preliminary Specification

DCE 1.1: Distributed File Service Specification

X/Open Document Number: P409

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@tog.org

Contents

List of Examples

List of Tables

Preface

The Open Group

In February 1996, X/Open and the Open Software Foundation (OSF) joined forces to become The Open Group, which represents one of the leading authorities in open systems. It is supported by most of the world's largest information systems suppliers, user organisations and software companies. By combining their complementary strengths — X/Open in providing specifications and its trade mark branding scheme, and OSF in facilitating collaboration among customers and system/software vendors toward the development of new open systems technologies — The Open Group is well positioned to assist vendors and users to develop and implement products which support adoption and proliferation of open systems.

Established in 1984, X/Open is an organisation dedicated to the identification, agreement and widescale adoption of Information Technology standards which provide compatibility (portability and interoperability) between software products, and so help users realise the business benefits of open information systems — lower costs, increased choice and greater flexibility. It achieves this by combining existing and emerging standards into an evolving set of integrated, high-value and usable open system specifications, which form a Common Applications Environment (CAE). It licenses a trade mark — called the X/Open Brand — for use on products which vendors have demonstrated are conformant to this CAE. The X/Open brand is recognised by users worldwide as the guarantee of compliance to open systems standards. X/Open is also responsible for the management of the UNIX trade mark on behalf of the industry.

Founded in 1988, the Open Software Foundation (OSF) delivers technology innovations in all areas of open systems, including interoperability, scalability, portability and usability. OSF is a worldwide coalition of vendors and customers in industry, government and academia, who work together to provide advanced open-systems technology solutions for use in a distributed computing environment. It runs programmes in collaborative research and development, to deliver vendor-neutral technology (software source code and supporting documentation) in key IT areas. These include OSF/1, Distributed Computing Environment (DCE), OSF/Motif and Common Desktop Environment (CDE).

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focused on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand trade mark. CAE specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  These specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification; rather, it is as stable as can be achieved, through applying X/Open's rigorous development and review procedures.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE specification. While the intent is to progress Preliminary specifications to corresponding CAE specifications, the ability to do so depends on consensus among X/Open members.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity. Although at the time of its publication, there may be an intention to progress the activity towards development of a Specification, Guide or Technical Study, the ability to do so depends on consensus among X/Open members.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.xopen.org** under Sales and Ordering.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.xopen.org** under Sales and Ordering.

This Document

This document is a Preliminary Specification (see above). It specifies Distributed File System (DFS) services, interface, protocols, encoding rules and the Interface Definition Language (IDL).

The purpose of this document is to provide a portability guide for DFS application programs and a conformance specification for DFS implementations, particularly those with extensions to the layer known as the Virtual File System (VFS). These extensions will be known as the Virtual File System additions (VFS+).

Structure

This document is organised into six parts.

Part 1, *Distributed File System Introduction*, describes this volume in detail, covering the DFS Interface Definition Language and data, DFS Versioning Scheme, DFS Client/Server and Token Manager Interfaces, the Access Control List Model used, the operations on aggregates (collections of filesets), the operations on volumes (filesets), and the vnode (VFS) interface and its operations. It contains material relevant to both application programmers and implementors.

Part 2, *DFS Interface Definition Language and Data*, describes the File Exporter IDL Declarations, Cache Manager IDL Declarations, and Common IDL Data used by DFS.

Part 3, *DFS Versioning Scheme*, describes the DFS RPC Versioning Scheme that permits providers to supply enhancements to the RPC functions of DFS. It includes data structures; the DFS versioning API; a set of examples including IDL files, Client, Server, and Manager applications; and a set of rules to follow for extending the DFS interface.

Part 4, *DFS Client/Server and Token Manager Interfaces*, provides a set of manpages for both the DCE DFS File Exporter and the Token Manager.

Part 5, *Access Control Lists (ACLs) in DFS*, provides an overview of Access Control Lists in DFS, their formats and the interface provided to manipulate them.

Part 6, *DCE DFS VFS+ Interface Specification*, specifies a portable Application Programmer's Interface (API). It contains material on the DFS aggregate operations, volume operations, and vnode (VFS) operations that are relevant to both application programmers and implementors.

Intended Audience

This document is written for DFS application programmers and developers of DFS implementations.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used for text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — variable names; for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — utility names
  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*( ).
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width font`.
- Variables within syntax statements are shown in *`italic fixed width font`*.

Trade Marks

OSF™ is a trade mark of The Open Software Foundation, Inc.

Transarc®, Encina® and AFS® are registered trade marks of Transarc Corporation.

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open® is a registered trade mark, and the ''X'' device is a trade mark, of X/Open Company Limited.

This list represents, as far as possible, those products that are trade marked. X/Open acknowledges that there may be other products that might be covered by trade mark protection and advises the reader to verify them independently.

Acknowledgements

Part 6, The DCE DFS VFS+ Interface Specification of the *DCE* 1.1: *Distributed* File *Service* Specification is based upon the specification of the extensions to the Virtual File System (VFS) from Hewlett-Packard known as the VFS+ Specification, which was originally written by Daryl Kinney and given to the Open Software Foundation in July, 1993, with subsequent updates by Hewlett-Packard.

Referenced Documents

The following documents are referenced in this specification:

DCE RPC
X/Open CAE Specification, August 1994, X/Open DCE: Remote Procedure Call (ISBN: 1-85912-041-5, C309).

OSF DCE DFS Administration Guide and Reference
for OSF DCE 1.1.

OSF DCE Administration Guide — Core Components
for OSF DCE 1.1.

X/Open Preliminary Specification

Part 1:

Distributed File System Introduction

Chapter 1

Introduction to the DFS Specification

This document specifies both portability and interoperability for the Distributed File System (DFS). The specification contains material directed at two audiences:

- It provides a portability guide for certain application programmers.
- It provides both portability and interoperability specifications for those who are implementing their own Virtual File System Extended (VFS+) or porting the DFS VFS+.

This document may be thought of as an implementation specification, covering both portability and interoperability, that contains within it a portability guide for those that are porting the DFS VFS+ LFS, as well as an implementation specification for those who choose not to port the DFS VFS+ portion.

The implementation specification for the DFS VFS+ LFS is contained in a manner that is intended to be self sufficient. It is mostly self describing, although it does reference other parts of the portability specification; for instance, the information on **ACL**s and data types.

Note: As a result of the intent to be self sufficient, there is some repetition of information, particularly **ACL** fields, and access algorithms. Thus, chapter 12 describes ACL information that was presented in chapters 8 and 9 from the point of view of an LFS (Local File System).

1.1 Portability

1.1.1 Overview

This part of the specification consists of chapters 2 through 7. These chapters describe all RPC interfaces used within the kernel resident components of the DCE DFS distributed file system, namely, the file exporter, the cache manager, and the rules for adding new or modified RPC functions to the file exporter. Chapter 6 on page 99 describes the RPC interface to the file exporter, which is used by cache managers to access and manipulate files. Chapter 7 on page 145 describes the RPC interface the cache manager exports in order to handle token management and other administrative inquiries from the file exporter. Chapter 5 on page 69 describes the mechanism for adding new or modified RPC functions to the file exporter.

1.1.2 IDLs, Data Types, Constants and Flags

In addition, a set of chapters provides related information such as data types, definitions of constants and flags. Chapter 2 on page 13 describes the IDL definitions and related constants and flags associated with the *DCE DFS File Exporter Interface*. In addition, Chapter 3 on page 51 describes the same kinds of information associated with the *Cache Manager Service Interface* . There is additional data type, flag, and constant information relating to both interfaces in Chapter 4 on page 57, titled *Common IDL Data*.

1.2 Document Organization

The **DCE 1.1: Distributed File Service** specification is organised as described in the following. This organisation follows the pattern that the first part is at the lowest level (RPC) and proceeds to higher levels, such as Access Control Lists, and finally, the extensions to the VFS known as the VFS+ Interface.

1.2.1 RPC Interfaces for DCE DFS

Parts 2 through 4 of the **DCE 1.1: Distributed File Service** specification describe the exported interface used within the kernel resident portion of the **DCE DFS** distributed file system. This interface description includes all RPC interfaces used within the kernel and the rules for adding new or modified RPC functions to the file exporter (sometimes called the protocol exporter) and consists of:

- Part 2, *DFS Interface Definition Language and Data*, consisting of Chapter 2 on page 13, the *File Exporter IDL Declarations* for both clients and servers, Chapter 3 on page 51, *Cache Manager IDL Declarations*, and Chapter 4 on page 57, *Common IDL Data* shared by both the File Exporter and the Cache Manager
- Part 3, *DFS Versioning Scheme*, consisting of Chapter 5 on page 69, *DFS RPC Versioning Scheme*, which provides a mechanism and rules for extending the DFS interface at the RPC level
- Part 4, *DFS Client/Server and Token Manager Interfaces*, consisting of Chapter 6 on page 99, *DCE DFS File Exporter Interface* and Chapter 7 on page 145, *Cache Manager Service Interface* describing the manpages for these interfaces.

1.2.2 Access Control Lists (ACLs) in DFS

Part 5 of the **DCE 1.1: Distributed File Service** specification describes the Acccess Control Lists used in **DCE DFS**. It consists of:

- Chapter 8 on page 155, *Access Control List Overview*, describing the *ACL* types, and rules governing entries in them for filesystem **ACL**s, including delegation, as well as access checking algorithms for principals and delegation
- Chapter 9 on page 165, *ACL Storage Format*, describing the structured formats used for **ACL**s in memory (which differs from the unstructured byte-stream format used on disk)
- Chapter 10 on page 171, *Acess Control List Interface Functions*, describing the functions available in the **ACL** interface by which the **ACL**s can be manipulated. The functions are used for both in-kernel filesystem **ACL**s and Administrative Lists **ACL**s, which are used in user space.

In addition, there are two appendixes in Part 5:

- Appendix A on page 221, *Mapping DFS ACLs to UNIX mode bits*, which discusses the relationship between **ACL**s and UNIX mode bits
- Appendix B on page 229, *Access Control List Package Error List*, which lists the return values that can occur in this interface.

1.2.3 The DCE DFS VFS+ Interface Specification

Part 6 of the **DCE 1.1: Distributed File Service** specification describes the **DCE DFS** VFS+ interface specification for a **DCE LFS** . It consists of chapters 11 through 17, and 10 appendixes, organized as follows:

- Chapter 11 on page 233, *DCE DFS VFS+ Interface Introduction*, consists of an overview of the interfaces for an LFS, the organization of the VFS+ switch for extending the VFS interface, the fileset registry, fileset and vnode interfaces, and LFS references back into DFS.

  It has an appendix, Appendix C, *Components of a Typical VFS+ Package*, which provides information on items such as naming conventions in this interface extension, example wrapper routines called "glue" functions, that can be used to transparently extend the VFS vector so that the extensions can be invoked, and extensions to the **vnode** attributes.

- Chapter 12 on page 251, *The DCE DFS ACL Model for an LFS*, discusses **ACL**s, security and protection checking as they relate to a DFS physical file system (known as a local file system, or LFS).

- Chapter 13 on page 267, *VFS+ Data Types*, describes the data types used in a **DCE LFS**, including primitive data types, as well as data types for aggregates, filesets and vnodes.

- Chapter 14 on page 303, *Aggregate Operations Interface*, describes the portion of the VFS+ interface which allows operations on aggregates, the **DCE DFS** data containers which house filesets, as introduced in Section 11.3 on page 235. The aggregate array functions found in the **struct aggrops** structure, are also described in this chapter.

  Additional information for the aggregate interface is found in Appendix D through Appendix F.

- Chapter 15 on page 325, *Fileset (Volume) Operations Interface*, describes the facilities provided for the VFS+ Fileset Operations, including fileset types, clones, fileset and clone requirements, fileset indices, LFS modification of fileset status, zero link count files, quotas, anode generation numbers, file identifiers, vnode to LFS association, and lastly, fileset and *fileset registry* operations.

  Appendix G through Appendix L provide additional information about filesets, such as values for opening a fileset, getting status, and creating a fileset.

- Chapter 16 on page 415, *VFS (Vnode) Interface and Operations*, describes the enhanced **vnode** operations vector, converted **vnode**s, synchronization between **vnode** and **fileset** operations, as well as the functions provided for operating on **vnode**s. It also includes the interface description for the extended **vnode** interface.

- Chapter 17 on page 469, *DCE DFS VFS+ Extended Credential Package*, describes the interface used for authentication within the VFS+ layer. It presents an extensible and upwards-compatible interpretation of the VFS UNIX **credential** structure.

  It describes what is essentially a subroutine library allowing VFS functions to interpret the VFS credentials structure in a more general manner.

1.3 Terminology

This is a list of commonly used terms in DFS.

ACL
Short for *access control list*; a description of which users (*principals*) and groups (groups of principals) can use a file or directory and what types of uses are permitted (*rights*). Every file can have an **ACL**. Directories have three **ACL** types; one governs operations on the directory itself, a second provides an initial **ACD** for files created in the directory, and a third provides the **ACL** for sub-directories created within the directory. These **ACL** types are known as the *initial object*, *object*, and *initial container* **ACL**s, respectively.

Aggregate
An expanse of physical disk which is managed by a DFS LFS. It is similar in context to a UNIX partition and is identified by both a device number and a device file. In many cases, the storage provided to an aggregate will be by a Logical Volume Manager (LVM).

Anode
A 252-byte area of disk which is contiguous. It is the primitive data structure in the **DCE LFS** file system. All objects in LFS reside in a *container*; a logical variable-sized object, sometimes called a "bucket", that can hold data. An *anode* stores status and disk addressing information -- called meta-data as well as information describing user files.

Anode Index
An index which can be used to refer to the *anode*. This index is used in many situations by an LFS to name and locate *anode*s. A *fileset* is implemented as an array of *anode*s.

Backing Anode
This is an *anode* whose storage is shared with other *COW anodes* and whose contents are fixed. Backing anodes are **read-only**. Multiple COW anodes can refer to the same backing anodes.

Block
The larger unit in a system, for allocating disk space to *containers* and doing I/O and buffering. Block size must be a power of two in **DCE DFS**.

Cell
A collection of DCE machines administered as a single entity. In this context, the key point is that a cell is serviced by a single registry with a single, consistent set of user and group identifiers.

Clone
A loosely-used term to describe a *fileset* which is related to another *fileset*. Sometimes it refers to a read-write (R/W) fileset and sometimes to a *replica*. The term clone is also applied to the operation on individual anodes during the fileset clone process. A R/W fileset is comprised of *COW anodes* each of which was "cloned" from the *backing anode* in the corresponding *anode index* in a read-only snapshot of a fileset. A snapshot is a consistent, read-only version of a fileset.

COW Anode
Also known as a Copy-on-Write anode. This is an *anode* whose storage is shared with a *backing anode* and whose contents can be changed. Modifications to a COW anode are made by allocating new storage, initializing the storage from the backing anode, then making the desired modifications. Both containers (anodes) must reside in the same *aggregate*.

Container
An open-ended, ordered collection of disk space; the basic storage unit for data on a disk in a **DCE LFS**. It is variable-sized and is represented by an anode. A container can contain

other containers. For instance, an *aggregate* is a container that holds other containers, such as filesets and also representations of free space, or a log. A *fileset*, in turn, is a container holding files.

DFS
The DCE Distributed File System.

EFS
The optional portion of DFS which deals with extended fileset operations (cloning, backup, replicas, and so on).

Xcred
Also known as an *extended credential*. A credential structure, like that used in the UNIX kernel, but augmented in various ways - including a *realm* identifier; it distinguishes a *local group* from a *foreign group*. The xcred credential identifies a *principal*.

File
A *container* used to implement UNIX files, directories and symbolic links. It has a status area in which the standard information returned by **stat(2)** is stored. The status area also contains pointers to information for access control.

Fileset
A set of related files, connected via a sub-tree with a single root point, that is administered as an entity by DFS. Although filesets cannot span aggregates, a single aggregate can hold many filesets. Filesets which have an associated maximum size or quota, grow independently of each other and compete for space within their aggregate. Individual filesets can be mounted locally, backed up and restored, cloned or transparently moved to another aggregate.

The terms fileset and volume are used interchangeably, although the term volume is becoming archaic.

Fragment
The smaller unit in a system, for allocating disk space to *containers*. The number of fragments in a *block* must be a power of two.

Group
A membership list, identified by a **uuid_t**, which contains a list of principals (by **uuid_t**) which are members of that group. Again, in a non-DCE environment, they are identified by a 32-bit (at best) **gid**.

LFS
Local File System: a physical file system, which in the context of this document, provides DFS semantics.

Episode is the name of the LFS that Transarc Corporation provides as part of the DFS EFS package.

Magic Cookie
A piece of data, passed back and forth between two functions which is opaque to the higher-level function. It is often used as a token of progress in an iteration.

PAC
Privilege Attribute Certificate. A data structure, generated by the DCE Privilege Server, that contains the identity of a DCE authenticated principal. To a first approximation, it contains the principal's **uuid_t** along with the **uuid_t** of each group of which it is a member.

Principal
An entity that is interacting with DFS. Although it usually refers to a person, it could as easily correspond to a DCE server or a computer within a DCE cell. In the DCE environment, principals are identified by a **uuid_t**. In a non-DCE, standard UNIX, environment, they are identified by a 32- bit (at best) **uid**.

Quota
Maximum space allowed to be allocated to a *fileset*, independent of how much space remains in the *aggregate*.

Realm
Equivalent to a Cell. In security DCE security discussions, the term realm is typically used instead of cell.

Replica
A read-only *fileset* whose contents match that of a consistent read-only version of a fileset on a different *aggregate*. A replica differs from a snapshot in that the snapshot is really linked to the primary fileset by a copy on write relationship and shares storage with it, while a replica is an independent copy of a snapshot in another aggregate.

Rights
The classes of operations whose access can be separately controlled. For traditional UNIX file systems, these are designated **rwx**. A **DCE LFS** has more rights for directories which allow separate control of *insert* (**i**) and *delete* (**d**) operations. In addition, the operations traditionally reserved to a file's owner are under the control of right (**c**).

Token
A data structure passed between a DFS file server and client (user of the file server) that guarantees to the client that it has permission to perform the operation described in the token. The DFS file servers issue and revoke tokens granted to clients, and clients follow this convention. Tokens are used to provide a coherent view of data across all DFS clients as if they were in a single system.

UFS
The **UNIX** File System. The name of the file system originally provided by vendors of **UNIX** systems. In this file system the *inode* is a container for user file data and directory data. It does not contain metadata such as found in an *anode*.

Uniquifier
Synonym used interchangeably with *unique*. It distinguishes between different uses (over time) of an entity. It is a number incremented each time an entity is created, and can be thought of as a generation number. Its primary use is in the construction of file identifiers. In a *fileset*, it is a number incremented each time a file is created, which is used to distinguish between instantiations of a file using the same *anode index*. In a **vnode**, for each new use, a new *uniquifier* is used in the **afsFid**, the DFS file identifier.

VFS+ Interface
A VFS interface, including **vnode** operations and **VFS** operations, extended by adding **vnode** operations for portability, for **Episode**, and for other **DCE LFS** file systems. An extension to this interface usually includes **fileset** operations and **aggregate** operations.

In addition, for portability, this interface requires portability "glue" or wrapper functions to adapt the *vnode* operations to a particular kernel. This interface also requires token "glue" in order to enable a file system to be exported.

1.4 Conformance Requirements

To conform to this document, implementations must meet the following requirements:

- Implementations of file servers must support the interfaces defined in Chapter 6 on page 99.
- Implementations of file servers must support the AFS4Int interface defined in Chapter 2 on page 13.
- Implementations of file servers must support the common data structures defined in Chapter 4 on page 57.
- Implementations of cache managers must support the interfaces defined in Chapter 7 on page 145.
- Implementations of cache managers must support the TKN4Int interface defined in Chapter 3 on page 51.
- Implementations of cache managers must support the common data structures defined in Chapter 4 on page 57.
- Implementations of Local File Systems must support the interfaces and data structures defined in Part 6 on page 237.
- Implementations that extend either the file exporter interface defined in Chapter 2 on page 13 or the token manager interface defined in Chapter 3 on page 51 (or both) must conform to the mechanism defined in Chapter 5 on page 69.

X/Open Preliminary Specification

Part 2:

DFS Interface Definition Language and Data

Chapter 2

File Exporter IDL Declarations

This chapter specifies the RPC interfaces used by the AFS (4.0) Client and Cache Manager to the AFS (4.0) Server and File Exporter Interface.

The File Exporter interface consists of a set of structures, mask values and flags contained in an IDL file with a provider version number of 1 to indicate that it is an original interface. In addition, each RPC defined in this structure also has a provider version number of 1, indicating that it also is an original RPC. See Chapter 5 on page 69 for information on versioning which will explain in more detail about provider version numbers.

2.1 The File Exporter Interface

The File Exporter Interface is contained within the interface, **AFS4Int**, which starts out as shown below and contains all the items shown in this chapter exclusive of the definitions of the parameters in the RPC calls which are included from Chapter 6 on page 99 and the common data definitions in the configuration file **common_data.idl** which are contained in Chapter 4 on page 57 for clarity. There are twenty seven remote procedure calls defined that the client and cache manager can use in this interface which is defined as provider version 1. Chapter 5 on page 69 provides information for adding new interfaces, or new remote procedure calls to this interface.

```
[
uuid(4d37f2dd-ed93-0000-02c0-37cf1e000000),
version(4.0)
/* provider_version(1) */
]

interface AFS4Int
{

import "dcedfs/common_data.idl";

 .
 .
 .
/* } at end of this chapter concludes this interface. */
```

2.2 Mask Values for afsStoreStatus Structure

```
const unsigned32 AFS_SETMODTIME = 1;
const unsigned32 AFS_SETOWNER = 2;
const unsigned32 AFS_SETGROUP = 4;
const unsigned32 AFS_SETMODE = 8;
const unsigned32 AFS_SETACCESSTIME = 0x10;
const unsigned32 AFS_SETCHANGETIME = 0x20;
                              /* don't allow this for now */
const unsigned32 AFS_SETLENGTH = 0x40;
const unsigned32 AFS_SETTYPEUUID = 0x80;
const unsigned32 AFS_SETDEVNUM = 0x100;
const unsigned32 AFS_SETMODEXACT = 0x200;
                              /* allow setting back */
const unsigned32 AFS_SETTRUNCLENGTH = 0x400;
                              /* truncate to this size first */
const unsigned32 AFS_SETCLIENTSPARE = 0x800;
                              /* set clientSpare1 */
```

2.3 AFS_GetTime Sync Constants

These are distance and dispersion constants.

```
const unsigned32 AFS_SYNCUNSYNC = 0x7ffffff0;
const unsigned32 AFS_SYNCINITIAL = 0x7ffffff1;
```

2.4 Flag Parameters for ACL Type Parameter

These are flag parameters for the high order 16 bits of the getacl and setacl **ACL** type parameter.

```
const unsigned32 AFS_ACLFLAG_COPY = 1;
    /* copy the ACL from the specified fid */
```

2.5 Definitions for Flag Used in Getting a Token

These are the definitions for the flage used in getting a token for each afs4int call.

```
const unsigned32 AFS_FLAG_RETURNTOKEN = 1;
    /* Return tokens to caller not to the token manager */

const unsigned32 AFS_FLAG_TOKENJUMPQUEUE = 2;
    /* in AFS_GetToken, call tkset_AddTokenSet with
       TKSET_ATS_WANTJUMPQUEUE flag */

const unsigned32 AFS_FLAG_SKIPTOKEN = 4;
    /* don't obtain tokens, client already  has them */

const unsigned32 AFS_FLAG_NOOPTIMISM = 0x8;
    /* return only the exact asking token, no "optimism" tokens */

const unsigned32 AFS_FLAG_TOKENID = 0x10;
```

```
    /* to say get-the-token-with-this-ID */

const unsigned32 AFS_FLAG_RETURNBLOCKER = 0x20;
    /* To return blocker's info for Sys V record lock */

const unsigned32 AFS_FLAG_ASYNCGRANT = 0x40;
    /* permit async grant in AFS_GetToken if can't get now */

const unsigned32 AFS_FLAG_NOREVOKE = 0x80;
    /* Ask server not to revoke other's token, if there is a
       conflict. Just return a failure  */
```

2.6 Definitions for Flag in Token Recovery (TSR)

These are the definitions for the flag used in performing token recovery (TSR) operations.

```
const unsigned32 AFS_FLAG_MOVE_REESTABLISH = 0x100;
    /* to say you're reestablishing tokens after a move */

const unsigned32 AFS_FLAG_SERVER_REESTABLISH = 0x200;
    /* to say you're reestablishing tokens after a server crash */

const unsigned32 AFS_FLAG_NO_NEW_EPOCH = 0x400;
    /* to prevent advancing the indicated token into a new epoch */

const unsigned32 AFS_FLAG_MOVE_SOURCE_OK = 0x800;
```

2.7 Definitions for Flag in Fileset Operations

These are the definitions for the flag used in performing fileset operations.

```
const unsigned32 AFS_FLAG_SYNC = 0x1000;
    /* fsync all modified files after operation */

const unsigned32 AFS_FLAG_ZERO = 0x2000;
    /* Data is all zero instead of in pipe */

const unsigned32 AFS_FLAG_SKIPSTATUS = 0x4000;
    /* Dont bother filling Out file status */

const unsigned32 AFS_FLAG_FORCEREVOCATIONS = 0x8000;
    /* In AFS_GetToken, insist on getting this token, and make conflicting
       tokens be revoked with the AFS_REVOKE_FORCED_REVOCATION flag. */

const unsigned32 AFS_FLAG_FORCEVOLQUIESCE = 0x10000;
    /* Ensure that this whole-fileset token blocks all per-file tokens. */
```

2.8 Definitions for Flags in AFS_SetParams( )

These are the definitions for the flags used only in calling *AFS_SetParams*( ).

```
const unsigned32 AFS_PARAM_RESET_CONN   = 0x1;
const unsigned32 AFS_PARAM_TSR_COMPLETE = 0x2;
```

2.9 Definitions for Flags in AFS_SetContext( )

```
const unsigned32 AFS_FLAG_SEC_SERVICE   = 0x1;
    /* client needs a secondary service */

const unsigned32 AFS_FLAG_CONTEXT_NEW_IF = 0x2;
    /* this is a new-interface call, with different DOWN semantics */

const unsigned32 AFS_FLAG_CONTEXT_DO_RESET = 0x4;
    /* client will reset all token state itself, to a
       new-interface server */
const unsigned32 AFS_FLAG_CONTEXT_NEW_ACL_IF = 0x8;
    /* client understands new delegation acl entry types */
```

2.10 Definitions for Client-only Attribute Flags

```
const unsigned32 AFS_CLIENTATTR_HIDDEN = 0x1;
const unsigned32 AFS_CLIENTATTR_SYSTEM = 0x2;
const unsigned32 AFS_CLIENTATTR_ARCHIVE = 0x4;
const unsigned32 AFS_CLIENTATTR_READONLY = 0x8;
```

2.11 Data Types

The following data types are used to define structures used by the remote procedure calls in provider version 1 of the File Exporter Interface.

2.11.1 Define Generic Network Address Information

```
typedef struct afsNetData {
    afsNetAddr sockAddr;
    NameString_t principalName;
} afsNetData;
```

2.11.2 Define afsVolSync Structure for Tracking Replicas

This is the structure by which replicas are tracked.

```
typedef struct afsVolSync {
    afsHyper VolID;
    afsHyper VV;          /* volume's version */
    unsigned32 VVAge;     /* age, in seconds, of the knowledge that the
                             given VolVers is current */
    unsigned32 VVPingAge; /* age, in seconds, of the last probe from
                             the callee (the secondary) to the primary */
    unsigned32 vv_spare1;
```

```
    unsigned32 vv_spare2;
} afsVolSync;
```

2.11.3 Define the afsFetchStatus Structure

```
typedef struct afsFetchStatus {
    unsigned32      interfaceVersion;
    unsigned32      fileType;
    unsigned32      linkCount;
    afsHyper        length;
    afsHyper        dataVersion;
    unsigned32      author;
    unsigned32      owner;
    unsigned32      group;
    unsigned32      callerAccess;
    unsigned32      anonymousAccess;
    unsigned32      aclExpirationTime;
    unsigned32      mode;
    unsigned32      parentVnode;
    unsigned32      parentUnique;
    afsTimeval      modTime;
    afsTimeval      changeTime;
    afsTimeval      accessTime;
    afsTimeval      serverModTime;
    afsUUID         typeUUID;
    afsUUID         objectUUID;
    unsigned32      deviceNumber;
    unsigned32      blocksUsed;
    unsigned32      clientSpare1;      /* client-only attrs */
    unsigned32      deviceNumberHighBits;
    unsigned32      spare0;
    unsigned32      spare1;
    unsigned32      spare2;
    unsigned32      spare3;
    unsigned32      spare4;
    unsigned32      spare5;
    unsigned32      spare6;
} afsFetchStatus;
```

2.11.4 Define the afsStoreStatus Structure

```
typedef struct afsStoreStatus {
    unsigned32      mask;
    afsTimeval      modTime;
    afsTimeval      accessTime;
    afsTimeval      changeTime;
    unsigned32      owner;
    unsigned32      group;
    unsigned32      mode;
    afsHyper        truncLength;       /* applied first */
    afsHyper        length;
    afsUUID         typeUUID;
    unsigned32      deviceType;        /* character or block */
```

```
    unsigned32      deviceNumber;
    unsigned32      cmask;
    unsigned32      clientSpare1;      /* client-only attrs */
    unsigned32      deviceNumberHighBits;
    unsigned32      spare1;
    unsigned32      spare2;
    unsigned32      spare3;
    unsigned32      spare4;
    unsigned32      spare5;
    unsigned32      spare6;
} afsStoreStatus;
```

2.11.5 Structure for Physical File System Type

```
typedef struct afsDisk {/* used by Statistics */
    unsigned32      BlocksAvailable;
    unsigned32      TotalBlocks;
    unsigned32      spare1;
    unsigned32      spare2;
    unsigned32      spare3;
    afsDiskName     Name;
} afsDisk;
```

2.11.6 Structure for Statistics

```
typedef struct afsStatistics {
    unsigned32      CurrentMsgNumber;
    unsigned32      OldestMsgNumber;
    unsigned32      CurrentTime;
    unsigned32      BootTime;
    unsigned32      StartTime;
    unsigned32      CurrentConnections;
    unsigned32      TotalAfsCalls;
    unsigned32      TotalFetchs;
    unsigned32      FetchDatas;
    unsigned32      FetchedBytes;
    unsigned32      HighFetchedBytes;
    unsigned32      FetchDataRate;
    unsigned32      TotalStores;
    unsigned32      StoreDatas;
    unsigned32      StoredBytes;
    unsigned32      HighStoredBytes;
    unsigned32      StoreDataRate;
    unsigned32      TotalRPCBytesSent;
    unsigned32      HighTotalRPCBytesSent;
    unsigned32      TotalRPCBytesReceived;
    unsigned32      HighTotalRPCBytesReceived;
    unsigned32      TotalRPCPacketsSent;
    unsigned32      TotalRPCPacketsReceived;
    unsigned32      TotalRPCPacketsLost;
    unsigned32      TotalRPCBogusPackets;
    unsigned32      SystemCPU;
    unsigned32      UserCPU;
```

```
        unsigned32      NiceCPU;
        unsigned32      IdleCPU;
        unsigned32      TotalIO;
        unsigned32      ActiveVM;
        unsigned32      TotalVM;
        unsigned32      EtherNetTotalErrors;
        unsigned32      EtherNetTotalWrites;
        unsigned32      EtherNetTotalInterupts;
        unsigned32      EtherNetGoodReads;
        unsigned32      EtherNetTotalBytesWritten;
        unsigned32      EtherNetTotalBytesRead;
        unsigned32      ProcessSize;
        unsigned32      WorkStations;
        unsigned32      ActiveWorkStations;
        unsigned32      Spare1;
        unsigned32      Spare2;
        unsigned32      Spare3;
        unsigned32      Spare4;
        unsigned32      Spare5;
        unsigned32      Spare6;
        unsigned32      Spare7;
        unsigned32      Spare8;
        afsDisk         Disk1;
        afsDisk         Disk2;
        afsDisk         Disk3;
        afsDisk         Disk4;
        afsDisk         Disk5;
        afsDisk         Disk6;
        afsDisk         Disk7;
        afsDisk         Disk8;
        afsDisk         Disk9;
        afsDisk         Disk10;
        afsDisk         Disk11;
        afsDisk         Disk12;
        afsDisk         Disk13;
        afsDisk         Disk14;
        afsDisk         Disk15;
        afsDisk         Disk16;
    } afsStatistics;
```

2.12 Various Bulk Data Type Definitions for RPC Operations

2.12.1 The afsFidExp Structure

This structure is a parameter for BulkKeepAlive, so that the PX doesn't have to look up the lifetimes.

```
typedef struct afsFidExp {
    afsFid fid;
    unsigned32 keepAliveTime;
} afsFidExp;
```

A special value for keepAliveTime to force a re-check of the reclaimDally value.

```
const unsigned32 AFS_KA_TIME_RECHECK = 1;
```

2.12.2 The afsBulkFEX Structure

```
typedef struct afsBulkFEX {
    unsigned32 afsBulkFEX_len;
    [length_is(afsBulkFEX_len)] afsFidExp afsBulkFEX_val[AFS_BULKMAX];
} afsBulkFEX;
```

2.12.3 The afsACL Structure

```
typedef struct afsACL {
    unsigned32 afsACL_len;
    [length_is(afsACL_len)] byte afsACL_val[AFS_ACLMAX];
} afsACL;
```

2.12.4 The afsQuota Structure

```
typedef struct afsQuota {
    unsigned32 afsQuotaType;    /* BSD, Episode, etc */
    unsigned32 afsQuotaOp;      /* set or get */
    unsigned32 afsQuota_len;
    [length_is(afsQuota_len)] unsigned32 afsQuota_val[AFS_BULKMAX];
} afsQuota;
```

2.12.5 The afsBulkVVs Structure

```
typedef struct afsBulkVVs {
    unsigned32 afsBulkVVs_len;
    [length_is(afsBulkVVs_len)] afsVolSync afsBulkVVs_val[AFS_BULKMAX];
} afsBulkVVs;
```

2.12.6 The afsBulkVolIDs Structure

```
typedef struct afsBulkVolIDs {
    unsigned32 afsBulkVolIDs_len;
    [length_is(afsBulkVolIDs_len)] afsHyper afsBulkVolIDs_val[AFS_BULKMAX];
} afsBulkVolIDs;
```

2.12.7 The afsBulkStats Structure

```
typedef struct afsBulkStats {
    unsigned32 afsBulkStats_len;
    [length_is(afsBulkStats_len)] afsFetchStatus afsBulkStats_val[AFS_BULKMAX];
} afsBulkStats;
```

2.13 Definitions of File Server Exported Operations

Note: The Opcode of each operation is implicitly assigned based on the order of where the operation is placed. The recommendation is to always place the new operation at the end of this list.

2.13.1 AFS_SetContext

```
error_status_t AFS_SetContext
(/* provider_version(1) */
   [in]  handle_t          h,
   [in]  unsigned32        epochTime,
   [in]  afsNetData        *callbackAddr,
   [in]  unsigned32        Flags,
   [in]  afsUUID           *secObjectID,
   [in]  unsigned32        clientSizesAttrs,
   [in]  unsigned32        parm7
);
```

AFS_SetContext parameters are:

h	The RPC binding handle.
epochTime	The restart time of the DFS client.
callbackAddr	The RPC endpoint of the client, for token revocation purposes.
Flags	If 0 then the call only defines a primary interface UUID. If the flag, AFS_FLAG_SEC_SERVICE, is set then a a secondary interface is defined and is stored in the parameter, *secObjectID*.
secObjectID	When the AFS_FLAG_SEC_SERVICE flag is set, this parameter holds the secondary interface UUID.
clientSizeAttrs	For 64/32-bit compatibility. Through DCE1.1, this is a spare parameter, *parm6*.
parm7	A spare parameter.

2.13.2 AFS_LookupRoot

```
error_status_t AFS_LookupRoot
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *InFidp,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsFid            *OutFidp,
   [out]  afsFetchStatus    *OutFidStatusp,
   [out]  afsToken          *OutTokenp,
   [out]  afsVolSync        *Syncp
);
```

AFS_LookupRoot parameters are:

h	The RPC binding handle.
InFidp	The file identifier specifying the fileset whose root directory will be retrieved from the file server.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier describing the root directory of the selected fileset.
OutFidStatusp	Returns the status of that directory after the current operation.
OutTokenp	The promise the file server returns to the cache manager about the provided data; this is only returned if the file resides in a read/write fileset.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.3 AFS_FetchData

```
error_status_t AFS_FetchData
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *Fidp,
   [in]   afsHyper              *minVVp,
   [in]   afsHyper              *Position,
   [in]   signed32              Length,
   [in]   unsigned32            Flags,
   [out]  afsFetchStatus        *OutStatusp,
   [out]  afsToken              *OutTokenp,
   [out]  afsVolSync            *Syncp,
   [out]  pipe_t                *fetchStream
);
```

AFS_FetchData parameters are:

h The RPC binding handle.

Fidp The file identifier specifying the file whose contents will be retrieved from the file server.

minVVp The minimum-acceptable version number on the fileset containing this file or directory.

Position Specifies the first byte to be fetched by this call with zero (0) being the first byte in the file.

Length Specifies the number of bytes desired with the value 0xFFFFFFFF indicating the entire file contents.

Flags Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

OutStatusp Returns the status of the file after the current operation.

OutTokenp The promise the file server returns to the cache manager about the provided data; this is only returned if the file resides in a read/write fileset.

Syncp The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

fetchStream The character pipe parameter returning the data from the file. (See **X/Open DCE: Remote Procedure Call** specification for implementation details.)

2.13.4 AFS_FetchACL

```
error_status_t AFS_FetchACL
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *Fidp,
   [in]   unsigned32        aclType,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsACL            *AccessListp,
   [out]  afsFetchStatus    *OutStatusp,
   [out]  afsVolSync        *Syncp
);
```

AFS_FetchACL parameters are:

h	The RPC binding handle.
Fidp	The file identifier specifying the file or directory whose access control list (ACL) will be retrieved from the file server.
aclType	The type of the access list being modified. One of VNX_ACL_REGULAR_ACL, VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL.
minVVp	The minimum-acceptable version number on the fileset containing this ACL.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
AccessListp	The access control list returned by the file server for the specified file or directory.
OutStatusp	Returns the current status of the file or directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.5 AFS_FetchStatus

```
error_status_t AFS_FetchStatus
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *Fidp,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsFetchStatus    *OutStatusp,
   [out]  afsToken          OutTokenp,
   [out]  afsVolSync        *Syncp
);
```

AFS_FetchStatus parameters are:

h	The RPC binding handle.
Fidp	The file identifier specifying the file or directory whose status information will be retrieved from the file server.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutStatusp	Returns the current status of the file or directory.
OutTokenp	Returns a token from the file server reflecting guarantees granted by the file server.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.6 AFS_StoreData

```
error_status_t AFS_StoreData
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *Fidp,
   [in]   afsStoreStatus        *InStatusp,
   [in]   afsHyper              *Position,
   [in]   signed32              Length,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [in]   pipe_t                *storeStream,
   [out]  afsFetchStatus        *OutStatusp,
   [out]  afsVolSync            *Syncp
);
```

AFS_StoreData parameters are:

h	The RPC binding handle.
Fidp	The file identifier specifying the file whose status information will be updated from the file server.
InStatusp	The new status information that should be recorded for this file.
Position	Represents the position of the first byte of the data block.
Length	Represents the total length of the transferred data block.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
storeStream	The data stream containing the file updates. (See the **X/Open DCE: Remote Procedure Call** specification for implementation details.)
OutStatusp	Returns the status of the file.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.7 AFS_StoreACL

```
error_status_t AFS_StoreACL
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *Fidp,
   [in]   afsACL                *AccessListp,
   [in]   unsigned32            aclType,
   [in]   afsFid                *aclFidp,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFetchStatus        *OutStatusp,
   [out]  afsVolSync            *Syncp
);
```

AFS_StoreACL parameters are:

h The RPC binding handle.

Fidp The file identifier specifying the file or directory whose access control information will be updated from the file server.

AccessListp The access control list sent to the file server for the specified file or directory.

aclType The type of access control list being modified (VNX_ACL_REGULAR_ACL, VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL, in the low-order 8 bits. In the next-higher-order 8 bits, the type of access control list being copied from the file described by *aclFidp*.

aclFidp The file identifier specifying the file or directory whose access control information will be copied.

minVVp The minimum-acceptable version number on the fileset containing this file or directory.

Flags Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

OutStatusp Returns the updated file or directory status.

Syncp The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.8 AFS_StoreStatus

```
error_status_t AFS_StoreStatus
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *Fidp,
   [in]   afsStoreStatus    *InStatusp,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsFetchStatus    *OutStatusp,
   [out]  afsVolSync        *Syncp
);
```

AFS_StoreStatus parameters are:

h	The RPC binding handle.
Fidp	The file identifier specifying the file or directory whose access control information will be updated from the file server.
InStatusp	Contains the new status information for the specified file or directory.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutStatusp	Contains the updated status information for the file or directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.9 AFS_RemoveFile

```
error_status_t AFS_RemoveFile
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *DirFidp,
   [in]   afsFidTaggedName      *Namep,
   [in]   afsHyper              *returnTokenIDp,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFetchStatus        *OutDirStatusp,
   [out]  afsFetchStatus        *OutFileStatusp,
   [out]  afsFid                *OutFileFidp,
   [out]  afsVolSync            *Syncp
);
```

AFS_RemoveFile parameters are:

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory from which to remove the file.
Namep	The complex name of the file to delete.
returnTokenIDp	A token ID being returned, if any.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutDirStatusp	Contains the updated directory status information.
OutFileStatusp	Contains the updated file status information.
OutFileFidp	The file id of the file which was actually deleted.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.10 AFS_CreateFile

```
error_status_t AFS_CreateFile
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *DirFidp,
   [in]   afsTaggedName         *Namep,
   [in]   afsStoreStatus        *InStatusp,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFid                *OutFidp,
   [out]  afsFetchStatus        *OutFidStatusp,
   [out]  afsFetchStatus        *OutDirStatusp,
   [out]  afsToken              *OutTokenp,
   [out]  afsVolSync            *Syncp
);
```

AFS_CreateFile parameters are:

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which to create the requested file.
Namep	The character string name of the file to create.
InStatusp	Specifies the initial status fields for the new file.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier of the newly created file.
OutFidStatusp	The status fields for the newly created file.
OutDirStatusp	The status information of the specified directory.
OutTokenp	A new token granted against the new file.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.11 AFS_Rename

```
error_status_t AFS_Rename
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *OldDirFidp,
   [in]   afsFidTaggedName      *OldNamep,
   [in]   afsFid                *NewDirFidp,
   [in]   afsFidTaggedName      *NewNamep,
   [in]   afsHyper              *returnTokenIDp,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFetchStatus        *OutOldDirStatusp,
   [out]  afsFetchStatus        *OutNewDirStatusp,
   [out]  afsFid                *OutOldFileFidp,
   [out]  afsFetchStatus        *OutOldFileStatusp,
   [out]  afsFid                *OutNewFileFidp,
   [out]  afsFetchStatus        *OutNewFileStatusp,
   [out]  afsVolSync            *Syncp
);
```

AFS_Rename parameters are:

h The RPC binding handle.

OldDirFidp The file identifier specifying the directory in which the file is currently located.

OldNamep The complex name of the file or directory to rename.

NewDirFidp The file identifier specifying the directory into which the file is to be moved.

NewNamep The complex name of the file or directory after it is moved.

returnTokenIDP A token ID being returned, if any.

minVVp The minimum-acceptable version number on the fileset containing this file or directory.

Flags Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

OutOldDirStatusp
The status information of the old directory (the one from which the file is being moved) upon termination of the call.

OutNewDirStatusp
The status information of the new directory (the one to which the file is being moved) upon termination of the call.

OutOldFileFidp The file identifier for the file which was moved.

OutOldFileStatusp
The status information of the file which was moved.

OutNewFileFidp The file identifier for the file to which the file identified in *OutOldFileFidp* was, in fact, moved.

OutNewFileStatusp
The status information of the file identified by *OutNewFileFidp*.

Syncp The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.12 AFS_Symlink

```
error_status_t AFS_Symlink
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *DirFidp,
   [in]   afsTaggedName         *Namep,
   [in]   afsTaggedPath         *LinkContentsp,
   [in]   afsStoreStatus        *InStatusp,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFid                *OutFidp,
   [out]  afsFetchStatus        *OutFidStatusp,
   [out]  afsFetchStatus        *OutDirStatusp,
   [out]  afsToken              *OutTokenp,
   [out]  afsVolSync            *Syncp
);
```

AFS_Symlink parameters are:

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which the symbolic link is to be created.
Namep	The name of the link to create.
LinkContentsp	The target of the new symbolic link.
InStatusp	This specifies the clientModTime field and unixModeBits of the new link.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier for the newly created symbolic link.
OutFidStatusp	The status information of the newly created symbolic link upon termination of the call.
OutDirStatusp	The status information of the directory (the one in which the symbolic link was created) upon termination of the call.
OutTokenp	The token returned against the specified directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.13 AFS_HardLink

```
error_status_t AFS_HardLink
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *DirFidp,
   [in]   afsTaggedName         *Namep,
   [in]   afsFid                *ExistingFidp,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFetchStatus        *OutFidStatusp,
   [out]  afsFetchStatus        *OutDirStatusp,
   [out]  afsVolSync            *Syncp
);
```

AFS_HardLink parameters are:

h The RPC binding handle.

DirFidp The file identifier specifying the directory in which the file is currently located.

Namep The name to use for the new hard link.

ExistingFidp The file identifier specifying the file identifier of the file to which the hard link should be made.

minVVp The minimum-acceptable version number on the fileset containing this file or directory.

Flags Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

OutFidStatusp The status information of the newly created hard link upon termination of the call.

OutDirStatusp The status information of the directory (the one in which the hard link was created) upon termination of the call.

Syncp The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.14 AFS_MakeDir

```
error_status_t AFS_MakeDir
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *DirFidp,
   [in]   afsTaggedName     *Namep,
   [in]   afsStoreStatus    *InStatusp,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsFid            *OutFidp,
   [out]  afsFetchStatus    *OutFidStatusp,
   [out]  afsFetchStatus    *OutDirStatusp,
   [out]  afsToken          *OutTokenp,
   [out]  afsVolSync        *Syncp
);
```

AFS_MakeDir parameters are:

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which the new directory is to be created.
Namep	The name of the directory to create.
InStatusp	This specifies the new status information, including clientModTime field and unixModeBits, of the new directory point.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier for the newly created directory.
OutFidStatusp	The status information of the newly created directory upon termination of the call.
OutDirStatusp	The status information of the directory (the one in which the directory was created) upon termination of the call.
OutTokenp	A new token granted against the newly created directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.15 AFS_RemoveDir

```
error_status_t AFS_RemoveDir
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *DirFidp,
   [in]   afsFidTaggedName      *Namep,
   [in]   afsHyper              *returnTokenIDp,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFetchStatus        *OutDirStatusp,
   [out]  afsFid                *OutFidp,
   [out]  afsFetchStatus        *OutDelStatusp,
   [out]  afsVolSync            *Syncp
);
```

AFS_RemoveDir parameters are:

h The RPC binding handle.

DirFidp The file identifier specifying the directory in which the directory to be deleted is located.

Namep The complex name of the directory to delete.

returnTokenIDp A token ID being returned, if any.

minVVp The minimum-acceptable version number on the fileset containing this file or directory.

Flags Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

OutDirStatusp The status information of the directory (the one in which the directory was deleted) upon termination of the call.

OutFidp

OutDelStatusp

Syncp The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.16 AFS_Readdir

```
error_status_t AFS_Readdir
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *DirFidp,
   [in]   afsHyper          *Offsetp,
   [in]   unsigned32        Size,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsHyper          *NextOffsetp,
   [out]  afsFetchStatus    *OutDirStatusp,
   [out]  afsToken          *OutTokenp,
   [out]  afsVolSync        *Syncp,
   [out]  pipe_t            *dirStream
);
```

AFS_Readdir parameters are:

h	The RPC binding handle.
DirFidp	The file descriptor for the specified directory.
Offsetp	The offset into the directory for this entry.
Size	Number of bytes to read.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
NextOffsetp	The offset into the directory for the following entry.
OutDirStatusp	Status information for the directory pointed to by *DirFidp*.
OutTokenp	Token granted against *DirFidp*.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.
dirStream	The array of bytes making up the external representation of this part of the directory.

2.13.17 AFS_Lookup

```
error_status_t AFS_Lookup
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsFid                *DirFidp,
   [in]   afsTaggedName         *Namep,
   [in]   afsHyper              *minVVp,
   [in]   unsigned32            Flags,
   [out]  afsFid                *OutFidp,
   [out]  afsFetchStatus        *OutFidStatusp,
   [out]  afsFetchStatus        *OutDirStatusp,
   [out]  afsToken              *OutTokenp,
   [out]  afsVolSync            *Syncp
);
```

AFS_Lookup parameters are:

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory from which to obtain the directory information.
Namep	The character string name of the file for which information is being requested.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier of the requested file.
OutFidStatusp	Status information for the specified file.
OutDirStatusp	Status information of the specified directory.
OutTokenp	The token against the directory (allowing the directory entry to be cached).
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.18 AFS_GetToken

```
error_status_t AFS_GetToken
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *Fidp,
   [in]   afsToken          *MinTokenp,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsToken          *OutTokenp,
   [out]  afsRecordLock     *OutBlockerp,
   [out]  afsFetchStatus    *OutStatusp,
   [out]  afsVolSync        *Syncp
);
```

AFS_GetToken parameters are:

h	The RPC binding handle.
Fidp	File identifier of the file to obtain a token against.
MinTokenp	Specification of the minimum requested token.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutTokenp	Actual token granted.
OutBlockerp	Information about the possessor of the token that prevents the granting of the requested token (valid only for lock-family tokens).
OutStatusp	Status information on the file specified by *Fidp*.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.19 AFS_ReleaseTokens

```
error_status_t AFS_ReleaseTokens
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsReturns            *Tokens_Arrayp,
   [in]   unsigned32            Flags
);
```

AFS_ReleaseTokens parameters are:

h	The RPC binding handle.
Tokens_Arrayp	Tokens granted to this file server which are to be relinquished.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

2.13.20 AFS_GetTime

```
error_status_t AFS_GetTime
(/* provider_version(1) */
   [in]   handle_t          h,
   [out]  unsigned32        *Secondsp,
   [out]  unsigned32        *USecondsp,
   [out]  unsigned32        *SyncDistance,
   [out]  unsigned32        *SyncDispersion
);
```

AFS_GetTime parameters are:

h	The RPC binding handle.
Secondsp	Number of seconds since January 1, 1970 UTC.
USecondsp	Number of microseconds into the current segment.
SyncDistance	Estimated path length to source of reliable time.
SyncDispersion	Measure of SyncDistance's variance.

2.13.21 AFS_MakeMountPoint

```
error_status_t AFS_MakeMountPoint
(/* provider_version(1) */
   [in]   handle_t          h,
   [in]   afsFid            *DirFidp,
   [in]   afsTaggedName     *Namep,
   [in]   afsTaggedName     *cellNamep,
   [in]   afsFStype         Type,
   [in]   afsTaggedName     *volumeNamep,
   [in]   afsStoreStatus    *InStatusp,
   [in]   afsHyper          *minVVp,
   [in]   unsigned32        Flags,
   [out]  afsFid            *OutFidp,
   [out]  afsFetchStatus    *OutFidStatusp,
   [out]  afsFetchStatus    *OutDirStatusp,
   [out]  afsVolSync        *Syncp
);
```

AFS_MakeMountPoint parameters are:

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which the new mount point is to be created.
Namep	The name of the mount point.
cellNamep	The string name of the cell in which the mount point is being created.
Type	The type of the file system mount point.
volumeNamep	The name of the fileset to be mounted on the newly created mount point.
InStatusp	This specifies the clientModTime field and unixModeBits of the new mount point.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier for the newly created mount point.
OutFidStatusp	The status information of the newly created mount point upon termination of the call.
OutDirStatusp	The status information of the directory (the one in which the mount point was created) upon termination of the call.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.22 AFS_GetStatistics

```
error_status_t AFS_GetStatistics
(/* provider_version(1) */
   [in]   handle_t                h,
   [out]  afsStatistics           *Statisticsp
);
```

AFS_GetStatistics parameters are:

h	The RPC binding handle.
Statisticsp	File server statistics.

2.13.23 AFS_BulkFetchVV

```
error_status_t AFS_BulkFetchVV
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsHyper              *cellIdp,
   [in]   afsBulkVolIDs         *VolIDsp,
   [in]   unsigned32            NumVols,
   [in]   unsigned32            Flags,
   [in]   unsigned32            spare1,
   [in]   unsigned32            spare2,
   [out]  afsBulkVVs            *VolVVsp,
   [out]  unsigned32            *spare4
);
```

AFS_BulkFetchVV parameters are:

h	The RPC binding handle.
cellIdp	The cell identifier for the filesets whose Volume Version numbers are desired.
VolIDsp	The fileset identifiers (within cellIdp) whose Volume Version numbers are desired.
NumVols	(Redundant) number of identifiers in VolIDsp.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
spare1, spare2	Spare IN parameters.
VolVVsp	The Volume Version numbers for the filesets whose identifiers were passed in.
spare4	Spare OUT parameter.

2.13.24 AFS_BulkKeepAlive

```
error_status_t AFS_BulkKeepAlive
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   afsBulkFEX            *KAFEXp,
   [in]   unsigned32            numExecFids,
   [in]   unsigned32            Flags,
   [in]   unsigned32            spare1,
   [in]   unsigned32            spare2,
   [out]  unsigned32            *spare4
);
```

AFS_BulkKeepAlive parameters are:

h	The RPC binding handle.
KAFEXp	Collection of afsFid structures.
numExecFids	(Redundant) the number of afsFid structures in *KAFEXp*.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
spare1, spare2	Spare IN parameters.
spare4	Spare OUT parameter.

2.13.25 AFS_ProcessQuota

```
error_status_t AFS_ProcessQuota
(/* provider_version(1) */
   [in]    handle_t              h,
   [in]    afsFid                *Fidp,
   [in]    afsHyper              *minVVp,
   [in]    unsigned32            Flags,
   [in,out]afsQuota              *quotaListp,
   [out]   afsFetchStatus        *OutStatusp,
   [out]   afsVolSync            *Syncp
);
```

AFS_ProcessQuota parameters are:

h	The RPC binding handle.
Fidp	File ID of file or directory whose quota is being changed.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
quotaLisp	The quota information being provided or obtained.
OutStatusp	The updated file status after the operation.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

2.13.26 AFS_GetServerInterfaces

```
error_status_t AFS_GetServerInterfaces
(/* provider_version(1) */
   [in]    handle_t                h,
   [in,out]dfs_interfaceList       *serverInterfacesP
);
```

AFS_GetServerInterfaces parameters are:

h	The RPC binding handle.
serverInterfacesP	Will contain the interface information when the call returns. Currently there is only one interface defined.

2.13.27 AFS_SetParams

```
error_status_t AFS_SetParams
(/* provider_version(1) */
   [in]    handle_t              h,
   [in]    unsigned32            Flags,
   [in,out]afsConnParams         *paramsP
);
```

AFS_SetParams parameters are:

h	The RPC binding handle.
Flags	Either AFS_PARAM_TSR_COMPLETE or AFS_PARAM_RESET_CONN.
paramsP	A block of connection-oriented parameters.

2.14 The File Exporter Interface End

```
}
```

Chapter 3

Cache Manager IDL Declarations

This chapter specifies the RPC interfaces used by the AFS (4.0) Token Manager Interface. It contains the declaration of all structures and remote procedure calls (rpc) calls required for the AFS (4.0) Token Manager Interface.

The Token Manager Interface consists of a set of structures, mask values and flags contained in an IDL file with a provider version number of 1 to indicate that it is an original interface. In addition, each RPC defined in this structure also has a provider version number of 1, indicating that it also is an original RPC. See the topic, *DFS RPC Versioning Scheme*, Chapter 5 on page 69, for information on versioning which will explain in more detail about provider version numbers.

3.1 The Token Manager Interface

The Token Manager Interface is contained within the interface, **TKN4Int**, which starts out as shown below and contains all the items shown in this chapter exclusive of the definitions of the parameters in the RPC calls which are included from Chapter 7 on page 145 and the common data definitions in the configuration file **common_data.idl** which are contained in Chapter 4 on page 57 for clarity. There are nine remote procedure calls defined that the client and cache manager can use in this interface which is defined as provider version 1. Chapter 5 on page 69 provides information for adding new interfaces, or new remote procedure calls to this interface.

```
[
uuid(4d37f2dd-ed96-0000-02c0-37cf1e000000),
version(4.0)
/* provider_version(1) */
]

interface TKN4Int
{

import "dcedfs/common_data.idl";

  .
  .
  .
/* }  at end of this chapter concludes this interface. */
```

3.2 Definitions for Flags Field

These are the definitions for the flags field used in this interface.

```
const unsigned32 TKN_FLAG_BACK_UP = 0x001;
    /* this is an unannounced TKN_InitTokenState call from a
       recovering server */

const unsigned32 TKN_FLAG_CRASHED = 0x002;
    /* the file exporter is currently in a post-crash
       token-recovery phase */

const unsigned32 TKN_FLAG_DISALLOW_SAME = 0x004;
    /* this recovery cannot accept AFS_FLAG_TOKENID AFS_GetToken calls */

const unsigned32 TKN_FLAG_NEW_IF = 0x008;
    /* this call is made to set the timer values; the tokens are OK */
```

3.3 Definitions of Token Manager Exported Operations

Note: The Opcode of each operation is implicitly assigned based on the order of where the operation is placed. The recommendation is to always place the new operation at the end of this list.

3.3.1 TKN_Probe

```
error_status_t TKN_Probe    /* May be defined as "idempotent"  */
(/* provider_version(1) */
   [in]    handle_t    h
);
```

TKN_Probe parameters are:

h The RPC binding handle.

3.3.2 TKN_InitTokenState

```
error_status_t TKN_InitTokenState
(/* provider_version(1) */
   [in]    handle_t                h,
   [in]    unsigned32              Flags,
   [in]    unsigned32              hostLifeGuarantee,
   [in]    unsigned32              hostRPCGuarantee,
   [in]    unsigned32              deadServerTimeout,
   [in]    unsigned32              serverRestartEpoch,
   [in]    unsigned32              spare1,
   [in]    unsigned32              spare2,
   [in]    unsigned32              spare3,
   [out]   unsigned32              *spare4,
   [out]   unsigned32              *spare5,
   [out]   unsigned32              *spare6
);
```

TKN_InitTokenState parameters are:

h The RPC binding handle.

Flags Flags (TKN_FLAG*) indicating the server's likely token state.

hostLifeGuarantee
Duration of the host lifetime on the server.

hostRPCGuarantee
Duration of the promise of RPC attempts from the server.

deadServerTimeout
Suggested maximum interval between probes from Cache Managers to unresponsive servers.

serverRestartEpoch
Numeric tag for this run of the File Server, to allow Cache Managers to distinguish network partitions from server crashes.

spare1 .. spare6 Spares.

3.3.3 TKN_TokenRevoke

```
error_status_t TKN_TokenRevoke
(/* provider_version(1) */
   [in]    handle_t              h,
   [in,out]afsRevokes            *revokeDescp
);
```

TKN_TokenRevoke parameters are:

h The RPC binding handle.

revokeDescp The tokens which are being revoked.

3.3.4 TKN_GetCellName

```
error_status_t TKN_GetCellName
(/* provider_version(1) */
   [in]    handle_t              h,
   [in]    afsHyper              *Cellp,
   [out]   NameString_t          CellNamep
);
```

TKN_GetCellName parameters are:

h The RPC binding handle.

Cellp The ID for a cell.

CellNamep The string name corresponding to **cellP**.

3.3.5 TKN_GetLock

```
error_status_t TKN_GetLock
(/* provider_version(1) */
   [in]    handle_t              h,
   [in]    unsigned32            index,
   [out]   afsDBLock             *lock
);
```

TKN_GetLock parameters are:

h The RPC binding handle.

index Entry to return.

lock Lock information returned, including name, wait states, exclusive locks, readers reading and number waiting on it.

3.3.6 TKN_GetCE

```
error_status_t TKN_GetCE
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   unsigned32            index,
   [out]  afsDBCacheEntry       *ce
);
```

TKN_GetCE parameters are:

h The RPC binding handle.

index Entry to return.

[out] afsDBCacheEntry *ce

ce The cache entry with token information, number of opens, readers, writers, shared, exclusives; locks, , expiration, mask, reference count, network part of the fid, and so on.

3.3.7 TKN_GetServerInterfaces

```
error_status_t TKN_GetServerInterfaces
(/* provider_version(1) */
   [in]   handle_t              h,
   [in,out]dfs_interfaceList    *serverInterfacesP
);
```

TKN_GetServerInterfaces parameters are:

h The RPC binding handle.

serverInterfacesP List of supported interfaces.

3.3.8 TKN_SetParams

```
error_status_t TKN_SetParams
(/* provider_version(1) */
   [in]   handle_t              h,
   [in]   unsigned32            Flags,
   [in,out]afsConnParams        *paramsP
);
```

TKN_SetParams parameters are:

h The RPC binding handle.

Flags For future expansion.

paramsP A block of connection-oriented parameters.

3.3.9 TKN_AsyncGrant

```
error_status_t TKN_AsyncGrant
(/* provider_version(1) */
   [in]  handle_t          h,
   [in]  afsFid            *grantedFileIDp,
   [in]  afsToken          *grantedTokenP,
   [in]  long              grantedRequestID
);
```

TKN_AsyncGrant parameters are:

h	The RPC binding handle.
grantedFileIDp	File identifier of the file or directory to which the granted token applies.
grantedTokenP	The token that was granted.
grantedRequestID	Identifier of the original async grant request.

3.4 The Token Manager Interface End

```
}
```

Chapter 4

Common IDL Data

This chapter specifies the common data structures required for the AFS (4.0) Client and Server. The definitions here are necessary for both the File Exporter Interface and the Token Manager Interface and are imported into those interfaces.

4.1 Interface Common Data

```
interface common_data

{
  .  /* All the information in this chapter is part of the
  .   * common_data interface, from here to the ending brace
  .   * following the heading, "Interface Common Data end"   */
/* } at the end of this chapter concludes the interface common data. */
```

4.2 General AFS Constants

```
const unsigned32 AFS_UNDEFAFSID = -1;
                    /* Generic undefined AFS id */
const unsigned32 AFS_NAMEMAX = 256;
const unsigned32 AFS_PATHMAX = 1024;
const unsigned32 AFS_MAXHOSTS = 8;
const unsigned32 AFS_TOKENDEAD = 1235;
const unsigned32 AFS_ACLMAX = 8188;
                    /* 8k - sizeof(long), so afsACL in 8k; was 1024 */
const unsigned32 AFS_BULKMAX = 32;
const unsigned32 AFS_DISKNAMESIZE = 32;
const unsigned32 AFS_NAMEMAXLEN = 112;
                    /* Max len for fileset names */
```

4.3 Constants for Cell and Hosts

```
const unsigned32 MAXCELLCHARS = 128;
const unsigned32 MAXHOSTCHARS = 128;
const unsigned32 MAXHOSTSPERCELL = 64;
```

4.4 AFS Object Types Used by AFS_Mount

```
const unsigned32 Invalid = 0;
const unsigned32 File = 1;
const unsigned32 Directory = 2;
const unsigned32 SymbolicLink = 3;
const unsigned32 MountPoint = 4;
const unsigned32 FIFO = 5;
const unsigned32 Socket = 6;
const unsigned32 BlockDev = 7;
const unsigned32 CharDev = 8;
const unsigned32 FETCHSTATUS_VERSION = 2;
```

4.5 Quota Types for Quota Setting Commands

```
const unsigned32 AFS_FILESYS_BSD = 1;
const unsigned32 AFS_FILESYS_EPISODE = 2;
```

4.6 Quota Opcodes for Quota Setting Commands

```
const unsigned32 AFS_QUOTA_GET = 1;
const unsigned32 AFS_QUOTA_SET = 2;
```

4.7 Physical File System Types (for afsFStype)

```
const unsigned32 AFS_FS = 0;
const unsigned32 DEC_FS = 1;
const unsigned32 UFS_FS = 2;
const unsigned32 AIX_FS = 4;
```

4.8 Volume Types Used for afsVolumeType

```
const unsigned32 ReadOnly = 0;
const unsigned32 ReadWrite = 1;
```

4.9 Values for the afsRevokeDesc Flags

```
const unsigned32 AFS_REVOKE_COL_A_VALID = 0x1;
const unsigned32 AFS_REVOKE_COL_B_VALID = 0x2;
const unsigned32 AFS_REVOKE_FORCED_REVOCATION = 0x4;
const unsigned32 AFS_REVOKE_DUE_TO_GC = 0x8;
```

4.10 Values Used in afsRevokeDesc's outFlags

```
const unsigned32 AFS_REVOKE_LOCKDATA_VALID = 0x1;
```

4.11 Data Types

```
typedef uuid_t   afsUUID;
typedef long     afsFStype;
typedef long     afsVolumeType;

typedef pipe     byte    pipe_t;

typedef unsigned32   codesetTag;
const unsigned32 AFS_TAG_ORIGASCII = 0;
```

4.11.1 General Definitions for AFS Data Structures

```
typedef [string] byte    NameString_t[AFS_NAMEMAX];
typedef [string] byte    PathString_t[AFS_PATHMAX];
typedef byte  afsDiskName[AFS_DISKNAMESIZE];
```

4.11.2 General Tagged-name for a Pathname Component

Use of the tn_tag value AFS_TAG_ORIGASCII defined above in Section 4.11 means that the tn_length field doesn't need to be filled in.

Note: tn_chars is assumed to be null-terminated.

4.11.3 Define the afsTaggedName Structure

```
typedef struct afsTaggedName {
    codesetTag    tn_tag;
    unsigned16    tn_length;
    byte          tn_chars[AFS_NAMEMAX+1];
} afsTaggedName;
```

4.11.4 Define the afsTaggedPath Structure

```
typedef struct afsTaggedPath {
    codesetTag    tp_tag;
    unsigned16    tp_length;
    byte          tp_chars[AFS_PATHMAX+1];
} afsTaggedPath;
```

4.11.5 Define the afsNetAddr Structure

```
typedef struct afsNetAddr {
    unsigned16 type;
    unsigned8 data[14];
} afsNetAddr;
```

4.11.6 Define the afsTimeval Structure

```
typedef struct afsTimeval {
    unsigned32 sec;
    unsigned32 usec;
} afsTimeval;
```

4.11.7 Define the afsHyper Structure

Note: This is a 64-bit structure used by objects such as cells, volumes, and so on.

```
typedef struct afsHyper {
    unsigned32 high;
    unsigned32 low;
} afsHyper;

/*
 * Generic operations on afsHypers are defined in dcedfs/stds.h.
 */
```

4.11.8 Define the afsFid Structure

```
typedef struct afsFid {
    afsHyper Cell;
    afsHyper Volume;
    unsigned32 Vnode;
    unsigned32 Unique;
} afsFid;
```

4.11.9 Define the afsFidTaggedName Structure

```
typedef struct afsFidTaggedName {
    afsFid fid;
    afsTaggedName name;
} afsFidTaggedName;
```

4.11.10 Define the afsToken Structure

```
typedef struct afsToken {
    afsHyper tokenID;
    unsigned32 expirationTime;
    afsHyper type;
    unsigned32 beginRange;
    unsigned32 endRange;
    unsigned32 beginRangeExt;
    unsigned32 endRangeExt;
} afsToken;
```

4.11.11 Define the afsRecordLock Structure

Note: This is a Sys V Record Lock to return blocker's info.

```
typedef struct afsRecordLock {
    signed16     l_type;
    signed16     l_whence;
    unsigned32   l_start_pos;
    unsigned32   l_end_pos;
    unsigned32   l_pid;
    unsigned32   l_sysid;
    unsigned32   l_fstype;
    unsigned32   l_start_pos_ext;
    unsigned32   l_end_pos_ext;
} afsRecordLock;
```

4.11.12 Define the afsRevokeDesc Structure

Note: This structure contains information about the token the Cache Manager (cm) already holds.

```
typedef struct afsRevokeDesc {

    afsFid fid;          /* useful hint */
    afsHyper tokenID;
    afsHyper type;       /* mask */
    unsigned32 flags;    /* input flags to RPC*/
    unsigned32 outFlags; /* output flags from RPC */
    afsHyper errorIDs;   /* (mask[dude] == 1) <==> dude not revoked */
    /*
     * Info about the tokens to be offered for swapping
     * The first one
     */
    afsToken columnA;    /* the offer */
    afsHyper colAChoice; /* the accepted token types */
    /*
     * the other token to be offered
     */
    afsToken columnB;    /* the offer */
    afsHyper colBChoice; /* the accepted token types */
    afsRecordLock recordLock;  /* the blocker's info */
 } afsRevokeDesc;
```

4.11.13 Define the afsReturnDesc Structure

Note: This structure contains information for the return of tokens.

```
typedef struct afsReturnDesc {

    afsFid fid;          /* useful hint */
    afsHyper tokenID;
    afsHyper type;       /* mask */
    unsigned32 flags;    /* just in case */
} afsReturnDesc;
```

4.11.14 Define the afsConnParams Structure

Note: This is the method of adjusting connection parameters:

1. Low-order bit of Mask (0x1) declares that Values[0] contains information.
2. Next bit in Mask (0x2) declares that Values[1] contains information.

```
typedef struct afsConnParams {
    unsigned32 Mask;
    unsigned32 Values[20];
} afsConnParams;
const unsigned32 AFS_CONN_PARAM_HOSTLIFE = 0;
const unsigned32 AFS_CONN_PARAM_HOSTRPC = 1;
const unsigned32 AFS_CONN_PARAM_DEADSERVER = 2;
const unsigned32 AFS_CONN_PARAM_EPOCH = 3;
```

4.11.15 Define the afsDBLockDesc Structure

Note: This is a temporary structure for debugging purposes.

```
typedef struct afsDBLockDesc {
    unsigned8 waitStates;
    unsigned8 exclLocked;
    unsigned8 readersReading;
    unsigned8 numWaiting;
} afsDBLockDesc;
```

4.11.16 Define the afsDBCacheEntry Structure

Note: This is a temporary structure for debugging purposes.

```
typedef struct afsDBCacheEntry {
    unsigned32 addr;
    afsFid fid;              /* network part of the fid */
    afsHyper length;
    afsHyper dataVersion;
    afsDBLockDesc lock;
    unsigned32 tokenBaseID;
    unsigned32 tokenMask;
    unsigned32 tokenExpiration;
    signed16 refCount;
    signed16 opens;
    signed16 writers;
    signed16 readers;
```

```
    signed16 shareds;
    signed16 exclusives;
    unsigned8 mvstat;
    unsigned8 states;
} afsDBCacheEntry;
```

4.11.17 Define the afsDBLock Structure

Note: This is a temporary structure for debugging purposes.

```
typedef struct afsDBLock {
    byte name[16];
    afsDBLockDesc lock;
} afsDBLock;
```

4.12 Various Bulk Typedefs From Primitive Structures

These are from the primitive structures above.

4.12.1 Define the afsRevokes Structure

```
typedef struct afsRevokes {    /* needed also by tkn4int.idl  */
    long afsRevokes_len;
    [length_is(afsRevokes_len)] afsRevokeDesc afsRevokes_val[AFS_BULKMAX];
} afsRevokes;
```

4.12.2 Define the afsReturns Structure

```
typedef struct afsReturns {    /* needed also by tkn4int.idl  */
    long afsReturns_len;
    [length_is(afsReturns_len)] afsReturnDesc afsReturns_val[AFS_BULKMAX];
} afsReturns;
```

4.12.3 Define the afsFids Structure

```
typedef struct afsFids {
    unsigned32 afsFids_len;
    [length_is(afsFids_len)] afsFid afsFids_val[AFS_BULKMAX];
} afsFids;
```

4.12.4 Define the afsTokens Structure

```
typedef struct afsTokens {
    unsigned32 afsTokens_len;
    [length_is(afsTokens_len)] afsToken afsTokens[AFS_BULKMAX];
} afsTokens;
```

4.12.5 Define the afsStrings Structure

Note: afsStrings, used in AFS_BulkLookup, used to be identical to NameString_t itself.

```
typedef struct afsStrings {
    unsigned32 afsStrings_len;
    [length_is(afsStrings_len)] afsTaggedName afsStrings_val[AFS_BULKMAX];
} afsStrings;
```

4.13 Data Types for DFS RPC Versioning Scheme

4.13.1 Constants for RPC Versioning Scheme

```
const unsigned32 MAXSPARETEXT = 50;
const unsigned32 MAXINTERFACESPERVERSION = 10;
```

4.13.2 dfs_interfaceDescription Structure

```
typedef struct dfs_interfaceDescription {

    uuid_t     interface_uuid;  /* i/f UUID of the supported i/f */
    unsigned16 vers_major;      /* i/f major version number */
    unsigned16 vers_minor;      /* i/f minor version number */
    unsigned32 vers_provider;   /* provider version number */

    unsigned32 spare0;          /* some long spares */
    unsigned32 spare1;
    unsigned32 spare2;
    unsigned32 spare3;
    unsigned32 spare4;
    unsigned32 spare5;
    unsigned32 spare6;
    unsigned32 spare7;
    unsigned32 spare8;
    unsigned32 spare9;

    byte spareText[MAXSPARETEXT]; /* spare text */

} dfs_interfaceDescription;
```

4.13.3 dfs_interfaceList Structure

```
typedef struct dfs_interfaceList  {

    unsigned32 dfs_interfaceList_len;
    [length_is(dfs_interfaceList_len)] dfs_interfaceDescription
         dfs_interfaceList_val[MAXINTERFACESPERVERSION];

} dfs_interfaceList;
```

4.14 Interface Common Data End

```
}
```

X/Open Preliminary Specification

Part 3:

DFS Versioning Scheme

Chapter 5

DFS RPC Versioning Scheme

This is the formal specification document for the DFS client/server versioning scheme. This mechanism should be supported by all DFS clients and servers. This specification is organized into six sections.

Data Structures	Defined in Chapter 4 on page 57 with specific definitions for versioning found in Section 4.13 on page 64, and demonstrated in Section 5.1.
Versioning API	Defined in Section 5.2 on page 70.
Example IDL file	Defined in Section 5.3 on page 73.
Example Client Application	Defined in Section 5.4 on page 76.
Example Server Application	Defined in Section 5.5 on page 83.
Example Manager Application	Defined in Section 5.6 on page 87.
Extending DFS interfaces	Defined in Section 5.7 on page 95.

5.1 Data Structures

This example structure defines the interface Description for each interface that is supported by the server. The server maintains an array of such structures for each interface that it exports. Also, upon startup the server fills these arrays with the interfaces that it registers with the runtime.

The actual structure is defined in Section 4.13.2 on page 64 and is repeated here so that this chapter is self-explanatory.

Example 5-1 Interface Description for an Interface

```
const long MAXSPARETEXT = 50;
const long MAXINTERFACESPERVERSION = 10;

typedef struct dfs_interfaceDescription {

    uuid_t      interface_uuid;  /* i/f UUID of the supported i/f */
    unsigned16  vers_major;      /* i/f major version number */
    unsigned16  vers_minor;      /* i/f minor version number */
    unsigned32  vers_provider;   /* provider version number */

    unsigned32 spare0;           /* some long spares */
    unsigned32 spare1;
    unsigned32 spare2;
    unsigned32 spare3;
    unsigned32 spare4;
    unsigned32 spare5;
    unsigned32 spare6;
    unsigned32 spare7;
    unsigned32 spare8;
```

```
    unsigned32 spare9;

    byte spareText[MAXSPARETEXT];/* spare text */
} dfs_interfaceDescription;
```

The following example structure shows a list of interface descriptions for a particular interface. A Client requests a list of different interface versions for a particular RPC interface using the RPC_GetServerInterfaces( ) remote procedure call.

The actual structure is defined in Section 4.13.3 on page 64 and is repeated here so that this chapter is self-explanatory.

Example 5-2 List of Interface Descriptions for an Interface

```
typedef struct dfs_interfaceList  {

    unsigned32 dfs_interfaceList_len;
    [length_is(dfs_interfaceList_len)] dfs_interfaceDescription \
        dfs_interfaceList_val[MAXINTERFACESPERVERSION];
} dfs_interfaceList;
```

This example list (defined in Section 4.13.3 on page 64) shows the structure that servers fill with the interface(s) they support. They return a set of such (interface description) structures (defined in Section 4.13.2 on page 64) back to the client which is requesting it. This list is used by the versioning code described in Section 5.2 to find the best match between the facilities supported by a server and the facilities desired by a client.

5.2 DFS Versioning API

The versioning API for DFS makes use of some definitions whose syntax is unique to DFS. They are *IMPORT* and *_TAKES*. The syntax for *IMPORT* is:

```
#define IMPORT  extern
```

The syntax for *_TAKES* is shown in its usage via the following example:

Example 5-3 Usage of IMPORT definition with _TAKES notation

```
IMPORT char * getenv _TAKES((
       char * varName
       ));
```

As can be seen, *IMPORT* is defined as *extern*. The *getenv* function is a utility function defined in the header **<stdlib.h>**, that is used to declare functions for number conversion, storage allocation and similar items. *_TAKES* is used simply as notation showing what the shown *IMPORT* definition *takes* as a construct.

Example 5-4 Construction of the *getenv* utility

getenv is an implementation-dependent function. It has the following format:

```
char * getenv(const char *name)
```

getenv returns the environment string associated with name, or NULL if no string exists.

5.2.1 Register an Interface with Versioning Mechanism

This function is used to register an interface with the versioning mechanism for DFS. It is called by server applications only.

```
IMPORT void dfs_installInterfaceDescription _TAKES((
                rpc_if_handle_t if_spec,
                rpc_if_handle_t orig_spec,
                unsigned32 vers_provider,
                unsigned_char_t *text,
                error_status_t *code
                ));
```

dfs_installInterfaceDescription parameters are:

if_spec	Interface specification that is to be registered.
orig_spec	Interface specification that is extended (that is, Transarc's interface).
vers_provider	Provider version number of this interface.
text	Text describing the new interface (maximum of 50 characters).
code	Status that is returned from this call.

5.2.2 Printing a Returned List of Interfaces

This function is useful only for debugging purposes. It prints in human-readable form, the list of interfaces returned by the server. It is called by client/server applications.

```
IMPORT void dfs_printInterfaceDescription _TAKES ((
                dfs_interfaceList *interfaces,
                error_status_t *code
                ));
```

dfs_printInterfaceDescription parameters are:

interfaces	A list of interfaces returned by the server. See Section 4.13.2 on page 64 and Section 4.13.3 on page 64 for more details.
code	Status that is returned from this call.

5.2.3 Function Called at Startup to Identify the Server Interface

This function is called by the client at startup to identify the server interface that the client is to use for RPCs. It is called by Client applications only.

```
IMPORT void dfs_GetServerInterfaces _TAKES((
                rpc_if_handle_t if_spec,
                dfs_interfaceList *interfaces,
                error_status_t *code
                ));
```

dfs_GetServerInterfaces parameters are:

if_spec	Interface specification that is to be examined.
interfaces	A list of server supported interfaces returned by the server. See Section 4.13.2 on page 64 and Section 4.13.3 on page 64 for more details.

code Status that is returned from this call.

5.2.4 Function to Compare Interfaces

This function is called by the client to compare the interface spec that it desires to get access to and the interfaces in the list of interface descriptions returned by the RPC_GetServerInterfaces call. It returns:

MATCH_GOOD UUID and major version numbers of the client and server interfaces are identical. (Refer to Example 5-1 on page 69 for more information on version numbers, minor and major, and UUIDs, if desired.)

This means that either:

1. Client and server interfaces are identical - the minor version numbers and the provider version numbers are equal.
2. Server minor version is greater than client's.
3. Server provider version is greater than client's.

MATCH_MEDIUM
UUID and major version numbers of client and server are identical, and the minor version of client is less than or equal to server's. In addition the provider version number of client is greater than server's.

MATCH_BAD The UUID or the major version number of the client is different from the server's. Client-Server binding in this case is impossible and hence this case is least desirable.

```
IMPORT long dfs_sameInterface _TAKES ((
                rpc_if_handle_t if1,
                unsigned int if1_provider,
                dfs_interfaceList *serverInterfaces
                ));
```

dfs_sameInterface parameters are:

if1 Interface specification that is to be used for a comparison.

if1_provider Provider version number.

serverInterfaces A list of server supported interfaces returned by the server. See Section 4.13.2 on page 64 and Section 4.13.3 on page 64 for more details.

5.3 Example IDL File

Provider numbers for the interface and for each RPC are added as a comment. The provider version number for the interface (in the IDL header) signifies a new version of the interface. The provider version number associated with each RPC signifies the provider version number of the interface at which time this RPC was added to the interface. In this example, the IDL file is the orignal interface and so, the provider number for the interface and for the RPC are both 1.

This example is taken from the UPDATE Server (AFS 4.0) Interface Definition which is owned by Transarc Corporation. The portion necessary in order to show how versioning works is the exported procedure for fetching the interfaces supported by the upserver, namely, *UPDATE_GetServerInterfaces*( ). However, for completeness, other procedures are also shown since they are included in the example manager application in Section 5.6 on page 87.

5.3.1 The UPDATE Server Interface

```
[
uuid(4d37f2dd-ed43-0000-02c0-37cf1e001000),
version(4.0)
/* provider_version(1) */
]

interface UPDATE
{
/*
 * Include the AFS basic Header files
 */

    /**
     **   import "cellconfig.idl" ;
     **/
    import "dcedfs/common_data.idl";
        ...
        ...

/* } at end of this section ends this interface. */
```

5.3.2 Constants for the UPDATE Interface

```
const    long    TIMEOUT = 300;
const    long    MAX_PROTSEQ_LEN = 32;
const    long    BINDING_LEN = 256;
const    long    MAX_BINDING_LEN = 1024;
const    long    MAX_NAME_SIZE = 256;    /* include null byte at end */
```

5.3.3 Define the updateFileStatS Structure

```
typedef   struct updateFileStatS
{
    afsHyper fileLength;
    unsigned32 mode;
    unsigned32 uid;
    unsigned32 gid;
    unsigned32 mtime;     /* modify time */
    unsigned32 atime;     /* access time */
    unsigned32 spare1;
    unsigned32 spare2;
    unsigned32 spare3;
    unsigned32 spare4;
} updateFileStatT;
```

5.3.4 UPDATE_GetServerInterfaces

This exported procedure is called by the upclient to fetch the interfaces supported by the upserver. A bulk parameter of upto MAXINTERFACES number of interfaces is returned.

```
error_status_t UPDATE_GetServerInterfaces
(/* provider_version(1) */
   [in]    handle_t                h,
   [in,out]dfs_interfaceList       *serverInterfaces
);
```

UPDATE_GetServerInterfaces parameters are:

h The RPC binding handle.

serverInterfaces A list of server supported interfaces returned by the server. See Section 4.13.2 on page 64 and Section 4.13.3 on page 64 for more details.

5.3.5 UPDATE_FetchInfo

This exported procedure is called by a client to fetch the information (that is, the status) of a specified directory in the remote host.

An AFS client calls this remote procedure periodically to bring object files at its site update-to-date.

```
error_status_t UPDATE_FetchInfo
(/* provider_version(1) */
   [in]    handle_t                h,  /* Necessary for explicit_handle */
   [in]    NameString_t            DirName,
   [out]   pipe_t                  *Stream
);
```

UPDATE_FetchInfo parameters are:

h The RPC binding handle.

DirName The string name corresponding to the specified directory.

Stream The character pipe parameter returning the data from the directory. (See **X/Open DCE: Remote Procedure Call** specification for implementation details.)

5.3.6 UPDATE_FetchFile

This exported procedure is called by a client to fetch the whole data from a specified file in the remote host.

An AFS client calls this remote procedure periodically to bring object files at the client site update-to-date.

```
error_status_t UPDATE_FetchFile
(/* provider_version(1) */
   [in]   handle_t              h,  /* Necessary for explicit_handle */
   [in]   NameString_t          FileName,
   [out]  pipe_t                *Stream
);
```

UPDATE_FetchFile parameters are:

h	The RPC binding handle.
FileName	The string name corresponding to the specified file.
Stream	The character pipe parameter returning the data from the file. (See **X/Open DCE: Remote Procedure Call** specification for implementation details.)

5.3.7 UPDATE_FetchObjectInfo

This exported procedure is called by a client to fetch the information (that is, the status) of a specified file in the remote host.

An AFS client calls this remote procedure periodically to bring object files at its site update-to-date.

```
error_status_t UPDATE_FetchObjectInfo
(/* provider_version(1) */
   [in]   handle_t              h,  /* Necessary for explicit_handle */
   [in]   NameString_t          objectName,
   [out]  updateFileStatT       *fileStatP
);
```

UPDATE_FetchObjectInfo parameters are:

h	The RPC binding handle.
objectName	The string name corresponding to the specified file object.
fileStatP	Returns the data from the specified object. See Section 5.3.3 on page 74 for details.

5.3.8 The UPDATE Server Interface End

```
}
```

5.4 Example Client Application

The client application, upon startup retrieves server supported interfaces and sets a global to identify the interface (stub) that the client would use to make RPCs. The dfs_selectInterface( ) function sets the global and each RPC wrapper first checks the global and calls the appropriate stub.

5.4.1 Typical Client Headers

```
#include <dcedfs/param.h>
#include <dcedfs/stds.h>
#include <dcedfs/osi.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <dirent.h>
#include <pthread.h>
#include <dce/rpc.h>
#include <dcedfs/compat.h>
#include <dcedfs/cmd.h>
#include <update.h>
#include <upcommon.h>
```

5.4.2 Headers for Serviceability

The following headers get included for serviceability. The last three are specific to this subcomponent. They would need to be defined for each subcomponent that has implemented serviceability, and the names would be chosen to be specific to the function. The first three are included as is regardless of what subcomponent they are in, as they are needed for general serviceability and are not specific to the subcomponent.

The explanation of serviceability is beyond the scope of this document and this example.

```
/* For serviceability */
#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <dce/dcesvcmsg.h>
#include <dfsudtmac.h>
#include <dfsudtsvc.h>
#include <dfsudtmsg.h>
```

5.4.3 Constants for the Client Application

```
/* flag to check if UPDATE interface accessible */
unsigned int UPDATE_INTERFACE = 0;
unsigned int UPDATE_CHECK_PROVIDER_OR_RANGE = 0;

/* List of interfaces supported */
#define DFS_UPDATE        1
#define UPDATE_PROVIDER   1

/* should be NIDL_PIPE_BUFF_SIZE defined in stubbase.h */
#define PREFER_BUFFSIZE         2048
#define UPDATE_MAXBINDINGS      32
#define MAX_RPC_FAILURES        10
```

```
#define UPDATE_RETRY_TIME       60
#define UPDATE_MAX_CANCEL_ITER  20
```

5.4.4 RPC prewrap and postwrap functions

```
/* macros to refresh ubik connections upon ticket expiration */
unsigned int DFS_RPC_WRAP_ERROR;
#define DFS_RPC_PREWRAP { int rpc_attempts, retry_rpc, dfsauthRtn; \
    rpc_attempts = dfsauthRtn = 0; \
    retry_rpc = 1; \
    while (retry_rpc) { \
        retry_rpc = 0; \
        rpc_attempts++;

#define DFS_RPC_POSTWRAP(error_status) \
    if (error_status == rpc_s_auth_tkt_expired) { \
    if (rpc_attempts == 1) { \
    dfsauthRtn =dfsauth_client_RefreshLocalAuthContext();\
    if (!dfsauthRtn) retry_rpc = 1; \
    else {\
    retry_rpc = 0; \
    DFS_RPC_WRAP_ERROR = 1; \
    }}}}}
```

5.4.5 Typical Serviceability initialization

```
char       *whoami = "upclient";

dce_svc_handle_t   udt_svc_handle;

void initialize_svc()
{
  error_status_t st;

  udt_svc_handle = dce_svc_register(udt_svc_table, (idl_char *)"udt", &st);
  if (st != svc_s_ok)
     fprintf(stderr, "Cannot register svc 0x%lx0, st);
  dce_svc_routing((unsigned char *) "NOTICE:STDOUT:--", &st);
  if (st != svc_s_ok)
     fprintf(stderr, "Cannot route NOTICE messages  0x%lx0, st);
  dce_svc_routing((unsigned char *) "WARNING:STDOUT:--", &st);
  if (st != svc_s_ok)
     fprintf(stderr, "Cannot route WARNING messages  0x%lx0, st);
  dce_svc_routing((unsigned char *) "ERROR:STDERR:--", &st);
  if (st != svc_s_ok)
     fprintf(stderr, "Cannot route ERROR messages  0x%lx0, st);

  dfs_define_udt_message_tables();
}
```

5.4.6 ANSI C Declaration

```
/* Forward declaration for ANSI C */
long isCompatibleObject();
```

5.4.7 Client's import list

```
/* filestr - client's list of items to import */

struct filestr {
    struct filestr *next;
    int type;
    char *name;
};
```

5.4.8 Client Globals

```
/* client globals */
upLogP  logP = 0;   /* upclient's handle on it's log file */

void pipeAlloc();
void ReceiveFile();
int  AddToList();
int  ZapList();
int  IsCompatible();
int  NotOnHost();
```

5.4.9 Using dfs_selectInterface to Select One

This finction checks to see if client has access to the desired interface. It can associate priorities with interfaces in the case of servers exporting multiple interfaces.

```
static error_status_t
dfs_selectInterface (h)
    rpc_binding_handle_t h;

{
    rpc_if_id_t if_id;
    static dfs_interfaceList serverInterfaces;
    unsigned int i, j, result;
    error_status_t code;

    serverInterfaces.dfs_interfaceList_len = 0;
    bzero((char *)serverInterfaces.dfs_interfaceList_val,
        MAXINTERFACESPERVERSION * sizeof(dfs_interfaceDescription));

    DFS_RPC_PREWRAP
    code = UPDATE_GetServerInterfaces (h, &serverInterfaces);
    DFS_RPC_POSTWRAP(code)

    if ( IS_COMM_ERR(code) )
        return code;

#ifdef DEBUG
```

```
    dfs_printInterfaceDescription (&serverInterfaces, &code);
#endif

    /* Check to see if you have access to desired interface. The
       following are the different classes from most desired to
       least.

       Good:   UUID and major version numbers of the client and server
               interfaces are identical.

               1. client and server interfaces are identical -- the minor
                  version numbers and the provider version numbers are equal.
               2. Server minor version is greater than client's.
               3. Server provider version is greater than client's.

       Medium: UUID and major version numbers of client and server are
               identical, and the minor version of client is less than
               or equal to server's. In addition the provider version
               number of client is greater than server's.

       Bad:    The UUID or the major version number of the client is
               different from the server's. Client-Server binding in this
               case is impossible and hence this case is least desirable. */

    if (( result =  dfs_sameInterface (UPDATE_v4_0_c_ifspec,
        UPDATE_PROVIDER,
        &serverInterfaces)) == MATCH_GOOD ||
            result == MATCH_MEDIUM) {

        /* check for new interfaces derived from the original interface
           should be in here, if multiple versions of a DFS interface is
           supported. Here we only have one version of the interface.
         */
        UPDATE_INTERFACE = DFS_UPDATE; /* original interface */

    if (result == MATCH_MEDIUM)
        UPDATE_CHECK_PROVIDER_OR_RANGE = 1;
    }
    if (!UPDATE_INTERFACE)
        return UP_BAD_INTERFACE;
    else return 0;
}
```

5.4.10 Example RPC Wrappers

The following are example RPC wrappers that a client can use.

The update_Fetchfile Wrapper

```
error_status_t
update_FetchFile (h, filename, pipeP)
    rpc_binding_handle_t h;
    NameString_t filename;
    pipe_t *pipeP;
{
    error_status_t code;
    static unsigned long FUNCTION_PROVIDER = 1;
                     /* provider version # this RPC is supported */
    static unsigned long FUNCTION_NOOP = 0;
                    /* Is this function supported by server or not? */

    if (FUNCTION_NOOP) /* this RPC not supported */
        return (rpc_s_op_rng_error);

    /* Check to see if the desired interface provider is supported. If not
       we may try alternate interfaces. Also check if this RPC is supported
       by the server interface. This is done by comparing provider versions
    */

    if (UPDATE_INTERFACE == DFS_UPDATE &&
        UPDATE_PROVIDER >= FUNCTION_PROVIDER) {

#ifdef UPDATE_CANCEL_TEST
    Log(udt_s_call_upd_fetchfile);
#endif /* UPDATE_CANCEL_TEST */
    DFS_RPC_PREWRAP
    code = UPDATE_FetchFile (h, filename, pipeP);
    DFS_RPC_POSTWRAP(code)

#ifdef UPDATE_CANCEL_TEST
    Log(udt_s_cmpltd_upd_fetchfile);
#endif /* UPDATE_CANCEL_TEST */

    if ( IS_COMM_ERR(code) ) {
        if (UPDATE_CHECK_PROVIDER_OR_RANGE && code == rpc_s_op_rng_error){
            code = UP_BAD_INTERFACE;
        FUNCTION_NOOP = 1;
        }
    }
    return code;
    }
    else
      return(UP_BAD_INTERFACE);
}
```

The update_FetchInfo Wrapper

```
error_status_t
update_FetchInfo (h, name, pipeP)
    rpc_binding_handle_t h;
    NameString_t name;
    pipe_t *pipeP;
{
    error_status_t code;
    static unsigned long FUNCTION_PROVIDER = 1;
                    /* provider version # this RPC is supported */
    static unsigned long FUNCTION_NOOP = 0;
                    /* Is this function supported by server or not? */

    if (FUNCTION_NOOP) /* this RPC not supported */
        return (rpc_s_op_rng_error);

    if (UPDATE_INTERFACE == DFS_UPDATE &&
        UPDATE_PROVIDER >= FUNCTION_PROVIDER) {

#ifdef UPDATE_CANCEL_TEST
    Log(udt_s_call_upd_fetchinfo);
#endif /* UPDATE_CANCEL_TEST */
    DFS_RPC_PREWRAP
    code = UPDATE_FetchInfo (h, name, pipeP);
    DFS_RPC_POSTWRAP(code)

#ifdef UPDATE_CANCEL_TEST
    Log(udt_s_cmpltd_upd_fetchinfo);
#endif /* UPDATE_CANCEL_TEST */

    if ( IS_COMM_ERR(code) ) {
        if (UPDATE_CHECK_PROVIDER_OR_RANGE && code == rpc_s_op_rng_error){
          code = UP_BAD_INTERFACE;
        FUNCTION_NOOP = 1;
        }
    }
    return code;
    }
    else
      return(UP_BAD_INTERFACE);
}
```

The update_FetchObjectInfo Wrapper

```
error_status_t
update_FetchObjectInfo (h, name, buf)
    rpc_binding_handle_t h;
    char *name;
    updateFileStatT *buf;
{
    error_status_t code;
    static unsigned long FUNCTION_PROVIDER = 1;
```

```
                    /* provider version # this RPC is supported */
    static unsigned long FUNCTION_NOOP = 0;
                    /* Is this function supported by server or not? */
    NameString_t object;
                    /* For use in UPDATE_FetchObjectInfo call */

    if (FUNCTION_NOOP) /* this RPC not supported */
        return (rpc_s_op_rng_error);

    if (UPDATE_INTERFACE == DFS_UPDATE &&
        UPDATE_PROVIDER >= FUNCTION_PROVIDER) {

        strcpy((char *)object, name);

#ifdef UPDATE_CANCEL_TEST
    Log(udt_s_call_upd_fobjinfo);
#endif /* UPDATE_CANCEL_TEST */

    DFS_RPC_PREWRAP
    code = UPDATE_FetchObjectInfo (h, object, buf);
    DFS_RPC_POSTWRAP(code)

#ifdef UPDATE_CANCEL_TEST
    Log(udt_s_cmpltd_upd_fobjinfo);
#endif /* UPDATE_CANCEL_TEST */

    if (IS_COMM_ERR(code)) {
        if (UPDATE_CHECK_PROVIDER_OR_RANGE && code == rpc_s_op_rng_error){
        code = UP_BAD_INTERFACE;
        FUNCTION_NOOP = 1;
        }
    }
    return code;
    }
    else
      return(UP_BAD_INTERFACE);
}
```

Skeleton Client Main

```
int main(argc,argv)
    int   argc;
    char  **argv;
{
    ...

/* Initialize the serviceability function */

    initialize_svc();
    ...
}
```

5.5 Example Server Application

The server registers the interfaces with the runtime and with the versioning mechanism.

5.5.1 Typical Server Headers

```
#include <dcedfs/param.h>
#include <dcedfs/stds.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#include <dce/rpc.h>
#include <pthread.h>
#include <dcedfs/osi.h>
#include <dcedfs/compat.h>
#include <dcedfs/dfsauth.h>
#include <dcedfs/cmd.h>
#include <dirent.h>

#include <update.h>
#include <upcommon.h>
```

5.5.2 Headers for Serviceability

The following headers get included for serviceability. The last three are specific to this subcomponent. They would need to be defined for each file that has implemented serviceability, and the names would be chosen to be specific to the function. The first three are included as is regardless of what subcomponent they are in, as they are needed for general serviceability and are not specific to the subcomponent.

The explanation of serviceability is beyond the scope of this document and this example.

```
/* For serviceability */

#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <dce/dcesvcmsg.h>
#include <dfsudtmac.h>
#include <dfsudtsvc.h>
#include <dfsudtmsg.h>
```

5.5.3 Constants for the Server Application

```
#define UPDATE_MAXCALLS 5
#define DEFAULT_ADMIN_FILENAME  "admin.up"
```

5.5.4 Define IS_COMM_ERR Function

```
#define IS_COMM_ERR(s)  (s == rpc_s_ok ? 0 : 1$
```

5.5.5 Typical Serviceability Initialization

```
char *whoami = "upserver";

dce_svc_handle_t   udt_svc_handle;

void initialize_svc()
{
  error_status_t st;

  udt_svc_handle = dce_svc_register(udt_svc_table,  (idl_char *)"udt",  &st);
  if (st != svc_s_ok)
      fprintf(stderr, "Cannot register svc 0x%lx0, st);
  dce_svc_routing((unsigned char *) "NOTICE:STDOUT:--", &st);
  if (st != svc_s_ok)
      fprintf(stderr, "Cannot route NOTICE messages  0x%lx0, st);
  dce_svc_routing((unsigned char *) "WARNING:STDOUT:--", &st);
  if (st != svc_s_ok)
      fprintf(stderr, "Cannot route WARNING messages  0x%lx0, st);
  dce_svc_routing((unsigned char *) "ERROR:STDERR:--", &st);
  if (st != svc_s_ok)
      fprintf(stderr, "Cannot route ERROR messages  0x%lx0, st);

  dfs_define_udt_message_tables();
}
```

5.5.6 Skeleton Server Main

```
int
main(argc, argv)
    int   argc;
    char  *argv[];
{

    extern char *dfs_dceErrTxt();
    extern UPDATE_v4_0_epv_t UPDATE_v4_0_manager_epv;

    osi_setlocale(LC_ALL, "");

    ...

    initialize_svc();


    initialize_upd_error_table();
```

```
    ...

    us_argSetup();
    code = cmd_Dispatch(argc, argv);
    if ( code )
      exit(1);

    }

    initUpserverLog();
    updatePthreadInit();
```

Shutting Down the Duplicate Server

```
code = compat_ShutdownDuplicateServer (
    (rpc_if_handle_t)UPDATE_v4_0_s_ifspec,
    (uuid_t *)NULL,
    0);
if ( code) {
    LogError(udt_s_dup_server_shutdown_failed, code);
    goto exit;
}
```

Recording the Shutdown Registered Interface

The following example shows how to register an interface with the versioning mechanism.

```
/* recording the above registered interface */
dfs_installInterfaceDescription ((rpc_if_handle_t)UPDATE_v4_0_s_ifspec,
    (rpc_if_handle_t)UPDATE_v4_0_s_ifspec,
     1, /* provider version number */
    (unsigned_char_t *)"Transarc Update Server Interface",
     &st);

if (IS_COMM_ERR(st))
{
    LogError(udt_s_install_reg_intfc_failed, st);
    code = st;
    goto exit;
}

uuid_create_nil(&nil_uuid, &st);
if (st != uuid_s_ok) {
    LogError(udt_s_nil_UUID_create_failed, st);
    goto exit;
}
```

Registering the Interface with the Runtime

The following is an example of registering an interface with the RPC Runtime.

```
rpc_register_dfs_server ((rpc_if_handle_t)UPDATE_v4_0_s_ifspec,
    (rpc_mgr_epv_t)&UPDATE_v4_0_manager_epv,
    (uuid_t *)&nil_uuid, (uuid_t *)&nil_uuid,
    UPDATE_MAXCALLS, adminFilename,
    "DFS upserver", &st);
if (IS_COMM_ERR(st))
{
    LogError(udt_s_servr_regist_failed, st);
    code = st;
    goto exit;
}

Log(udt_s_upsrvr_started);

rpc_mgmt_set_server_com_timeout(rpc_c_binding_default_timeout+2, &st);
rpc_server_listen (UPDATE_MAXCALLS, &st);
if (IS_COMM_ERR(st))
{
    LogError(udt_s_srvr_listen_failed, st);
    code = st;
    goto exit;
}

code = compat_UnregisterServer ((rpc_if_handle_t)UPDATE_v4_0_s_ifspec,
    (uuid_t *)NULL);
if (code) {
    LogError(udt_s_unregister_intfc_failed, code);
    goto exit;
}
else exit(0);
```

Sample Error Exit for Preceeding Functions

```
exit:
   exit(2);
}
```

5.6 Example Manager Application

The manager implements all RPCs. An additional RPC that it should support is the RPC_GetServerInterfaces call which is called by the client to retrieve the server interfaces.

5.6.1 Typical MANAGER Headers

```
#include <dcedfs/param.h>
#include <dcedfs/stds.h>
#include <dcedfs/osi.h>

#include <netdb.h>
#include <netinet/in.h>

#include <dirent.h>

#ifdef  AFS_AIX_ENV
    /*
     *  in AIX valuable fields such as st_blksize are not in
     *  the stat structure
     */
#include <sys/statfs.h>
#endif /* AFS_AIX_ENV */

#include <dce/rpc.h>
#include <pthread.h>

#include <dcedfs/compat.h>
#include <dcedfs/dfsauth.h>

#include <update.h>
#include <upcommon.h>
```

5.6.2 Headers for Serviceability

The following headers get included for serviceability. The last three are specific to this subcomponent. They would need to be defined for each subcomponent that has implemented serviceability, and the names would be chosen to be specific to the function. The first three are included as is regardless of what subcomponent they are in, as they are needed for general serviceability and are not specific to the subcomponent.

The explanation of serviceability is beyond the scope of this document and this example.

```
/* For serviceability        */

#include <dce/dce.h>
#include <dce/dce_msg.h>
#include <dce/dcesvcmsg.h>
#include <dfsudtmac.h>
#include <dfsudtsvc.h>
#include <dfsudtmsg.h>
```

5.6.3 Typical Initialization

```
extern char *whoami;
extern pthread_mutex_t updateMutex;
```

5.6.4 ANSI C Declaration

```
/* Forward declarations for ANSI C */
long update_SendFile();
long update_SendDirInfo();
```

5.6.5 Example Mutex Lock

The following macro is used as an example to demonstrate locking that could be used.

Note: Locking is done in this example per exported procedure, rather than per subroutine call. While it serializes operations more than per subroutine locking, it is used here to demonstrate a consistent manner of locking.

```
#define    LOCK_UPDATE_MUTEX                        \
    if (pthread_mutex_lock (&updateMutex) != 0)     \
    {                                               \
        Log(udt_s_global_lock_locking_error,        \
                whoami, errno);                     \
        exit(1);                                    \
    }
```

5.6.6 Example Mutex Unlock

```
#define    UNLOCK_UPDATE_MUTEX                      \
    if (pthread_mutex_unlock (&updateMutex) != 0) \
    {                                               \
    Log(udt_s_global_lock_unlock_error,             \
        whoami, errno);                             \
    exit(1);                                        \
    }
```

5.6.7 Example Error Exit

The following example defines the common error exit that is used in the Manager's example exported procedures.

```
/* used for premature routine exit */
#define    ERROR(n)                                 \
{                                                   \
    code = n;                                       \
    goto error_exit;                                \
}
```

5.6.8 UPDATE_GetServerInterfaces

This example exported procedure sends a list of registered upserver interfaces to the caller.

```
error_status_t UPDATE_GetServerInterfaces (h, serverInterfaces)
    /* [in] */   rpc_binding_handle_t         h;
    /* [out] */ dfs_interfaceList *serverInterfaces;
{
    long code = 0, code2 = 0;
    static char    routineName[] = "UPDATE_FetchFile";
    int savedCancelState;

    DFS_DISABLE_CANCEL(&savedCancelState, code);
    if (code) {
        LogError(udt_s_disable_cancel_failed, code, whoami, routineName);
    serverInterfaces->dfs_interfaceList_len = 0;
    return(code);
    }

    code = dfsauth_server_CheckAuthorization(&h,
        (dacl_permset_t *)NULL,
        (char *)NULL,
        (uuid_t *)NULL);
    if ( code )
    {
    if ( code != DAUT_ERROR_ACCESS_DENIED) {
        LogError(udt_s_auth_check_failed, code, whoami, routineName);
    }
    serverInterfaces->dfs_interfaceList_len = 0;
    ERROR(code);
    }
    dfs_GetServerInterfaces (UPDATE_v4_0_s_ifspec, serverInterfaces,
        (unsigned32 *)&code);
    if (code) {
      LogError(udt_s_getserver_intfcs_failed, code, whoami, routineName);
      ERROR(code);
    }

error_exit:
    DFS_ENABLE_CANCEL(savedCancelState, code2);
    if (code2) {
        LogError(udt_s_enable_cancel_failed, code2, whoami, routineName);
    serverInterfaces->dfs_interfaceList_len = 0;
    }
    if (code)
      return code;
    else return code2;
}
```

5.6.9 UPDATE_FetchFile

This example exported procedure sends the contents of a file to the caller.

```
error_status_t UPDATE_FetchFile(h, FileName, pipeStream)
    /* [in] */   rpc_binding_handle_t h;
    /* [in] */   NameString_t         FileName;
    /* [out] */  pipe_t               *pipeStream;
{
    int     fd = -1;
    int     lockHeld = 0;
    struct  stat status;
    char    *pipeBuffP = (char *)0;
    long code = 0, code2= 0;
    static char    routineName[] = "UPDATE_FetchFile";
    int savedCancelState;

    /* place a TRY/FINALLY around critical region */
    TRY
    {
    DFS_DISABLE_CANCEL(&savedCancelState, code);
    if (code) {
        /* terminate the pipe */
        pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);

        LogError(udt_s_pipe_disable_cancel_failed, code, whoami,
            routineName);
        ERROR(code);
    }

    LOCK_UPDATE_MUTEX;
    lockHeld = 1;
    code = dfsauth_server_CheckAuthorization(&h,
        (dacl_permset_t *)NULL,
        (char *)NULL,
        (uuid_t *)NULL);
    if ( code )
    {
        if ( code != DAUT_ERROR_ACCESS_DENIED) {
        LogError(udt_s_daut_error_auth_failed, code, whoami, routineName);
        }

        /* terminate the pipe */
        pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);
        ERROR(code);
    }
    if ( canExportObject(FileName) == 0 )
    {
        /* terminate the pipe */
        pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);

        Log(udt_s_req_not_in_list, FileName);
        ERROR(UP_NOT_EXPORTABLE);
```

```
        }

        fd = open(FileName, O_RDONLY, 0);       /* Open the target file */
        if (fd < 0 || fstat(fd, &status) < 0) {
            /* terminate the pipe */
            pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);

            Log(udt_s_open_file_failed, FileName);
            ERROR(UP_SOFT_ERROR);
        }
        code = update_SendFile(fd, pipeStream, &status);
        if (fd >=0)
            close(fd);

        error_exit:;
        }
        FINALLY
        {
        if (lockHeld)
            UNLOCK_UPDATE_MUTEX;
        DFS_ENABLE_CANCEL(savedCancelState, code2);
        }
        ENDTRY

        if (code2)
            LogError(udt_s_enable_cancel_failed2, code2, whoami, routineName);
        if (code)
            return code;
        else
            return code2;
    }
```

5.6.10 UPDATE_FetchInfo

This example exported procedure fetches directory information about directory name and sends it to remote client.

```
error_status_t UPDATE_FetchInfo(h, DirName, pipeStream)
    /* [in] */   rpc_binding_handle_t h;
    /* [in] */   NameString_t         DirName;
    /* [out] */  pipe_t               *pipeStream;
{
    int fd = -1;
    int lockHeld = 0;
    struct stat status;
    char   *pipeBuffP = (char *)0;
    long code = 0, code2 = 0;
    static char    routineName[] = "UPDATE_FetchInfo";
    int savedCancelState;

    TRY
    {
    DFS_DISABLE_CANCEL(&savedCancelState, code);
```

```
if (code) {
    /* terminate the pipe */
    pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);

    LogError(udt_s_disable_cancel_failed2, code, whoami, routineName);
    ERROR(code);
}

LOCK_UPDATE_MUTEX;
lockHeld = 1;
code = dfsauth_server_CheckAuthorization(&h,
    (dacl_permset_t *)NULL,
    (char *)NULL,
    (uuid_t *)NULL);
if ( code )
{
    if (code != DAUT_ERROR_ACCESS_DENIED) {
    LogError(udt_s_daut_error_auth_failed2, code, whoami, routineName);
    }

    /* terminate the pipe */
    pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);
    ERROR(code);
}
if ( canExportObject(DirName) == 0 )
{
    /* terminate the pipe */
    pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);

    Log(udt_s_req_not_in_list2, DirName);
    ERROR(UP_NOT_EXPORTABLE);
}

fd = open(DirName, O_RDONLY, 0);
if (fd < 0 || fstat(fd, &status) < 0) {
    /* terminate the pipe */
    pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);

    Log(udt_s_open_dir_failed, DirName);
    ERROR(UP_SOFT_ERROR);
}

if((status.st_mode & S_IFMT) != S_IFDIR){
    /* terminate the pipe */
    pipeStream->push(pipeStream->state, (unsigned char *)pipeBuffP, 0);

    Log(udt_s_obj_not_dir,DirName);
    ERROR(ENOENT);
}
code  =  update_SendDirInfo(DirName, pipeStream, &status);

error_exit:;
```

```
    }
    FINALLY
    {
    if (fd >= 0)
        close(fd);
    if (lockHeld)
        UNLOCK_UPDATE_MUTEX;

    DFS_ENABLE_CANCEL(savedCancelState, code2);
    }
    ENDTRY

    if (code2)
      LogError(udt_s_enable_cancel_failed3, code2, whoami, routineName);
    if (code)
      return code;
    else
      return code2;
}
```

5.6.11 UPDATE_FetchObjectInfo

This example exported procedure returns a stat of the named object. Selected items from the stat are returned to the caller.

```
error_status_t
UPDATE_FetchObjectInfo(h, objectName, fileStatP)
    /* [in] */  rpc_binding_handle_t h;
    /* [in] */  NameString_t         objectName;
    /* [out]*/  updateFileStatP      fileStatP;
{
    struct stat statBuf;
    long code = 0, code2 = 0;
    int savedCancelState;
    static char    routineName[] = "UPDATE_FetchObjectInfo";

    DFS_DISABLE_CANCEL(&savedCancelState, code);
    if (code) {
        LogError(udt_s_disable_cancel_failed4, code, whoami, routineName);
    return(code);
    }

    LOCK_UPDATE_MUTEX;
#ifdef notdef
    code = dfsauth_server_CheckAuthorization(&h,
        (dacl_permset_t *)NULL,
        (char *)NULL,
        (uuid_t *)NULL);

    if ( code )
    {
        if (code != DAUT_ERROR_ACCESS_DENIED) {
            LogError(udt_s_daut_error_auth_failed4, code, whoami,
```

```
                routineName);
            }
        ERROR(code);
        }
    #endif
        if ( canExportObject(objectName) == 0 )
        {
            Log(udt_s_obj_export_failed, objectName);
        ERROR(UP_NOT_EXPORTABLE);
        }
        code = stat((char *)objectName, &statBuf);
        if ( code )
        {
        Log(udt_s_stat_obj_failed, objectName, errno);
        ERROR(UP_SOFT_ERROR);
        }

        /* copy the information back to the caller's structure */
        hset32(fileStatP->fileLength, statBuf.st_size);
        fileStatP->mode =   statBuf.st_mode;
        fileStatP->uid =    statBuf.st_uid;
        fileStatP->gid =    statBuf.st_gid;
        fileStatP->mtime =  statBuf.st_mtime;
        fileStatP->atime =  statBuf.st_atime;

    error_exit:
        UNLOCK_UPDATE_MUTEX;

        DFS_ENABLE_CANCEL(savedCancelState, code2);
        if (code2)
            LogError(udt_s_enable_cancel_failed4, code2, whoami, routineName);
        if (code)
            return code;
        else
            return code2;
    }
```

5.6.12 UPDATE_v4_0_manager_epv

This example declares and initializes the example manager's entry point vector.

```
UPDATE_v4_0_epv_t UPDATE_v4_0_manager_epv = {
                      UPDATE_GetServerInterfaces,
                      UPDATE_FetchInfo,
                      UPDATE_FetchFile,
                      UPDATE_FetchObjectInfo};
```

5.7 Extending the DFS Interface

The owner of an interface implies it is the entity that created the interface with a UUID, major and minor version numbers. Other entities can extend the interface. When an entity extends an interface not owned by it, it needs to create a new interface (UUID, major and minor version numbers) and add new operations in addition to the operations from the interface that is being extended. The DFS interfaces are owned by Transarc Corporation and others would need to create a new interface in order to extend any of the original DFS interfaces.

There are two cases involved in extending an interface. One case is when an owner of an interface extends it; the other is when an entity that is not the owner of an interface extends it. These two cases involve the following changes:

5.7.1 Case 1 — An Owner Extends a DFS Interface

When an owner extends a DFS interface, the changes needed are the following:

1. Deciding either to bump the minor version number or the provider version number. This decision is made based upon whether the changes are extensive enough to warrant it's being a new version or not. If extensive, then the provider version number would be increased by one which connotes the meaning of being bumped. Otherwise, increasing or bumping the minor version number by one would be sufficient.
2. If the minor version number is bumped, the change is recorded in the IDL file header. Also a change is to be made in the *dfs_selectInterface*( ) function in the client, to select the appropriate interface(stub) to call.
3. If the provider version number is bumped, then the IDL file header should have the new provider version number, and each added RPC should have a comment indicating its provider version level. The server should indicate its provider level when calling *dfs_installInterfaceDescription*( ). Servers must maintain compatability with clients running at lower provider version levels. Clients must not rely on the availability of any RPCs other than those in provider version 1. When a client makes an RPC to a server running at a lower provider version level, the request is handled by the RPC runtime environment, and the client gets an [rpc_s_op_rng_error]. Clients should be written so that they can recover from this error by retrying the operation using the functionality available in provider version 1.

   Clients should keep an estimate of the provider version level of each server. Initially, this estimate should be the provider version level desired by the client. If an [rpc_s_op_rng_error] is received for an RPC at provider version N, the server's provider version estimate should be set to N−1. The client should not make any more RPC calls to that server at a provider version higher than N−1. The server's provider version estimate should be reset whenever the server's binding handle is reset.

   As an example, the Ubik client interface includes support for server provider versioning, and maintains an estimate of each server's provider version level. Ubik clients making RPCs for provider versions other than provider version 1 should use *ubik_PVCall*( ) instead of *ubik_Call*( ). *ubik_PVCall*( ) has a parameter for the provider version level of the RPC. The ubik client library keeps an estaimte of the provider version level of each server, and returns [UDOWNLVL] when an RPC fails because the server is not at the desired version level. Clients should be written to recover from [UDOWNLVL] errors by retrying the operation using the functionality in provider version 1.

5.7.2 Case 2 — A Non-owner Extends a DFS Interface

An entity not owning an interface will herein be called an agent. When an entity extends a DFS interface not owned by it, the changes needed are the following:

1. If the agent modifies an interface not owned (created) by it, then a new Interface UUID (and a new IDL file) should be created. The RPCs in the original interface are supported in the new interface (with a distinguishing prefix) and new RPCs are appended to the older ones.
2. The original DFS interface (Transarc's) and the newly created interface are registered with the RPC runtime and the versioning mechanism. The agent has the option of supporting any intermediate interfaces.
3. A new manager end point vector is created for the new interface with the old entries and the newly added RPCs appended to this. The old entries could refer to functions in any of the managers associated with the interfaces other than the newly added one. This way, the new manager need not duplicate functions already implemented in the previous interfaces.
4. The *dfs_selectInterface*( ) function in the client has to be modified to select the new interface. If not avaliable, then the client could use one of the older interfaces supported by the client, and the wrapper for the newly added RPCs could check the interface global to determine if the RPC could be made successfully or not.
5. The wrappers for the new RPCs should check for the new interface. If not supported, then the new RPCs should fail.

X/Open Preliminary Specification

Part 4:
DFS Client/Server and Token Manager Interfaces

Chapter 6

DCE DFS File Exporter Interface

This chapter describes the RPCs that can be issued against the file exporter via a client cache manger. These RPCs allow the cache manager to manipulate files on a file server machine.

The error codes passed back from the DFS server to the DFS client do not come through [ERRNO], and are not the values from the server's version of **<errno.h>**, but are an architected set that the client will then map into local error codes.

The return codes returned by these calls are the union of the set of errors returned by the RPC, the token manager, and an abstraction of the DFS server's vnode layer itself (for example, DFS_ENOENT for file not found). The full set of these abstractions of the DFS server's error codes is as follows:

```
#define DFS_ESUCCESS        0    /* Successful */
#define DFS_EPERM           1    /* Operation not permitted */
#define DFS_ENOENT          2    /* No such file or directory */
#define DFS_ESRCH           3    /* No such process */
#define DFS_EINTR           4    /* Interrupted system call */
#define DFS_EIO             5    /* I/O error */
#define DFS_ENXIO           6    /* No such device or address */
#define DFS_E2BIG           7    /* Arg list too long */
#define DFS_ENOEXEC         8    /* Exec format error */
#define DFS_EBADF           9    /* Bad file number */
#define DFS_ECHILD          10   /* No children */
#define DFS_EDEADLK         11   /* Operation would cause deadlock */
#define DFS_ENOMEM          12   /* Not enough core */
#define DFS_EACCES          13   /* Permission denied */
#define DFS_EFAULT          14   /* Bad address */
#define DFS_ENOTBLK         15   /* Block device required */
#define DFS_EBUSY           16   /* Mount device busy */
#define DFS_EEXIST          17   /* File exists */
#define DFS_EXDEV           18   /* Cross-device link */
#define DFS_ENODEV          19   /* No such device */
#define DFS_ENOTDIR         20   /* Not a directory */
#define DFS_EISDIR          21   /* Is a directory */
#define DFS_EINVAL          22   /* Invalid argument */
#define DFS_ENFILE          23   /* File table overflow */
#define DFS_EMFILE          24   /* Too many open files */
#define DFS_ENOTTY          25   /* Not a typewriter */
#define DFS_ETXTBSY         26   /* Text file busy */
#define DFS_EFBIG           27   /* File too large */
#define DFS_ENOSPC          28   /* No space left on device */
#define DFS_ESPIPE          29   /* Illegal seek */
#define DFS_EROFS           30   /* Read-only file system */
#define DFS_EMLINK          31   /* Too many links */
#define DFS_EPIPE           32   /* Broken pipe */
#define DFS_EDOM            33   /* Argument too large */
#define DFS_ERANGE          34   /* Result too large */
#define DFS_EWOULDBLOCK     35   /* Operation would block */
#define DFS_EINPROGRESS     36   /* Operation now in progress */
#define DFS_EALREADY        37   /* Operation already in progress */
```

```
/*
 * ipc/network software
 */
#define DFS_ENOTSOCK        38   /* Socket operation on non-socket */
#define DFS_EDESTADDRREQ    39   /* Destination address required */
#define DFS_EMSGSIZE        40   /* Message too long */
#define DFS_EPROTOTYPE      41   /* Protocol wrong type for socket */
#define DFS_ENOPROTOOPT     42   /* Protocol not available */
#define DFS_EPROTONOSUPPORT 43   /* Protocol not supported */
#define DFS_ESOCKTNOSUPPORT 44   /* Socket type not supported */
#define DFS_EOPNOTSUPP      45   /* Operation not supported on socket */
#define DFS_EPFNOSUPPORT    46   /* Protocol family not supported */
#define DFS_EAFNOSUPPORT    47   /* Address family not supported */
#define DFS_EADDRINUSE      48   /* Address already in use */
#define DFS_EADDRNOTAVAIL   49   /* Can't assign requested address */
#define DFS_ENETDOWN        50   /* Network is down */
#define DFS_ENETUNREACH     51   /* Network is unreachable */
#define DFS_ENETRESET       52   /* Network dropped conn on reset */
#define DFS_ECONNABORTED    53   /* Software caused connection abort */
#define DFS_ECONNRESET      54   /* Connection reset by peer */
#define DFS_ENOBUFS         55   /* No buffer space available */
#define DFS_EISCONN         56   /* Socket is already connected */
#define DFS_ENOTCONN        57   /* Socket is not connected */
#define DFS_ESHUTDOWN       58   /* Can't send after socket shutdown */
#define DFS_ETOOMANYREFS    59   /* Too many references: can't splice */
#define DFS_ETIMEDOUT       60   /* Connection timed out */
#define DFS_ECONNREFUSED    61   /* Connection refused */
#define DFS_ELOOP           62   /* Too many levels of symbolic links */
#define DFS_ENAMETOOLONG    63   /* File name too long */
#define DFS_EHOSTDOWN       64   /* Host is down */
#define DFS_EHOSTUNREACH    65   /* No route to host */
#define DFS_ENOTEMPTY       66   /* Directory not empty */
#define DFS_EPROCLIM        67   /* Too many processes */
#define DFS_EUSERS          68   /* Too many users */
#define DFS_EDQUOT          69   /* Disc quota exceeded */
/*
 *  NFS errors.
 */
#define DFS_ESTALE          70   /* Stale NFS file handle */
#define DFS_EREMOTE         71   /* Too many levels of remote in path */
#define DFS_EBADRPC         72   /* RPC struct is bad */
#define DFS_ERPCMISMATCH    73   /* RPC version wrong */
#define DFS_EPROGUNAVAIL    74   /* RPC prog. not avail */
#define DFS_EPROGMISMATCH   75   /* Program version wrong */
#define DFS_EPROCUNAVAIL    76   /* Bad procedure for program */
/*
 * misc ..
 */
#define DFS_ENOLCK          77   /* No locks available */
#define DFS_ENOSYS          78   /* Function not implemented */
#define DFS_EAGAIN          79   /* Resource temporarily unavailable */
/*
```

```
 * SYS V IPC errors
 */
#define DFS_ENOMSG          80   /* No msg matches receive request */
#define DFS_EIDRM           81   /* Msg queue id has been removed */
/*
 * STREAMS
 */
#define DFS_ENOSR           82   /* Out of STREAMS resources */
#define DFS_ETIME           83   /* System call timed out */
#define DFS_EBADMSG         84   /* Next message has wrong type */
#define DFS_EPROTO          85   /* STREAMS protocol error */
#define DFS_ENODATA         86   /* No message on stream head read q */
#define DFS_ENOSTR          87   /* fd not associated with a stream */
/*
 * Not visible outside kernel
 */
#define DFS_ECLONEME        88   /* Tells open to clone the device */
/*
 *  Filesystem
 */
#define DFS_EDIRTY          89   /* Mounting a dirty fs w/o force */
/*
 * Loader errors
 */
#define DFS_EDUPPKG         90   /* duplicate package name on install */
#define DFS_EVERSION        91   /* version number mismatch */
#define DFS_ENOPKG          92   /* unresolved package name */
#define DFS_ENOSYM          93   /* unresolved symbol name */
/*
 * To be filled
 */
#define DFS_SPARE94         94
#define DFS_SPARE95         95
#define DFS_SPARE96         96
#define DFS_SPARE97         97
#define DFS_SPARE98         98
#define DFS_SPARE99         99
#define DFS_SPARE100        100
#define DFS_SPARE101        101
#define DFS_SPARE102        102
#define DFS_SPARE103        103
#define DFS_SPARE104        104
#define DFS_SPARE105        105
#define DFS_SPARE106        106
#define DFS_SPARE107        107
#define DFS_SPARE108        108
#define DFS_SPARE109        109
#define DFS_SPARE110        110
#define DFS_SPARE111        111
/*
 * security
 */
```

```
#define DFS_ENOATTR         112  /* no attribute found */
#define DFS_ESAD            113  /* security authentication denied */
#define DFS_ENOTRUST        114  /* not a trusted program */
/*
 * To be filled
 */
#define DFS_SPARE115        115
#define DFS_SPARE116        116
#define DFS_SPARE117        117
#define DFS_SPARE118        118
#define DFS_SPARE119        119
#define DFS_SPARE120        120
#define DFS_SPARE121        121
#define DFS_SPARE122        122
/*
 * Internal Disk/Block Device error codes
 */
#define DFS_ESOFT           123  /* I/O completed but needs relocation*/
#define DFS_EMEDIA          124  /* media surface error */
#define DFS_ERELOCATED      125  /* a relocation request performed ok */
#define DFS_SPARE126        126
#define DFS_SPARE127        127
/*
 * Errors not defined by local kernel.
 */
#define DFS_ENOTDEFINED     128
```

NAME

AFS_SetContext — Establish client context

SYNOPSIS

```
error_status_t  AFS_SetContext(
    /* IN  */ handle_t    h,
    /* IN  */ unsigned32  epochTime,
    /* IN  */ afsNetData  *callbackAddr,
    /* IN  */ unsigned32  Flags,
    /* IN  */ afsUUID     *secObjectID,
    /* IN  */ unsigned32  clientSizeAttrs,
    /* IN  */ unsigned32  parm7
    );
```

ARGUMENTS

h	The RPC binding handle.
epochTime	The restart time of the DFS client.
callbackAddr	The RPC endpoint of the client, for token revocation purposes.
Flags	If 0 then the call only defines a primary interface UUID. If the flag, AFS_FLAG_SEC_SERVICE, is set then a a secondary interface is defined and is stored in the parameter, *secObjectID*.
secObjectID	When the AFS_FLAG_SEC_SERVICE flag is set, this parameter holds the secondary interface UUID.
clientSizeAttrs	For 64/32-bit compatibility. Through DCE1.1, this is a spare parameter, *parm6*.
parm7	A spare parameter.

DESCRIPTION

This function establishes some context information at the file exporter that is required before this client can execute calls at the server. If a server crashes, the client must make a new AFS_SetContext call before it makes other calls to this server. The UUID's passed via this function must have been obtained via the *dfsuuid_Create*( ) function.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.

NAME

AFS_LookupRoot — Look up fileset root on file server

SYNOPSIS

```
error_status_t  AFS_LookupRoot(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *InFidp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ signed32    Flags,
    /* OUT */ afsFid      *OutFidp,
    /* OUT */ afsFetchStatus *OutFidStatusp,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
InFidp	The file identifier specifying the fileset whose root directory will be retrieved from the file server.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier describing the root directory of the selected fileset.
OutFidStatusp	Returns the status of that directory after the current operation.
OutTokenp	The promise the file server returns to the cache manager about the provided data; this is only returned if the file resides in a read/write fileset.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call is used to obtain the description of the root directory of a fileset from the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The file specified by *inFidp* does not exist.
[DFS_EACCES]	The caller lacks read permission on *inFidp*.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_FetchData — Retrieve data from file server

SYNOPSIS

```
error_status_t  AFS_FetchData(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid     *Fidp,
    /* IN  */ afsHyper   *minVVp,
    /* IN  */ afsHyper   *Position,
    /* IN  */ signed32   Length,
    /* IN  */ unsigned32 Flags,
    /* OUT */ afsFetchStatus *OutStatusp,
    /* OUT */ afsToken   *OutTokenp,
    /* OUT */ afsVolSync *Syncp,
    /* OUT */ pipe_t     *fetchStream
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	The file identifier specifying the file whose contents will be retrieved from the file server.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Position	Specifies the first byte to be fetched by this call with zero (0) being the first byte in the file.
Length	Specifies the number of bytes desired with the value 0xFFFFFFFF indicating the entire file contents.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutStatusp	Returns the status of the file after the current operation.
OutTokenp	The promise the file server returns to the cache manager about the provided data; this is only returned if the file resides in a read/write fileset.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.
fetchStream	The character pipe parameter returning the data from the file. (See **X/Open DCE: Remote Procedure Call** specification for implementation details.)

DESCRIPTION

This call is used to obtain the contents of the specified file from the file server.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The file specified by *Fidp* does not exist.

[DFS_EACCES]	The caller lacks read permission on *Fidp*.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_FetchACL — Retrieve Access Control List

SYNOPSIS

```
error_status_t  AFS_FetchACL(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *Fidp,
    /* IN  */ unsigned32  aclType,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsACL      *AccessListp,
    /* OUT */ afsFetchStatus *OutStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	The file identifier specifying the file or directory whose access control list (ACL) will be retrieved from the file server.
aclType	The type of the access list being modified. One of VNX_ACL_REGULAR_ACL, VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL.
minVVp	The minimum-acceptable version number on the fileset containing this ACL.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
AccessListp	The access control list returned by the file server for the specified file or directory.
OutStatusp	Returns the current status of the file or directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

Note that a new token is not returned by this call; however, any existing tokens remain in effect.

See *Access Control List Overview* for a description of access list encoding.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The file, dir or symlink specified by *Fidp* does not exist.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_FetchStatus — Obtain file status information

SYNOPSIS

```
error_status_t  AFS_FetchStatus(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *Fidp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFetchStatus *OutStatusp,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	The file identifier specifying the file or directory whose status information will be retrieved from the file server.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutStatusp	Returns the current status of the file or directory.
OutTokenp	Returns a token from the file server reflecting guarantees granted by the file server.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call retrieves the status information associated with the specified file or directory. The file server returns a token pertaining to this file if the fileset upon which it resides is read/write.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The file named by *Fidp* does not exist.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_StoreData — Write data to file

SYNOPSIS

```
error_status_t  AFS_StoreData(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *Fidp,
    /* IN  */ afsFetchStatus *InStatusp,
    /* IN  */ afsHyper    *Position,
    /* IN  */ signed32    Length,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* IN  */ pipe_t      *storeStream,
    /* OUT */ afsStoreStatus *OutStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	The file identifier specifying the file whose status information will be updated from the file server.
InStatusp	The new status information that should be recorded for this file.
Position	Represents the position of the first byte of the data block.
Length	Represents the total length of the transferred data block.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
storeStream	The data stream containing the file updates. (See the **X/Open DCE: Remote Procedure Call** specification for implementation details.)
OutStatusp	Returns the status of the file.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call writes the specified data back to the file server for the specified file.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The file specified by *Fidp* does not exist.
[DFS_EACCES]	The caller lacks write permission to perform this operation.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.

[TKM_ERROR*] A token managment failure occurred.

NAME

AFS_StoreACL — Update access control information for a file

SYNOPSIS

```
error_status_t  AFS_StoreACL(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *Fidp,
    /* IN  */ afsACL      *AccessListp,
    /* IN  */ unsigned32  aclType,
    /* IN  */ afsFid      *aclFidp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFetchStatus *OutStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	The file identifier specifying the file or directory whose access control information will be updated from the file server.
AccessListp	The access control list sent to the file server for the specified file or directory.
aclType	The type of access control list being modified (VNX_ACL_REGULAR_ACL, VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL, in the low-order 8 bits. In the next-higher-order 8 bits, the type of access control list being copied from the file described by *aclFidp*.
aclFidp	The file identifier specifying the file or directory whose access control information will be copied.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutStatusp	Returns the updated file or directory status.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call updates the access control information associated with a file or directory. It can either set an ACL to a specific value via *AccessListp* or copy an ACL from that possessed by another file via *aclFidp*.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The file specified by *Fidp* does not exist.

[DFS_EACCES]	The caller lacks the required access to this file.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_StoreStatus — Update status information

SYNOPSIS

```
error_status_t  AFS_StoreStatus(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *Fidp,
    /* IN  */ afsStoreStatus *InStatusp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFetchStatus *OutStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	The file identifier specifying the file or directory whose access control information will be updated from the file server.
InStatusp	Contains the new status information for the specified file or directory.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutStatusp	Contains the updated status information for the file or directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call updates the status information associated with the file or directory.

It returns the new status information because this operation may change some status fields.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The entry named by *Fidp* does not exist.
[DFS_EACCES]	The caller lacks the required access rights.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_RemoveFile — Remove a file or symbolic link

SYNOPSIS

```
error_status_t  AFS_RemoveFile(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsFidTaggedName *Namep,
    /* IN  */ afsHyper    *returnTokenIDp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsFetchStatus *OutFileStatusp,
    /* OUT */ afsFid      *OutFileFidp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory from which to remove the file.
Namep	The complex name of the file to delete.
returnTokenIDp	A token ID being returned, if any.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutDirStatusp	Contains the updated directory status information.
OutFileStatusp	Contains the updated file status information.
OutFileFidp	The file id of the file which was actually deleted.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call removes a file or symbolic link, but not a directory, from the file system. The complex name **afsFidTaggedName** *must* include the string name component - it should be filled in with both a **fid** and an **afsTaggedName**. The **afsTaggedName** should have a codesetTag (*tn_tag*) value of 0 as nothing else is presently defined, and a valid *tn_length* field (if something other than zero was specified for *tn_tag*).

The cache manager is responsible for decrementing the link count in the file's associated cached status by 1.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL] The caller passed in some invalid argument.

[DFS_ENOENT]	The entry named by *DirFidp* does not exist, or the entry named by *Namep* does not exist.
[DFS_EACCES]	The caller lacks the required access rights.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_Lookup — Obtain directory entry for a file

SYNOPSIS

```
error_status_t  AFS_Lookup(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsTaggedName *Namep,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFid      *OutFidp,
    /* OUT */ afsFetchStatus *OutFidStatusp,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory from which to obtain the directory information.
Namep	The character string name of the file for which information is being requested.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier of the requested file.
OutFidStatusp	Status information for the specified file.
OutDirStatusp	Status information of the specified directory.
OutTokenp	The token against the directory (allowing the directory entry to be cached).
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call takes a directory identifier and a file name, and returns the file ID of the named file. It returns the status of the directory at the time the operation was performed, as well as a callback on that directory, so that this mapping can be cached.

A non-existent entry is denoted *not* by a special error code, but instead by a special file ID (represented as all zeroes), so that a callback can be returned for entries that don't exist. Caching the non-existence of entries may turn out to be important in some cases.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL] The caller passed in some invalid argument.

[DFS_ENOENT]	The entry named by *DirFidp* does not exist (note that entries that don't exist are handled by success returns).
[DFS_EACCES]	The caller lacks the required access rights to the directory.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_CreateFile — Create a file

SYNOPSIS

```
error_status_t  AFS_CreateFile(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsTaggedName *Namep,
    /* IN  */ afsStoreStatus *InStatusp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFid      *OutFidp,
    /* OUT */ afsFetchStatus *OutFidStatusp,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which to create the requested file.
Namep	The character string name of the file to create.
InStatusp	Specifies the initial status fields for the new file.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier of the newly created file.
OutFidStatusp	The status fields for the newly created file.
OutDirStatusp	The status information of the specified directory.
OutTokenp	A new token granted against the new file.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call is used to create a file, but not a symbolic link or a directory.

If the call succeeds, it is the cache manager's responsibility to either create an entry locally in the directory specified by *DirFid*, or to invalidate this directory's cache entry.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT]	The entry named by *DirFidp* does not exist.
[DFS_EACCES]	The caller lacks the required access rights.

[DFS_EEXIST]	An entry named *Namep* already exists in this dir.
[DFS_EINVAL]	The entry is named ‘‘.’’ or ‘‘..’’, or the caller passed in some invalid argument.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_Rename — Rename a file, directory or symbolic link

SYNOPSIS

```
error_status_t  AFS_Rename(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *OldDirFidp,
    /* IN  */ afsFidTaggedName *OldNamep,
    /* IN  */ afsFid      *NewDirFidp,
    /* IN  */ afsFidTaggedName *NewNamep,
    /* IN  */ afsHyper    *returnTokenIDp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFetchStatus *OutOldDirStatusp,
    /* OUT */ afsFetchStatus *OutNewDirStatusp,
    /* OUT */ afsFid      *OutOldFileFidp,
    /* OUT */ afsFetchStatus *OutOldFileStatusp,
    /* OUT */ afsFid      *OutNewFileFidp,
    /* OUT */ afsFetchStatus *OutNewFileStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h The RPC binding handle.

OldDirFidp The file identifier specifying the directory in which the file is currently located.

OldNamep The complex name of the file or directory to rename.

NewDirFidp The file identifier specifying the directory into which the file is to be moved.

NewNamep The complex name of the file or directory after it is moved.

returnTokenIDP A token ID being returned, if any.

minVVp The minimum-acceptable version number on the fileset containing this file or directory.

Flags Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

OutOldDirStatusp
The status information of the old directory (the one from which the file is being moved) upon termination of the call.

OutNewDirStatusp
The status information of the new directory (the one to which the file is being moved) upon termination of the call.

OutOldFileFidp The file identifier for the file which was moved.

OutOldFileStatusp
The status information of the file which was moved.

OutNewFileFidp The file identifier for the file to which the file identified in *OutOldFileFidp* was, in fact, moved.

OutNewFileStatusp
The status information of the file identified by *OutNewFileFidp*.

Syncp The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call renames the specified file or directory, located in the specified directory, to be the new file or directory located in the newly specified directory. The updated directory status for both directories is returned. If the two directories are the same, identical status information is returned in each status output parameter.

The rename must not result in hard links existing to the same object from two different filesets, or the error code EXDEV will be returned.

If a directory is moved from one directory to another, the cache manager must either update the cached copy of the moved directory in order to update its '..' entry, or the cache manager must invalidate the cache entry for the moved directory. The directory link counts will be updated by the server in the returned directory status blocks.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT] The entry named by *Fidp* does not exist.

[DFS_EACCES] The caller lacks the required access rights.

[DFS_ENOTEMPTY]
The target specifies a non-empty directory.

[DFS_ENOTDIR] The source specifies a directory but the target is not a directory.

[DFS_EISDIR] The source specifies a file but the target specifies a directory.

[DFS_EXDEV] Attempting to rename across filesets.

[DFS_EINVAL] The entry is named ''.'' or ''..'', or the caller passed in some invalid argument.

[FSHS_ERR*] File server does not recognize client, or is in some form of recovery.

[TKM_ERROR*] A token managment failure occurred.

NAME

AFS_Symlink — Create a symbolic link

SYNOPSIS

```
error_status_t  AFS_Symlink(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsTaggedName *Namep,
    /* IN  */ afsTaggedPath *LinkContentsp,
    /* IN  */ afsStoreStatus *InStatusp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFid      *OutFidp,
    /* OUT */ afsFetchStatus *OutFidStatusp,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which the symbolic link is to be created.
Namep	The name of the link to create.
LinkContentsp	The target of the new symbolic link.
InStatusp	This specifies the clientModTime field and unixModeBits of the new link.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier for the newly created symbolic link.
OutFidStatusp	The status information of the newly created symbolic link upon termination of the call.
OutDirStatusp	The status information of the directory (the one in which the symbolic link was created) upon termination of the call.
OutTokenp	The token returned against the specified directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call is used to create a symbolic link.

No token is granted on the new symbolic link because symbolic links cannot change, they can only be deleted. It is recommended that the cache manager make use of this fact.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT] The entry named by *DirFidp* does not exist.

[DFS_EACCES] The caller lacks the required access rights.

[DFS_EINVAL] The entry is named ‘‘.’’ or ‘‘..’’, or the caller passed in some invalid argument.

[DFS_EEXIST] The directory already contains an element named *Namep*.

[FSHS_ERR*] File server does not recognize client, or is in some form of recovery.

[TKM_ERROR*] A token managment failure occurred.

NAME

AFS_MakeMountPoint — Make an AFS mount point

SYNOPSIS

```
error_status_t  AFS_MakeMountPoint(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsTaggedName *Namep,
    /* IN  */ afsTaggedName *cellNamep,
    /* IN  */ afsFStype   Type,
    /* IN  */ afsTaggedName *volumeNamep,
    /* IN  */ afsStoreStatus *InStatusp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFid      *OutFidp,
    /* OUT */ afsFetchStatus *OutFidStatusp,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which the new mount point is to be created.
Namep	The name of the mount point.
cellNamep	The string name of the cell in which the mount point is being created.
Type	The type of the file system mount point.
volumeNamep	The name of the fileset to be mounted on the newly created mount point.
InStatusp	This specifies the clientModTime field and unixModeBits of the new mount point.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier for the newly created mount point.
OutFidStatusp	The status information of the newly created mount point upon termination of the call.
OutDirStatusp	The status information of the directory (the one in which the mount point was created) upon termination of the call.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This function call is obsolete. This call was used to create a new mount point for the specified fileset.

Note that this call does *not* require that the underlying file system support a separate file type for this type of object; for instance, in previous versions of AFS, these semantics were supported by

using a symbolic link with certain special attributes. This interface simply isolates the client Cache Manager from the server's implementation of **DCE DFS** Ename mount points.

RETURN VALUE

This function always returns the error [DFS_ESRCH].

ERRORS

This function fails if:

[DFS_ESRCH] Operation not supported in this DFS version.

NAME

AFS_HardLink — Create a hard link

SYNOPSIS

```
error_status_t  AFS_HardLink(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsTaggedName *Namep,
    /* IN  */ afsFid      *ExistingFidp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFetchStatus *OutFidStatusp,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which the file is currently located.
Namep	The name to use for the new hard link.
ExistingFidp	The file identifier specifying the file identifier of the file to which the hard link should be made.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidStatusp	The status information of the newly created hard link upon termination of the call.
OutDirStatusp	The status information of the directory (the one in which the hard link was created) upon termination of the call.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call creates a hard link to the file specified by *existingFid* to the new name *nameP* in the specified directory. The file specified by *existingFid* must not be a directory and must be on the same **DCE DFS** file system fileset.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT]	The entry named by *DirFidp* does not exist.
[DFS_EACCES]	The caller lacks the required access rights.
[DFS_EXDEV]	The target is in a different fileset than the source directory.

[DFS_EEXIST]	The target name *nameP* already exists.
[DFS_EINVAL]	The entry is named ‘‘.’’ or ‘‘..’’, or the caller passed in some invalid argument.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_MakeDir — Create a directory

SYNOPSIS

```
error_status_t  AFS_MakeDir(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsTaggedName *Namep,
    /* IN  */ afsStoreStatus *InStatusp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFid      *OutFidp,
    /* OUT */ afsFetchStatus *OutFidStatusp,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file identifier specifying the directory in which the new directory is to be created.
Namep	The name of the directory to create.
InStatusp	This specifies the new status information, including clientModTime field and unixModeBits, of the new directory point.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutFidp	The file identifier for the newly created directory.
OutFidStatusp	The status information of the newly created directory upon termination of the call.
OutDirStatusp	The status information of the directory (the one in which the directory was created) upon termination of the call.
OutTokenp	A new token granted against the newly created directory.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call creates a new directory; a token is granted against the newly created directory.

It is the cache manager's responsibility to either create an entry locally in the directory specified by *DirFidp*, or to invalidate this directory's cache entry.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT]	The entry named by *DirFidp* does not exist.
[DFS_EACCES]	The caller lacks the required access rights.
[DFS_EEXIST]	The target name *nameP* already exists.
[DFS_EINVAL]	The entry is named ‘‘.’’ or ‘‘..’’, or the caller passed in some invalid argument.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_RemoveDir — Remove a directory

SYNOPSIS

```
error_status_t  AFS_RemoveDir(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsFidTaggedName *Namep,
    /* IN  */ afsHyper    *returnTokenIDp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsFid      *OutFidp,
    /* OUT */ afsFetchStatus *OutDelStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h The RPC binding handle.

DirFidp The file identifier specifying the directory in which the directory to be deleted is located.

Namep The complex name of the directory to delete.

returnTokenIDp A token ID being returned, if any.

minVVp The minimum-acceptable version number on the fileset containing this file or directory.

Flags Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

OutDirStatusp The status information of the directory (the one in which the directory was deleted) upon termination of the call.

OutFidp

OutDelStatusp

Syncp The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This call removes a directory from the file system.

The directory must be empty (containing only entries for '.' and '..'), otherwise this call will fail. It is the cache manager's responsibility to either create an entry locally in the directory specified by *DirFid*, or to invalidate this directory's cache entry.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT] The entry named by *DirFidp* does not exist, or an entry named *nameP* does not exist in this directory.

[DFS_EACCES] The caller lacks the required access rights.

[DFS_EINVAL] The entry is named ‘‘.’’ or ‘‘..’’, or the caller passed in some invalid argument.

[DFS_ENOTEMPTY]
The directory is not empty.

[DFS_EBUSY] On some server systems, a process has this directory as its working directory.

[FSHS_ERR*] File server does not recognize client, or is in some form of recovery.

[TKM_ERROR*] A token managment failure occurred.

NAME

AFS_Readdir — Read an entry from a directory

SYNOPSIS

```
error_status_t  AFS_Readdir(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *DirFidp,
    /* IN  */ afsHyper    *Offsetp,
    /* IN  */ unsigned32  Size,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsHyper    *NextOffsetp,
    /* OUT */ afsFetchStatus *OutDirStatusp,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsVolSync  *Syncp,
    /* OUT */ pipe_t      *dirStream
    );
```

ARGUMENTS

h	The RPC binding handle.
DirFidp	The file descriptor for the specified directory.
Offsetp	The offset into the directory for this entry.
Size	Number of bytes to read.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
NextOffsetp	The offset into the directory for the following entry.
OutDirStatusp	Status information for the directory pointed to by *DirFidp*.
OutTokenp	Token granted against *DirFidp*.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.
dirStream	The array of bytes making up the external representation of this part of the directory.

DESCRIPTION

This call reads the specified directory entry, returning the requested entry and information to obtain the following entry.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	The caller passed in some invalid argument.
[DFS_ENOENT]	The entry named by *Fidp* does not exist.

[DFS_EACCES] The caller lacks the required access rights.

[DFS_ENOTDIR] The entry is not a directory.

[FSHS_ERR*] File server does not recognize client, or is in some form of recovery.

[TKM_ERROR*] A token managment failure occurred.

NAME

AFS_GetToken — Obtain a token

SYNOPSIS

```
error_status_t  AFS_GetToken(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *Fidp,
    /* IN  */ afsToken    *MinTokenp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /* OUT */ afsToken    *OutTokenp,
    /* OUT */ afsRecordLock *OutBlockerp,
    /* OUT */ afsFetchStatus *OutStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	File identifier of the file to obtain a token against.
MinTokenp	Specification of the minimum requested token.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
OutTokenp	Actual token granted.
OutBlockerp	Information about the possessor of the token that prevents the granting of the requested token (valid only for lock-family tokens).
OutStatusp	Status information on the file specified by *Fidp*.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This routine is called to obtain a token from the file exporter's token management layer.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT]	The entry named by *Fidp* does not exist.
[DFS_EACCES]	The caller lacks the required access rights.
[TKM_*]	The token manager failed to grant the token.
[DFS_EINVAL]	The file specified by *Fidp* does not exist, or the caller passed in some invalid argument.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.

[TKM_ERROR*] A token managment failure occurred.

NAME

AFS_GetStatistics — Obtain file server statistics

SYNOPSIS

```
error_status_t  AFS_GetStatistics(
    /* IN  */ handle_t    h,
    /* OUT */ afsStatistics *Statisticsp
    );
```

ARGUMENTS

h	The RPC binding handle.
Statisticsp	File server statistics.

DESCRIPTION

This call returns statistics concerning file server throughput, resource usage and disk storage at the file server to which the call is directed. It is used for status monitoring purposes only. It always returns the value 0 indicating that the call was successful.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

NAME

AFS_ReleaseTokens — Release tokens granted by file server

SYNOPSIS

```
error_status_t  AFS_ReleaseTokens(
    /* IN  */ handle_t    h,
    /* IN  */ afsReturns  *Tokens_Arrayp,
    /* IN  */ unsigned32  Flags
    );
```

ARGUMENTS

h	The RPC binding handle.
Tokens_Arrayp	Tokens granted to this file server which are to be relinquished.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.

DESCRIPTION

This call releases an array of tokens to a given file server. The tokens must have been granted originally by the file server to which this call is directed. The cache manager should not attempt to give up more than {MAX_TOKEN_RELEASE} tokens in any one call to a file server.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL]	Caller has not registered a token revocation service.
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.
[TKM_ERROR*]	A token managment failure occurred.

NAME

AFS_GetTime — Returns the time of day

SYNOPSIS

```
error_status_t  AFS_GetTime(
    /* IN  */ handle_t    h,
    /* OUT */ unsigned32  *Secondsp,
    /* OUT */ unsigned32  *USecondsp,
    /* OUT */ unsigned32  *SyncDistance,
    /* OUT */ unsigned32  *SyncDispersion
    );
```

ARGUMENTS

h	The RPC binding handle.
Secondsp	Number of seconds since January 1, 1970 UTC.
USecondsp	Number of microseconds into the current segment.
SyncDistance	Estimated path length to source of reliable time.
SyncDispersion	Measure of SyncDistance's variance.

DESCRIPTION

This call returns the time of day in seconds and microseconds in the same format as returned by the gettimeofday*system*( )call

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.

NAME

AFS_BulkFetchVV — Obtain many volume version numbers

SYNOPSIS

```
error_status_t  AFS_BulkFetchVV(
    /* IN  */ handle_t    h,
    /* IN  */ afsHyper    *cellIdp,
    /* IN  */ afsBulkVolIDs *VolIDsp,
    /* IN  */ unsigned32  NumVols,
    /* IN  */ unsigned32  Flags,
    /* IN  */ unsigned32  spare1,
    /* IN  */ unsigned32  spare2,
    /* OUT */ afsBulkVVs  *VolVVsp,
    /* OUT */ unsigned32  *spare4
    );
```

ARGUMENTS

h	The RPC binding handle.
cellIdp	The cell identifier for the filesets whose Volume Version numbers are desired.
VolIDsp	The fileset identifiers (within cellIdp) whose Volume Version numbers are desired.
NumVols	(Redundant) number of identifiers in VolIDsp.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
spare1, spare2	Spare IN parameters.
VolVVsp	The Volume Version numbers for the filesets whose identifiers were passed in.
spare4	Spare OUT parameter.

DESCRIPTION

This call is used to retrieve *en masse* the Volume Version number of a collection of filesets on the same cell. This allows for optimization for maintaining groups of filesets, replicated or otherwise, as might be done by a cache manager or a replication server.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT]	The fileset specified does not exist.
[DFS_EACCES]	The caller lacks the required access rights.
[DFS_EINVAL]	The value for *NumVols* is too small .
[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.

NAME

AFS_ProcessQuota — Get or set quota information

SYNOPSIS

```
error_status_t  AFS_ProcessQuota(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *Fidp,
    /* IN  */ afsHyper    *minVVp,
    /* IN  */ unsigned32  Flags,
    /*INOUT*/ afsQuota    *quotaListp,
    /* OUT */ afsFetchStatus *OutStatusp,
    /* OUT */ afsVolSync  *Syncp
    );
```

ARGUMENTS

h	The RPC binding handle.
Fidp	File ID of file or directory whose quota is being changed.
minVVp	The minimum-acceptable version number on the fileset containing this file or directory.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
quotaLisp	The quota information being provided or obtained.
OutStatusp	The updated file status after the operation.
Syncp	The current synchronization information about the fileset containing this file; this allows for detection of changes in the fileset containing the specified file or directory.

DESCRIPTION

This function is obsolete. It does not perform any useful function. It was used to get or set quota information; the same call was used for both operations. The quota information is interpreted in a file system-specific manner. For example, the quota set with a UFS file system is a Berkeley quota, having per user components, while with a **DCE LFS** file system, the relevant quota is a **DCE LFS** per-fileset quota field.

RETURN VALUE

This function always fails and returns the error [DFS_EINVAL]. A value of –1 is returned and *errno* is set to indicate the error.

ERRORS

This function fails if:

[DFS_EINVAL] The quota structure is illegal. This function always returns this error.

NAME

AFS_BulkKeepAlive — Keep multiple tokens active

SYNOPSIS

```
error_status_t  AFS_BulkKeepAlive(
    /* IN  */ handle_t    h,
    /* IN  */ afsBulkFEX  *KAFEXp,
    /* IN  */ unsigned32  numExecFids,
    /* IN  */ unsigned32  Flags,
    /* IN  */ unsigned32  spare1,
    /* IN  */ unsigned32  spare2,
    /* OUT */ unsigned32  *spare4
    );
```

ARGUMENTS

h	The RPC binding handle.
KAFEXp	Collection of afsFid structures.
numExecFids	(Redundant) the number of afsFid structures in *KAFEXp*.
Flags	Any of the AFS_FLAG_* flags defined in the **<afs4int.h>** file.
spare1, spare2	Spare IN parameters.
spare4	Spare OUT parameter.

DESCRIPTION

This call is used to cause the file exporter to maintain open-for-preserve tokens on the indicated files until some characteristic interval passes. This call is expected to be useful in preventing storage for the lazy replicas of those files from being deleted before all cache managers are finished with it. The characteristic interval is a property of each read-only fileset.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[DFS_ENOENT] An entry in the *KAFEXp* structure represents a non-existent file.

NAME

AFS_SetParams — Set cache manager parameters

SYNOPSIS

```
error_status_t  AFS_SetParams(
    /* IN  */ handle_t    h,
    /* IN  */ unsigned32  Flags,
    /*INOUT*/ struct afsConnParams *paramsP
    );
```

ARGUMENTS

h	The RPC binding handle.
Flags	Either AFS_PARAM_TSR_COMPLETE or AFS_PARAM_RESET_CONN.
paramsP	A block of connection-oriented parameters.

DESCRIPTION

This call allows the Cache Manager to request that the File Server alter its connection parameters, in particular the lifetimes and timeouts associated with token state maintenance. Parameter suggestions are supplied by the Cache Manager as the input values in **paramsP*, and the values selected by the File Server in response are returned to the Cache Manager as the output values returned via **paramsP*.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a non-zero value will indicate the error.

ERRORS

This function fails if:

[FSHS_ERR*]	File server does not recognize client, or is in some form of recovery.

NAME

AFS_GetServerInterfaces — Obtain file server's supported interfaces

SYNOPSIS

```
error_status_t  AFS_GetServerInterfaces(
    /* IN  */ handle_t    h,
    /*INOUT*/ dfs_interfaceList *serverInterfacesP
    );
```

ARGUMENTS

h	The RPC binding handle.
serverInterfacesP	Will contain the interface information when the call returns. Currently there is only one interface defined.

DESCRIPTION

This call returns information about the RPC interfaces that the file server is exporting.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

Chapter 7

Cache Manager Service Interface

The cache manager interface describes the interface between the client's cache manager and the server's file exporter (where the *server* initiates the operation). This section details the services provided by the cache manager; those provided by the file exporter (file server) are described in the topic, *DCE DFS File Exporter Interface*, found in Chapter 6 on page 99.

All of these calls are expected to return success ([error_status_ok]) unless noted otherwise.

NAME

TKN_InitTokenState — Initialize token state

SYNOPSIS

```
error_status_t  TKN_InitTokenState(
    /* IN  */ handle_t    h,
    /* IN  */ unsigned32  Flags,
    /* IN  */ unsigned32  hostLifeGuarantee,
    /* IN  */ unsigned32  hostRPCGuarantee,
    /* IN  */ unsigned32  deadServerTimeout,
    /* IN  */ unsigned32  serverRestartEpoch,
    /* IN  */ unsigned32  spare1,
    /* IN  */ unsigned32  spare2,
    /* IN  */ unsigned32  spare3,
    /* OUT */ unsigned32  *spare4,
    /* OUT */ unsigned32  *spare5,
    /* OUT */ unsigned32  *spare6
    );
```

ARGUMENTS

h The RPC binding handle.

Flags Flags (TKN_FLAG*) indicating the server's likely token state.

hostLifeGuarantee
Duration of the host lifetime on the server.

hostRPCGuarantee
Duration of the promise of RPC attempts from the server.

deadServerTimeout
Suggested maximum interval between probes from Cache Managers to unresponsive servers.

serverRestartEpoch
Numeric tag for this run of the File Server, to allow Cache Managers to distinguish network partitions from server crashes.

spare1 .. spare6 Spares.

DESCRIPTION

Initialize token state at the workstation, with respect to the calling server. All tokens from the calling server should be discarded or renegotiated.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

NAME

TKN_Probe — Probe state of cache manager

SYNOPSIS

```
error_status_t  TKN_Probe(
    /* IN  */ handle_t    h
    );
```

ARGUMENTS

h The RPC binding handle.

DESCRIPTION

This call is a noop used by the server to check that a cache manager is still running. By periodically polling apparently inactive cache managers, a file server can timeout down workstations before actually trying to revoke a token at that workstation, allowing the operation eventually triggering a token revocation operation to complete faster.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

NAME

TKN_TokenRevoke — Request revocation of a previously granted token

SYNOPSIS

```
error_status_t  TKN_TokenRevoke(
    /* IN  */ handle_t    h,
    /*INOUT*/ afsRevokes  *revokeDescp
    );
```

ARGUMENTS

h The RPC binding handle.

revokeDescp The tokens which are being revoked.

DESCRIPTION

This call is used by the file exporter requesting that a set of previously granted tokens be relinquished. If possible, the client's cache manager is to release these tokens.

RETURN VALUE

If this function succeeds, it returns a value of zero. Otherwise, a value of zero is returned and the revoke descriptor is set to indicate exactly which tokens were not revoked. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success. Even when a token is not returned, this call returns success but indicates in the revoke descriptor exactly which tokens were not revoked.

NAME

TKN_GetCellName — Request client cell name

SYNOPSIS

```
error_status_t  TKN_GetCellName(
    /* IN  */ handle_t    h,
    /* IN  */ afsHyper    *Cellp,
    /* OUT */ NameString_t CellNamep
    );
```

ARGUMENTS

h	The RPC binding handle.
Cellp	The ID for a cell.
CellNamep	The string name corresponding to **cellP**.

DESCRIPTION

This routine is used to perform cell id to cell name translation.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

NAME

TKN_SetParams — Request client change parameters

SYNOPSIS

```
error_status_t  TKN_SetParams(
    /* IN  */ handle_t    h,
    /* IN  */ unsigned32  Flags,
    /*INOUT*/ afsConnParams *paramsP
    );
```

ARGUMENTS

h	The RPC binding handle.
Flags	For future expansion.
paramsP	A block of connection-oriented parameters.

DESCRIPTION

This call allows the File Server to request that a cache manager alter its connection parameters, in particular the lifetimes and timeouts associated with token state maintenance. Parameter suggestions are supplied by the file server as the input values in *paramsP,and manager in response are returned to the file server as the output values returned via ***paramsP*.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

NAME

TKN_GetServerInterfaces — Determine interfaces supported by client

SYNOPSIS

```
error_status_t  TKN_GetServerInterfaces(
    /* IN  */ handle_t    h,
    /*INOUT*/ dfs_interfaceList *serverInterfacesP
    );
```

ARGUMENTS

h The RPC binding handle.

serverInterfacesP List of supported interfaces.

DESCRIPTION

This call could be used by the file server to determine what interfaces a cache manager client supports. This would permit compatibility between older/newer versions of file exporters and cache managers.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

NAME

TKN_AsyncGrant — Notify client of token availability

SYNOPSIS

```
error_status_t  TKN_AsyncGrant(
    /* IN  */ handle_t    h,
    /* IN  */ afsFid      *grantedFileIDp,
    /* IN  */ afsToken    *grantedTokenP,
    /* IN  */ long grantedRequestID
    );
```

ARGUMENTS

h	The RPC binding handle.
grantedFileIDp	File identifier of the file or directory to which the granted token applies.
grantedTokenP	The token that was granted.
grantedRequestID	Identifier of the original async grant request.

DESCRIPTION

This call allows the file server to notify a cache manager client than an async grant request for a token has succeeded.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success.

ERRORS

None.

This function succeeds if:

[error_status_ok] This function always returns success.

X/Open Preliminary Specification

Part 5:

Access Control Lists (ACLs) in DFS

This part of the **DCE DFS** provides an overview of Access Control Lists (ACLs) in DFS, their formats and the interface provided to manipulate them.

Chapter 8

Access Control List Overview

This chapter describes the use of DCE Access Control Lists (ACLs) with DFS. An **ACL** lists the authorized users of an object and their allowed accesses. In DFS, **ACL**s control access to filesystem objects (files and directories) and server processes. These filesystem objects are stored in the DCE Local File System (LFS) as filesets. A fileset as used in this document is a hierarchical grouping of files that is mangaed as a single unit or entity. These filesets can vary in size but are usually smaller than a standard UNIX disk partition. In DFS, multiple DCE LFS filesets can be stored on a single UNIX disk partition. A non-LFS partition (for example, a UNIX Partition) in DFS can support (contain) only one fileset.

The **ACL**s used in **DCE LFS** filesets and discussed in this document in *Part 5, Access Control Lists (ACLs) in DFS*, differ from those used in other DCE Components. Refer to the *Security Service* portion of the **OSF DCE Administration Guide — Core Components** for details concerning DCE **ACL**s if further information is desired. This chapter also discusses *groups*. Further information about them can also be found in the *Security Service* portion of the **OSF DCE Administration Guide — Core Components** as well. Information on DFS usage of *groups* can also be found in the **OSF DCE Administration Guide — Core Components** for **OSF DCE 1.1**.

Associated with most DFS server processes is an administrative list that defines the users and server machines (called principals) and also the groups that can perform operations that affect the (server) process. Furthermore, server processes on different machines can have different lists (or each process may use a copy of the same list). Additionally, processes can share the same administrative list even though they may be of different types. Refer to Chapter 4 of the **OSF DCE DFS Administration Guide and Reference** for **OSF DCE 1.1** for information on using administrative lists in DFS.

8.1 ACL Types

There are three types of **ACL**s. They are:

Object ACL
: Controls access to the object itself. Both filesystem objects and administrative lists have this **ACL**.

Initial Object Creation ACL
: Possessed only by a directory. It is the default **ACL** inherited by a file created in the directory.

Initial Container Creation ACL
: Possessed only by a directory. It is the default **ACL** inherited by a sub-directory created in the directory.

The existence of any of these three types of **ACL**s is mutually exclusive of any other. They can exist independently of one another or may not exist at all. Or, a given directory can have all, some, or none of these **ACL**s. The **OSF DCE DFS Administration Guide and Reference** (for **OSF DCE 1.1**) discusses **ACL**s and their inheritance in detail and explains the correspondence between their existence or absence and their contents.

8.2 ACL Entry Types

An **ACL** consists of a number of entries, each of which typically specify the access allowed to a user, group or organization. Typically, an **ACL** defines the operations that a user or group (or organization) can perform on an object (file or directory) in the form of permissions granted to various users or groups. However, these permissions can also apply to an entity called a principal, such as a server, that might be invoked on behalf of a user in order to complete a function requested by the user.

There is the concept of a default cell that is discussed in this section. A default cell of an **ACL** is the cell with respect to which the entries for the **ACL** are defined. A *user* or *group* named as an entry type in an **ACL** is defined to be from the default cell unless the entry type explicitly names a different cell.

In this document, a local user is one whose local cell is the same as the default cell of an **ACL**. Similarly, a foreign user is one whose default cell is different from the default cell of an **ACL**. Also, a foreign cell is different from the default cell of the **ACL**. A user's local cell is the cell in whose registry database the user's principal and account are defined.

The various types of entries are as follows:

user_obj Specifies access rights for the owner principal of the object.

group_obj Specifies access rights for the member principals of the group that owns the object.

user Specifies access rights for a specific principal in the default cell of the **ACL**.

group Specifies access rights for the member principals of a specific group in the default cell of the **ACL**.

other_obj Specifies access rights for all other principals in the default cell of the **ACL**.

foreign_user Specifies access rights for a specific principal in a specific foreign cell.

foreign_group Specifies access rights for the member principals of a specific group in a specific foreign cell.

foreign_other Specifies access rights for all principals in a specific foreign cell who are not covered by the *foreign_user* and *foreign_group* entries.

any_other Specifies access rights for all other principals in all other foreign cells and unauthenticated users.

extended A provision for future types.

8.2.1 Rules Governing Entries for Filesystem ACLs

- A filesystem **ACL** must have the *user_obj*, *group_obj* and *other_obj* entries. All others are optional.
- An **ACL** can only have 1 entry each of types *user_obj*, *group_obj*, *other_obj* and *any_other*.
- If an entry other than *user_obj*, *group_obj* or *other_obj* exists on an **ACL**, then the optional entry, *mask_obj*, defined in Section 8.2.2 on page 157 must also exist.
- An **ACL** that has an *authenticated* entry, discussed in Section 8.2.3 on page 157, cannot be modified (until and unless this entry is removed).
- The *user_obj* entry must always explicitly retain the **c** permission in order that the owner not be denied access to an owned object.

8.2.2 Optional ACL Entry Mask Types for Authenticated Users

The following optional entries are masks that serve to limit the maximum access allowed. They apply to authenticated users. An authenticated uses is one whose DCE identity has been verified by the DCE Security Service and whose credentials have not expired.

mask_obj Specifies maximum access allowed to all principals except principals covered by *user_obj* and *other_obj*. Only the access found in both the entry and the *mask_obj* are granted. In absence of this mask, the default action is to place no limitations at all on types of access allowed.

This optional entry applies only to **DCE LFS** objects. It does not apply to **ACL**s on objects associated with other DCE components.

Authenticated Access Determination

In order to determine access permitted for an object for a user that is authenticated, the **ACL** is evaluated according to the steps in Section 8.7 on page 161.

8.2.3 ACL Entry Types for Unauthenticated Users

An unauthenticated user is one that has logged in to a machine but has not logged into DCE. Thus, the user's DCE identity has not been verified by the DCE Security Service. Once authenticated,a user is provided a set of credentials by the DCE Security Service. If the DCE credentials of such a user have expired, the user is also considered to be an unauthenticated user.

DCE **ACL**s used with objects for other DCE components include an additional *unauthenticated* entry type that masks the access that is permitted for unauthenticated users. There are no entry types on an **ACL** for unauthenticated users for **DCE LFS** objects.

Unauthenticated Access Determination

For users that are unauthenticated, access permissions are determined thusly rather than by Section 8.7 on page 161:

A. For access to an object in a **DCE LFS** fileset:

- The user is assigned the identity **nobody** which is treated as an authenticated user from an unknown foreign cell[1]. The primary group of the identity **nobody** is the group **nogroup**.
- The user permissions are associated with the *any_other* entry[2].

Note: Prior to DCE 1.1, it was possible for *unauthenticated* entries to exist on **ACL**s of **DCE LFS** objects; however, they were ignored when determining the access persissions allowed for the user on the objects. If an existing object contains such an entry, it should be removed[3]. For DCE 1.1 and newer versions, *unauthenticated* entries for an **ACL** continue to be ignored when determining the access permissions.

---

1. Since the cell is unknown, its default cell is also unknown. Thus, no entries pertaining to the unknown (foreign) cell can be created on any **ACL**.
2. If an *any_other* entry is not present on the **ACL**, the user has no permissions. If unauthenticated users are to be prevented from acquiring permissions for an object, this can be accomplished by not having an *any_other* entry on the object's **ACL**.

B. For access to an object in a **non-LFS** fileset:

- The user is assigned the identity **nobody** which is treated as an authenticated user regardless of their cell[4].
- The user permissions are associated with the *other* UNIX mode bits[5].

8.2.4 Simple and Complex Entry Types

Simple **ACL** entries refer to entries that do not need to store a principal id in them. These are the *user_obj*, *group_obj*, *other_obj*, *any_other* and *mask_obj* entries. *Complex* entries refer to all other entry types and store a principal id in the entry.

8.3 Delegates and DCE LFS Objects

A user initially requesting some operation on an **DCE LFS** object is referred to as the initiator of the operation. Typically the request is made for direct access to such an object. There are instances when a server may perform the work involved in a request on behalf of the initiator, such as a request to print a file. In such instances, the server principal that has been delegated the operation on the initiator's behalf is referred to as the delegate. The delegate performs operations on an object on behalf of the initiator. The operation performed in this manner is referred to as a delegation operation.

For an operation that does not involve delegation, only the initiator needs to have the necessary access permissions to perform a requested operation. However, for a delegation operation, both the initiator and the delegate must have the access permissions necessary to perform the operation.

DCE ACLS permit the granting of access permissions to a principal that apply only when the principal is a delegate on behalf of the operation. The next section describes the ACL entry types that are provided for delegation operations.

---

3. Moving or restoring a **DCE LFS** fileset to a fileserver that is running DCE 1.1 or a newer version will automatically result in having any *unauthenticated* **ACL** entries removed from all objects in the fileset.
4. This includes all *foreign* users (authenticated and unauthenticated alike).
5. For clarity, as previously stated, authenticated users from foreign cells are granted the permissions associated with their authenticated *foreign* identities when they access objects in **DCE LFS** filesets.

8.4 ACL Entry Types for Delegation

There are entries for delegation corresponding to most of the entries specified in that can be used for delegation with **DCE LFS** objects. Since **DCE LFS** does not examine the entry types associated with the following:

- *user_obj*
- *group_obj*
- *other_obj*.

for delegation, entries for these types shown in Section 8.2 on page 156 are omitted from this list of entry types for delegation. The valid entry types for delegation are the following:

user_delegate Specifies access rights for a specific principal from the default cell of the **ACL** acting as a delegate.

group_delegate Specifies access rights for the members of a specific group from the default cell of the **ACL** acting as a delegate.

foreign_user_delegate Specifies access rights for a specific principal from a specific foreign cell acting as a delegate.

foreign_group_delegate Specifies access rights for the member principals of a specific group in a specific foreign cell.

foreign_other_delegate Specifies access rights for all principals in a specific foreign cell who do not match any of the preceeding delegation entries acting as deletgates.

any_other_delegate Specifies access rights for all other principals in all other foreign cells who do not match any of the preceeding delegation entries acting as deletgates.

8.4.1 ACL Access Types for Delegation

Each of the delegation **ACL** entry types in the previous section can grant any of the access permissions available for **DCE LFS** objects. Further, each permission has the same meaning for a delegation entry that it has for a non-delegation entry. Refer to Section 8.5 for the list of the permissions and their meaning.

8.5 Access Types

A principal may access an object in a number of ways which define the operations the principal may perform on that object. These types of access are frequently termed permissions. If the object is a directory, certain permissions which apply only to it further define the types of access permitted on the directory object. These are the *insert* and *delete* permissions shown below.

The types of access that a principal can be granted to an object are defined to be the following:

read Rights grant the user the ability to read and/or stat the contents of the object in question. It gives the possessor the ability to read the names and file IDs stored within a directory, as well as the ability to stat the directory. The read right for a file gives the possessor the ability to **stat** the file as well as read the data stored within the file.

write Rights allow the principal to update the contents of a file. This right, for directories, is split into two other rights, *insert* and *delete* rights.

insert Rights allow the principal to add entries to a directory. This includes the right to perform **rename**s where the target is located within the directory. Note that if the target already exists, one will also require *delete* rights to remove the target of the **rename** system call.

delete Rights allow the principal to remove entries from a directory. This includes the right to **rename** an object from one directory, as well as the right to obliterate an entry by renaming another entry ''on top of it''.

execute/search Rights allow the principal to execute a program (for files), or examine individual directory entries (for directories). Examining individual entries differs from enumerating the contents of a directory in that the former allows the retrieval of a piece of information only if the entire correct string name of the relevant file is presented to the file system.

control Rights allow the principal to perform those operations normally reserved to a file or directory's owner. This includes executing the **chmod** and **utimes** system calls, as well as modifying the **ACL** on the object.

8.6 Interaction of Filesystem ACLs with UNIX Permission Bits

In the UNIX operating system, file system protection for files and directories (henceforth called file and directory objects in this document) is provided by a set of bits called mode bits. The bits govern the operations a user can perform, and are called *read*, *write*, and *execute* mode bits. These bits determine access permissions on file and directory objects for three classifications of users:

- the user who owns the object
- members of the group that owns the object
- all other users.

File and directory objects in DCE filesystems, frequently referred to as **DCE LFS** filesets, also have mode bits. In addition, the protection of such files and directories can be augmented with DCE **ACL**s.

Note: DCE **ACL**s are used only with objects in **DCE LFS** filesets. For most non-LFS filesets, UNIX mode bits are the only form of protection.

The interaction of filesystem **ACL**s with UNIX permission mode bits depends in part upon the characteristics of the parent directory, for both directories and files within the filesystem. It also depends upon whether the file or directory is created by a foreign user. The governance is as follows:

- If an object's (file or directory) parent directory has an appropriate *Initial Creation ACL* (object or container), the object has inherited an *Object ACL* as its form of protection. The object has UNIX mode bits but these are also augmented by the inherited *Object ACL*.
- If an object's parent directory does *not* have an appropriate *Initial Creation ACL*, the object relies entirely upon mode bits as its only form of protection.

  There is an exception to this rule in that if a file or directory is created by a foreign user, an *Object ACL* is always created for protection even if the parent directory does not have the appropriate *Initial Creation ACL*.

In DFS, an object's UNIX mode bits and its **ACL** permissions are synchronized. The UNIX *owner* permission bits always correspond to *user_obj* **ACL** entry permissions. The UNIX *other* permission bits always correspond to *other_obj* **ACL** entry permissions. The correspondence for

UNIX *group* permission bits depends upon the presence or absence of the *mask_obj* **ACL** entry. If the **ACL** contains a *mask_obj* entry, the UNIX group permission bits correspond to the *mask_obj* entry permissions, else the UNIX group permission bits correspond to *group_obj* **ACL** entry permissions.

Modifications to the simple **ACL** entries *user_obj*, *group_obj* and *other_obj* result in modification of the corresponding UNIX permission bits and vice versa. Appendix A on page 221, *Mapping DFS ACLs to UNIX mode bits*, discusses in more detail the relationship between **ACL**s and UNIX mode bits and provides a number of examples demonstrating the synchronization provided by DFS as changes are made.

8.7 Access Check Algorithm

To determine the access allowed to a principal on an object, the entry in the object's **ACL** matching the principal is located. This determination is made by examining the object's **ACL** entries using the following sequence of steps. Each step is only executed if the previous step did not result in a match.

1. The principal id is compared with the id specified in the *user_obj*, *user* and *foreign_user* **ACL** entries in sequence. The first matching entry is used. If the matching entry is not *user_obj*, the *mask_obj* permissions mask is applied to the permissions allowed by the matching entry. Otherwise (the principal owns the object), the access is *not* filtered through the *mask_obj* entry.
2. The principal id is matched against the *group_obj* entry. If a match is found the *mask_obj* entry is used, as specified earlier, to filter the access allowed.
3. The principal id is matched against all the *group* and *foreign_group* entries. The sequence in which these entries are matched is not relevant. The set-union of permissions allowed by all matched entries, is filtered by both the *mask_obj* entry (if it exists) to determine the access permitted.
4. If the principal is a member of the default cell, the permissions specified in the *other_obj* entry determine the access allowed.

   Note: The premissions are *not* filtered through the *mask_obj* entry.
5. If the principal is from a foreign cell, the principal is matched with the *foreign_other* entries. If a match is found, the permissions allowed in the matching entry are filtered by the *mask_obj* entry to determine the access allowed.
6. If the principal is from a foreign cell and if that cell does not have a *foreign_other* entry, the permissions allowed by the *any_other* entry are filtered through the *mask_obj* to determine the access allowed.
7. If a match is not found in any of the previous steps, the principal matches no entry and is denied access to the object.

8.8 Access Check Algorithm for Delegation

A principal that is initiating an operation is granted access permission from the **ACL** entry types described in Section 8.2 on page 156. A principal acquires permissions from a delegation entry only when acting as a delegate. Additionally, for delegation, **DCE LFS** filters all access permissions to be granted via delegation entries through the *mask_obj* entry of the **ACL**. The delegation entry types are described in Section 8.4 on page 159.

Note: All delegation entry types in an **ACL** are optional. No delegation is permitted for an **ACL** with no delegation entry types.

In the **DCE LFS**, an operation requested by a delegate is performed only if the initiator and the delegate have the access permissions necessary to perform the operation. An **DCE LFS** object may have more than one delegate that has been specified. In this instance, if the initiator or one of the delegates does not have the required access, **DCE LFS** will refuse to perform the request.

The following sequence is used in determining the access allowed to a principal that is acting as a delegate. The sequence involves both delegation and non-delegation **ACL** entry types. Each step is only executed if the previous step did not result in a match.

1. The delegate owns the object. The delegate is granted the access permissionf from the *user_obj* entry.
2. If one of the following entries exists, listed in checking priority order, the delegate is granted access permissions from the first of these entries that the delegate matches:
   - *user*
   - *user_delegate*
   - *foreign_user*
   - *foreign_user_delegate*.
3. One or the other of the following is true for the delegate:
   A. The delegate is a member of the group that owns the object. In this instance, the permissions are granted via the *group_obj* entry.
   B. The delegate is a member of a group for which one of the following entries listed in checking priority order, exist. In this instance, the permissions from *all* the entries that the delegate matches are granted to the delegate. The entries are:
      - *group*
      - *group_delegate*
      - *foreign_group*
      - *foreign_group_delegate*.
4. The delegate is from the default cell. In this instance, the delegate is granted the access permissions from the *other_obj* entry.
5. The delegate is from a foreign cell for which either of the following entries listed in checking priority order, exist. In this instance, the delegate is granted the access permissions associated with the first of these entries that the delegate matches. They are:
   - *foreign_other*
   - *foreign_other_delegate*.

6. The delegate is from a foreign cell and either of the following entries listed in checking priority order, exist. In this instance, the delegate is granted the access permissions associated with the first of these entries that the delegate matches. They are:
   - *any_other*
   - *any_other_delegate*.
7. The delegate matches no entry. In this instance, the delegate is denied access to the **DCE LFS** object.

8.8.1 **Delegation and non-DCE LFS** Objects

Since non-LSF objects do not have **DCE ACL**s, the access permissions required for delegation involving a non-LFS object are based solely upon the identity and access permissions of the last delegate in the chain. This is to say, the last delegate in the chain acquires the access permissions on a non-LFS object via the *user*, *group* or *other* (UNIX) mode bits.

Similarly, for administrative lists, delegation is not considered when determining administrative priviledges. Thus, the last delegate in the chain must be included in the appropriate administrative list in order to be able to perform a priviledged operation.

Also, since delegation is first available with DCE 1.1, the following restrictions apply. It is not possible to do the following for a fileset having **ACL**s containing one or more objects having delegation entries.

- Move the fileset to:
- Restore the fileset to:
- Add a replication site for the fileset on:

a File Server that uses a version of DCE earlier than DCE1.1.

Refer to the **OSF DCE DFS Administration Guide and Reference** for **OSF DCE 1.1**, for information relative to restrictions for delegation. Chapter 3 of that document provides information of this nature. Chapter 4 of that document provides information about administration considerations.

Chapter 9

ACL Storage Format

This chapter describes the structured formats used for **ACL**s in memory. **ACL**s are stored in different formats in memory (or, in core) than on disk. In-core **ACL**s use the structured format described in this chapter, but disk **ACL**s are stored in an unstructured byte-stream format. In-core **ACL**s are packed into a byte-stream before being stored on disk and similarly when **ACL**s are read from disk, they are unpacked into the in-core format.

This in-core format can seemingly differ depending upon whether the **ACL** is being manipulated in the kernel or in user space. The next section, Section 9.1, shows how the differences are accounted for. The in-user-space *ACL*s are referred to as administrative list **ACL**s. Chapter 8 on page 155, *Access Control List Overview*, provides additional information about administrative lists and their usage in **DCE DFS**.

ACLs specify access permissions for the users and groups that can perform the specified types of operations (read, or write, for instance) on files and directories. Administrative lists specify the users and groups that can perform specified actions affecting specified server processes. For a DFS server process, groups can be specified on the administrative list associated with it. Then if certain users need access to the process, they are included in the group. Similarly, if a group of users needs access to more than one server process, they can be specified in the Registry Database as a group. Thus, they can be added or removed from the appropriate administrative list(s) as a group. A system administration team is a good example of such a group.

9.1 Principal ID Formats

Administrative lists always store principal ids in 128 bits format on disk and in memory. In filesystem **ACL**s, principal ids represent users or groups, which have only 32 significant bits. Hence, a school of thought proposed storing filesystem **ACL**s on disk using only 32 bits instead of 128 bits to prevent wasting disk space. Though this idea has been refuted, the DFS **ACL** interface does provide such flexibility and this point is important to understand the interface. In-core filesystem **ACL**s use the full 128 bits always.

The next section, Section 9.2, shows the two formats defined as either the full 128 bits or the 32 bits format. The two formats are manipulated by the functions in Chapter 10 on page 171, *Access Control List Interface Functions.*

9.2 Principal Identifier Format

In-core **ACL**s store principal ids in the full 128 bit UUID[6] format.

```
typedef struct epi_uuid {
    u_int32        longField1;
    u_int16        shortField1;
    u_int16        shortField2;
    unsigned char  miscChars[8];
```

6. UUID is a universally unique identifier for an object.

```
} epi_uuid_t;

#if defined(EPI_USE_FULL_ID)
typedef epi_uuid_t epi_principal_id_t;
#else /* defined(EPI_USE_FULL_ID) */
typedef u_int32 epi_principal_id_t;
#endif /* defined(EPI_USE_FULL_ID) */
```

9.3 Foreign Cell Principal Identifier Format

A principal from a foreign cell is represented by type **epi_uuid_foreign_t**.

```
typedef struct epi_uuid_foreign {
    epi_uuid_t    id;
    epi_uuid_t    realm;
}   epi_uuid_foreign_t;
```

9.4 Access Type Format

The access permissions are represented with a bitset type **dacl_permset_t**. The set also includes provision for future additional types.

```
/* the permission sets stored in an acl */

typedef u_int32 dacl_permset_t;
```

9.4.1 Access Type Definitions

```
/* the various permissions that may be stored in such a set */
/* first the POSIX ones */
#define dacl_perm_read     0x01
#define dacl_perm_write    0x02
#define dacl_perm_execute  0x04
#define dacl_perm_control  0x08
#define dacl_perm_insert   0x10
#define dacl_perm_delete   0x20
#define DACL_USERLIST_PERMS (dacl_perm_read | dacl_perm_write | \
                                              dacl_perm_control)
#define DACL_USERLIST_PERMSTRING "rwc"

#define ntoh_dacl_entry_permset_t(permsetP) *entryTypeP = ntohl(*permsetP)
#define hton_dacl_entry_permset_t(permsetP) *entryTypeP = htonl(*permsetP)
```

9.5 ACL Entry Type Format

The **ACL** entry types are specified by the type **dacl_entry_type_t**.

```
typedef u_int32 dacl_entry_type_t;
#define dacl_entry_type_user_obj       0
#define dacl_entry_type_group_obj      1
#define dacl_entry_type_other_obj      2
#define dacl_entry_type_user           3
#define dacl_entry_type_group          4
#define dacl_entry_type_mask_obj       5
/*
#define dacl_entry_type_class_group    5
#define dacl_entry_type_class_owner    6
#define dacl_entry_type_class_other    7
*/
#define dacl_entry_type_foreign_user   8
#define dacl_entry_type_foreign_group  9
#define dacl_entry_type_foreign_other  10
#define dacl_entry_type_unauth         11
#define dacl_entry_type_extended       12
#define dacl_entry_type_anyother       13

#define dacl_entry_type_user_obj_delegate  14
#define dacl_entry_type_group_obj_delegate 15
#define dacl_entry_type_other_obj_delegate 16
#define dacl_entry_type_user_delegate      17
#define dacl_entry_type_group_delegate     18
#define dacl_entry_type_foreign_other_delegate  19
#define dacl_entry_type_foreign_user_delegate   20
#define dacl_entry_type_foreign_group_delegate  21
#define dacl_entry_type_any_other_delegate      22

#define ntoh_dacl_entry_type_t(entryTypeP) *entryTypeP = ntohl(*entryTypeP)
#define hton_dacl_entry_type_t(entryTypeP) *entryTypeP = htonl(*entryTypeP)
```

Note: The entry type *unauth* is no longer valid, for DCE 1.1 and newer versions of DFS. See the note in Section 8.2.3 on page 157 of the chapter, *Access Control List Overview*.

9.6 Simple ACL Entry Format

Each of the simple **ACL** entries is represented by type **dacl_simple_entry_t** defined as:

```
typedef struct dacl_simple_entry {
    u_int32        is_entry_good;
    dacl_permset_t perms; /* permissions allowed */
}   dacl_simple_entry_t;
```

The simple entries are stored in the **ACL** as an array indexed by the entry type **dacl_simple_entry_type_t**.

```
/* the following enumerated type is used only in memory */
typedef enum dacl_simple_entry_type {
    dacl_simple_entry_type_userobj,
    dacl_simple_entry_type_groupobj,
    dacl_simple_entry_type_otherobj,
    dacl_simple_entry_type_maskobj,
    dacl_simple_entry_type_anyother,
    dacl_simple_entry_type_userobj_delegate,
    dacl_simple_entry_type_groupobj_delegate,
    dacl_simple_entry_type_otherobj_delegate,
    dacl_simple_entry_type_anyother_delegate,
    dacl_simple_entry_type_unauthmask /* this must remain as the last
                                         of the enums */
} dacl_simple_entry_type_t;
```

Note: The entry, *unauthmask*, must remain as the last of the enums. See Section 9.8 on page 169 for usage information.

9.7 Complex ACL Entry Format

Each of the complex **ACL** entries is represented by type **dacl_complex_entry_t** defined as follows. This structure holds entries that have data associated with them.

```
typedef struct dacl_complex_entry {
    u_int32        num_entries;
    u_int32        entries_allocated;
    dacl_entry_t*  complex_entries;
}   dacl_complex_entry_t;

typedef struct dacl_entry {
    dacl_permset_t    perms;
    dacl_entry_type_t entry_type;
    union {
        epi_uuid_t    id; /* for locally keyed entries */
        epi_uuid_foreign_t foreign_id; /* for foreign keyed entries */
        dacl_extended_info_t extended_info; /* for uninterpreted data */
   } entry_info;
} dacl_entry_t;
```

The complex entries are stored in the **ACL** as an array indexed by the entry type **dacl_complex_entry_type_t**. The *user* and*group* entries include both *foreign* and *local* entries. The *other* entry includes both *foreign other* and *extended* entries.

```
typedef enum dacl_complex_entry_type {
    dacl_complex_entry_type_user, /* includes both user and foreign_user */
    dacl_complex_entry_type_group, /* includes both group and foreign_group */
    dacl_complex_entry_type_user_delegate,
    dacl_complex_entry_type_group_delegate,
    dacl_complex_entry_type_for_other_delegate,
    dacl_complex_entry_type_other /* includes both foreign_other
                                     & extended */
} dacl_complex_entry_type_t;
```

Note: The entry, *other*, must remain as the last of the enums. See Section 9.8 for usage information.

9.7.1 Extended Complex Entry Format

The extended complex **ACL** entry type **dacl_extended_info_t** is described as:

```
typedef struct dacl_format_label {
    u_int16    miscShorts[3];
    char       charField;
} dacl_format_label_t;

typedef struct dacl_extended_info {
    epi_uuid_t          extensionType;
    dacl_format_label_t formatLabel;
    u_int32             numberBytes;
    char *              infoP;
} dacl_extended_info_t;
```

9.8 ACL Structure

The in-core **ACL** is described by the type **dacl_t**.

```
typedef struct dacl {
    epi_uuid_t           mgr_type_tag; /* ACL Manager type */
    epi_uuid_t           default_realm; /* Default cell UUID */
    u_int32              num_entries; /* the total number of entries
                                         stored on disk */
    dacl_simple_entry_t  simple_entries[MAX_SIMPLE_ENTRIES];
    dacl_complex_entry_t complex_entries[MAX_COMPLEX_ENTRIES];
} dacl_t;
```

Note: MAX_SIMPLE_ENTRIES = ((int)dacl_simple_entry_type_unauthmask) + 1
MAX_COMPLEX_ENTRIES = ((int)dacl_complex_entry_type_other) + 1

9.9 The Structures for Reading Lists of sec_id_t

These are used to avoid copying **uuid**s out of lists in the **pac**s. Their usage can be seen in Chapter 10 on page 171, *Access Control List Interface Functions*. A **pac** contains certified data that describes a principal's privilege attributes including principal name, groups of which the principal is a member and the native cell of the principal.

9.9.1 Define the epi_sec_id Structure

```
typedef struct epi_sec_id {
    epi_uuid_t    uuid;
    char *        name;
} epi_sec_id_t;
```

9.9.2 Define the epi_sec_id_foreign Structure

```
typedef struct epi_sec_id_foreign {
    epi_sec_id_t  id;
    epi_sec_id_t  realm;
} epi_sec_id_foreign_t;
```

Chapter 10

ACL Interface Functions

This chapter describes the functions available in the **ACL** interface by which the **ACL**s can be manipulated. The functions are used for both in-kernel filesystem **ACL**s and in-user-space administrative list **ACL**s.

Some functions have another interface specialized for filesystem **ACL**s. These specialized functions have a **dacl_epi** prefix typically and are indicated in the interface description.

Appendix B on page 229 contains a list of the return values, along with their explanations, that can returned by the functions in this chapter. Refer to it for those instances where the function lists a return value without a reason for it being returned.

NAME

dacl_CheckAccessId — perform authorization check

SYNOPSIS

```
long  dacl_CheckAccessId(
    dacl_t *         aclP,
    u_int32 *        permBitsP,
    dacl_permset_t * accessRequestedP,
    epi_uuid_t *     realmIdP,
    epi_uuid_t *     userIdP,
    epi_uuid_t *     groupIdP,
    epi_sec_id_t *   groupIdListP,
    unsigned int     numGroups,
    epi_sec_id_foreign_t * foreignGroupIdListP,
    unsigned int     numForeignGroups,
    epi_principal_id_t * userObjP,
    epi_principal_id_t * groupObjP,
    int              isAuthn
    );
```

ARGUMENTS

aclP	Acl for the object to which access is desired.
permBitsP	UNIX permission bits for the object, if applicable.
accessRequested	Access desired.
realmIdP	Cell to which the specified principal belongs. A NULL value or all zero UUID specifies the default cell (the cell to which the object belongs) .
userIdP	Principal desiring access to the object.
groupIdP	Primary group of principal .
groupIdListP	Supplementary local groups of which the principal is a member.
numGroups	Number of local groups.
foreignGroupIdListP	Supplementary groups in foreign cells of which the principal is a member.
numForeignGroups	Num of supplementary foreign groups.
userObjP	Principal who owns the object.
groupObjP	Group that owns the object.
isAuthn	Non-zero if the principal desiring access is authenticated.

DESCRIPTION

This function performs the authorization check to determine if the access desired by the specified principal to the object, corresponding to the specified **ACL**, is allowed. If *permBitsP* is non-NULL, the UNIX permission bits are taken into account while performing the authorization check.

This function has a filesystem specific version called *dacl_epi_CheckAccessId*( ) with the same interface.

RETURN VALUE

[DACL_ERROR_ACCESS_DENIED], if the access desired is not available in the **ACL**,

[DACL_ERROR_ACCESS_EXPLICITLY_DENIED] if there was a user match but the access requested is not permitted.

[DACL_ERROR_MGR_PARAMETER_ERROR] if the manager type check found a null principal id or permission bits.

[DACL_ERROR_PARAMETER_ERROR] if *accessRequestedP* was null.

Otherwise, if the access requested is granted, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_DetermineAccessAllowed( ) which is invoked by this routine as part of its determination.

NAME

dacl_DetermineAccessAllowed — determine access allowed

SYNOPSIS

```
long  dacl_DetermineAccessAllowed(
    dacl_t *         aclP,
    u_int32 *        permBitsP,
    dacl_permset_t * accessRequestedP,
    epi_uuid_t *     realmIdP,
    epi_uuid_t *     userIdP,
    epi_uuid_t *     groupIdP,
    epi_sec_id_t *   groupIdListP,
    unsigned int     numGroups,
    epi_sec_id_foreign_t * foreignGroupIdListP,
    unsigned int     numForeignGroups,
    epi_principal_id_t * userObjP,
    epi_principal_id_t * groupObjP,
    int              isAuthn,
    dacl_permset_t * accessAllowedP
    );
```

ARGUMENTS

aclP	Acl for the object to which access is desired.
permBitsP	UNIX permission bits for the object, if applicable.
accessRequested	Access desired.
realmIdP	Cell to which the specified principal belongs. A NULL value or all zero UUID specifies the default cell (the cell to which the object belongs).
userIdP	Principal desiring access to the object.
groupIdP	Primary group of principal.
groupIdListP	Supplementary local groups of which the principal is a member.
numGroups	Number of local groups.
foreignGroupIdListP	Supplementary groups in foreign cells of which the principal is a member.
numForeignGroups	Num of supplementary foreign groups.
userObjP	Principal who owns the object.
groupObjP	Group that owns the object.
isAuthn	Non-zero if the principal desiring access is authenticated.
accessAllowedP	If non-NULL, set to access allowed.

DESCRIPTION

This function determines the set of access allowed to the specified principal to the object corresponding to the specified **ACL**. If *permBitsP* is non-NULL, the UNIX permission bits are taken into account while determining the access allowed.

This function has a filesystem specific version called *dacl_epi_DetermineAccessAllowed*( ) with the same interface.

RETURN VALUE

[DACL_ERROR_ACCESS_DENIED], if the access desired is not available in the **ACL**,

[DACL_ERROR_ACCESS_EXPLICITLY_DENIED] if there was a user match but the access requested is not permitted.

[DACL_ERROR_MGR_PARAMETER_ERROR] if the manager type check found a null principal id or permission bits.

[DACL_ERROR_PARAMETER_ERROR] if *accessRequestedP* was null.

Otherwise, if the access requested is granted, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_CheckAccessId( ) which invokes this routine as part of its determination.

NAME

dacl_CheckAccessPac — perform authorization check

SYNOPSIS

```
long  dacl_CheckAccessPac(
    dacl_t *         aclP,
    u_int32 *        permBitsP,
    dacl_permset_t * accessRequestedP,
    sec_id_pac_t *   pacP,
    epi_principal_id_t * userObjP,
    epi_principal_id_t * groupObjP
    );
```

ARGUMENTS

aclP	**ACL** for the object to which access is desired.
permBitsP	UNIX permission bits for the object, if applicable.
accessRequested	Access desired.
pacP	The PAC[7] for the principal desiring access to the object .
userObjP	Principal who owns the object.
groupObjP	Group that owns the object.

DESCRIPTION

This function performs the authorization check to determine if the access desired by the principal, corresponding to the specified PAC, to the object, corresponding to the specified **ACL**, is allowed. If *permBitsP* is non-NULL, the UNIX permission bits are taken into account while performing the authorization check.

This function has a file system specific version called *dacl_epi_CheckAccessPac*( ) with the same interface through DCE 1.1. For versions of DCE newer than 1.1, the two interfaces are no longer the same.

RETURN VALUE

[DACL_ERROR_ACCESS_DENIED], if the access desired is not available in the **ACL**,

Otherwise, if the access requested is granted, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_CheckAccessAllowedPac( ) which is invoked by this routine as part of its determination.

---

7. A PAC is an acronym of *Privilege Attribute Certificate*. It contains certified data that describes a principal's privilege attributes including principal name, groups of which the principal is a member and the native cell of the principal.

NAME

dacl_epi_CheckAccessPac — perform authorization check

SYNOPSIS

```
long  dacl_epi_CheckAccessPac(
    dacl_t *         aclP,
    u_int32 *        permBitsP,
    dacl_permset_t * accessRequestedP,
    sec_id_pac_t *   pacP,
    epi_principal_id_t * userObjP,
    epi_principal_id_t * groupObjP,
    int              networkRootCheck
    );
```

ARGUMENTS

aclP	**ACL** for the object to which access is desired.
permBitsP	UNIX permission bits for the object, if applicable.
accessRequested	Access desired.
pacP	The PAC[8] for the principal desiring access to the object.
userObjP	Principal who owns the object.
groupObjP	Group that owns the object.
networkRootCheck	This is a new argument for versions of DCE greater than 1.1. If true, a check is to be made for the network root ID which has full access. If true, and *pacP* principal UUID is the network root ID, this principal has all rights and the **ACL** is ignored.

DESCRIPTION

This function performs the authorization check to determine if the access desired by the principal, corresponding to the specified PAC, to the object, corresponding to the specified **ACL**, is allowed. If *permBitsP* is non-NULL, the UNIX permission bits are taken into account while performing the authorization check.

This function has a non-file system specific version called *dacl_CheckAccessPac*( ) with the same interface for versions of DCE prior to DCE 1.2. In DCE 1.2, the argument *networkRootCheck* has been added.

RETURN VALUE

[DACL_ERROR_ACCESS_DENIED], if the access desired is not available in the **ACL**,

Otherwise, if the access requested is granted, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_epi_CheckAccessAllowedPac( ) which is invoked by this routine as part of its determination.

---

8. A PAC is an acronym of *Privilege Attribute Certificate*. It contains certified data that describes a principal's privilege attributes including principal name, groups of which the principal is a member and the native cell of the principal.

NAME

dacl_CheckAccessAllowedPac — determine access allowed

SYNOPSIS

```
long  dacl_CheckAccessAllowedPac(
    dacl_t *         aclP,
    u_int32 *        permBitsP,
    dacl_permset_t * accessRequestedP,
    sec_id_pac_t *   pacP,
    epi_principal_id_t * userObjP,
    epi_principal_id_t * groupObjP,
    dacl_permset_t * accessAllowedP
    );
```

ARGUMENTS

aclP	**ACL** for the object to which access is desired.
permBitsP	UNIX permission bits for the object, if applicable.
accessRequested	Access desired.
pacP	The PAC for the principal desiring access to the object.
userObjP	Principal who owns the object.
groupObjP	Group that owns the object.
accessAllowedP	If non-NULL, set to access allowed.

DESCRIPTION

This function determines the set of access allowed to the principal, corresponding to the specified PAC, to the object, corresponding to the specified **ACL**. If *permBitsP* is non-NULL, the UNIX permission bits are taken into account while determining the access allowed.

This function has a filesystem specific version called *dacl_epi_CheckAccessAllowedPac*( ) with the same interface if the version of DCE is not newer then 1.1. For DCE 1.2, the interfaces are no longer the same, as a new argument, *networkRootCheck*, has been added to *dacl_epi_CheckAccessAllowedPac*( ) after DCE 1.1.

RETURN VALUE

[DACL_ERROR_ACCESS_DENIED], if the access desired is not available in the **ACL**,

Otherwise, if the access requested is granted, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_CheckAccessPac( ) which invokes this routine as part of its determination.

NAME

dacl_epi_CheckAccessAllowedPac — determine access allowed

SYNOPSIS

```
long  dacl_epi_CheckAccessAllowedPac(
    dacl_t *         aclP,
    u_int32 *        permBitsP,
    dacl_permset_t * accessRequestedP,
    sec_id_pac_t *   pacP,
    epi_principal_id_t * userObjP,
    epi_principal_id_t * groupObjP,
    int              networkRootCheck,
    dacl_permset_t * accessAllowedP
    );
```

ARGUMENTS

aclP	**ACL** for the object to which access is desired.
permBitsP	UNIX permission bits for the object, if applicable.
accessRequested	Access desired.
pacP	The PAC for the principal desiring access to the object.
userObjP	Principal who owns the object.
groupObjP	Group that owns the object.
networkRootCheck	This is a new argument for versions of DCE greater than 1.1. If true, a check is to be made for the network root ID which has full access. If true, and *pacP* principal UUID is the network root ID, this principal has all rights and the **ACL** is ignored.
accessAllowedP	If non-NULL, set to access allowed.

DESCRIPTION

This function determines the set of access allowed to the principal, corresponding to the specified PAC, to the object, corresponding to the specified **ACL**. If *permBitsP* is non-NULL, the UNIX permission bits are taken into account while determining the access allowed.

This function has a non-filesystem specific version called *dacl_CheckAccessAllowedPac*( ) with the same interface through DCE 1.1. After that, the interfaces are not the same, as this function then checks for the network root ID and grants it certain priviledges.

RETURN VALUE

[DACL_ERROR_ACCESS_DENIED], if the access desired is not available in the **ACL**,

Otherwise, if the access requested is granted, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_epi_CheckAccessPac( ) which invokes this routine as part of its determination.

NAME

dacl_PacFromUcred — construct a PAC from UNIX credentials

SYNOPSIS

```
void  dacl_PacFromUcred(
    sec_id_pac_t *    pacP,
    struct ucred *    ucredP
    );
```

ARGUMENTS

pacP The PAC.

ucredP The UNIX credentials.

DESCRIPTION

This function constructs a PAC corresponding to the specified UNIX credentials of a principal. If the local group pointer in the specified PAC is non-NULL, the memory associated with the local group list is reused to fill it with group information from the specified UNIX credentials. This assumes that the group list memory reused is big enough to hold all the groups from the ucred.

If the local group pointer is NULL, this routine will allocate a block of memory big enough to hold all the groups in the ucred. This memory must be released by the caller by calling osi_Free(pacP->groups, pacP->num_groups).

The foreign group list of the pacP is preserved. If there are any foreign groups associated with the pacP, pacP->foreign_groups and pacP->num_foreign_groups must be set appropriately. All other fields in the pacP will be modified, as appropriate.

Note: The caller must ensure that the group list is big enough and that the memory is released appropriately.

RETURN VALUE

None.

ERRORS

None.

NAME

dacl_FlattenAclWithModeBits — pack an **ACL**

SYNOPSIS

```
long  dacl_FlattenAclWithModeBits(
    dacl_t *       aclP,
    char **        fileBufferPP,
    unsigned int * bytesInBufferP,
    u_int32        modeBits,
    int            useModeBits,
    int            flattenForDisk
    );
```

ARGUMENTS

aclP	The in-core **ACL**.
fileBufferPP	The buffer to hold the packed **ACL**.
bytesInBufferP	Size of the buffer.
modeBits	UNIX permission bits for the object, if applicable.
useModeBits	Flag that whether the UNIX permission bits should be used.
flattenForDisk	Essentially specifies principal id format to be used in the packed **ACL**[9].

DESCRIPTION

This function converts an in-core **ACL** to its byte-stream representation. If the *useModeBits* flag is set, the packed **ACL** is modified to correspond to the UNIX permission bits.

RETURN VALUE

[DACL_ERROR_TOO_FEW_BYTES], if *bytesInBufferP* does not contain enough space. In this case, no processing is done.

[DACL_ERROR_PARAMETER_ERROR], if *aclP* or *fileBufferPP* or *bytesInBufferP* is not valid (null). In this instance the appropriate error(s) is(are) logged.

Otherwise, [DACL_SUCCESS] is returned to indicate a successful conversion.

ERRORS

None.

SEE ALSO

This function also uses the following routines — *Epi_PrinId_FromUuid*( ), *hton_epi_principal_id*( ), *hton_epi_uuid*( ).

---

9. This *flattenForDisk* parameter provides the support for storing principal ids in the 32 bits format for filesystem **ACL**s. When the corresponding argument is zero, the flattened **ACL** always has 128 bit principal ids. If the argument is non-zero, then the code performs a runtime check to determine the size of principal ids in the flat **ACL**. If other appropriate flags have been defined, this would result in 32 bits principal ids being used in the flattened **ACL**. However, it was decided not to use 32 bits format and currently, the runtime check should always result in 128 bits format being used. Also note, that in-core **ACL**s always store principal ids in the 128 bits format.

NAME

dacl_FlattenAcl — pack an **ACL**

SYNOPSIS

```
long  dacl_FlattenAcl(
    dacl_t *       aclP,
    char **        fileBufferPP,
    unsigned int * bytesInBufferP
    );
```

ARGUMENTS

aclP	The in-core **ACL**.
fileBufferPP	The buffer to hold the packed **ACL**.
bytesInBufferP	Size of the buffer.

DESCRIPTION

This function converts an in-core **ACL** to its byte-stream representation. There are no UNIX permission bits considered in This function. The size of principal ids is determined at run time. This is usually used when a filesystem **ACL** is flattened to be stored to disk.

This function uses *dacl_FlattenAclWithModeBits*( ) with *modeBits* set to DACL_NO_MODE_BITS (not applicable), *useModeBits* set to 0 (don't use mode bits), and *flattenForDisk* set to 1 (flatten for disk).

This function has a filesystem specific version called *dacl_epi_FlattenAcl* with a different interface described below.

RETURN VALUE

[DACL_ERROR_TOO_FEW_BYTES], if *bytesInBufferP* does not contain enough space. In this case, no processing is done.

[DACL_ERROR_PARAMETER_ERROR], if *aclP* or *fileBufferPP* or *bytesInBufferP* is not valid (null). In this instance the appropriate error(s) is(are) logged.

Otherwise, [DACL_SUCCESS] is returned to indicate a successful conversion.

ERRORS

None.

SEE ALSO

dacl_FlattenAclWithModeBits( ). This function also uses the following routines — *Epi_PrinId_FromUuid*( ), *hton_epi_principal_id* ( ), *hton_epi_uuid*( ).

NAME

dacl_epi_FlattenAcl — pack an **ACL**

SYNOPSIS

```
long  dacl_epi_FlattenAcl(
    dacl_t *       aclP,
    char **        fileBufferPP,
    unsigned int * bytesInBufferP,
    u_int32        modeBits
    );
```

ARGUMENTS

aclP	The in-core **ACL**.
fileBufferPP	The buffer to hold the packed **ACL**.
bytesInBufferP	Size of the buffer.
modeBits	UNIX permission bits for the object, if applicable.

DESCRIPTION

This function converts an in-core **ACL** to its byte-stream representation. The packed **ACL** is modified to correspond to the specified UNIX permission bits. The 128 bit principal id format is used in the flattened **ACL**.

This function uses *dacl_FlattenAclWithModeBits*( ) with *modeBits* set to the applicable mode bits, *useModeBits* set to 1 (use mode bits), and *flattenForDisk* set to 0 (not for disk).

RETURN VALUE

[DACL_ERROR_TOO_FEW_BYTES], if *bytesInBufferP* does not contain enough space. In this case, no processing is done.

[DACL_ERROR_PARAMETER_ERROR], if *aclP* or *fileBufferPP* or *bytesInBufferP* is not valid (null). In this instance the appropriate error(s) is(are) logged.

Otherwise, [DACL_SUCCESS] is returned to indicate a successful conversion.

ERRORS

None.

SEE ALSO

dacl_FlattenAclWithModeBits( ). This function also uses the following routines — *Epi_PrinId_FromUuid*( ), *hton_epi_principal_id*( ), *hton_epi_uuid*( ).

NAME

dacl_ParseAclDiskOption — unpack a flattened **ACL**

SYNOPSIS

```
long  dacl_ParseAclDiskOption(
    char *       fileBufferP,
    int          bytesInBuffer,
    dacl_t *     aclBufferP,
    epi_uuid_t * mgrUuidP,
    int          parseFromDisk
    );
```

ARGUMENTS

fileBufferPP	The buffer holding the packed **ACL**.
bytesInBufferP	Size of the packed **ACL**.
aclBufferP	The in-core **ACL**.
mgrUuidP	The UUID for the **ACL** manager.
parseFromDisk	Essentially specifies principal id format in the specified packed **ACL**[10]>

DESCRIPTION

This function parses the given packed byte-stream format **ACL** and constructs an in-core **ACL**.

It uses the following functions in the parse operation - *ntoh_epi_uuid*( ) and *Epi_PrinId_toUuid*( ). If a parameter error is found, it is logged.

RETURN VALUE

[DACL_ERROR_BUFFER_ALLOCATION], [DACL_ERROR_INCORRECT_MGR_UUID], [DACL_ERROR_TOO_FEW_BYTES], [DACL_ERROR_TOO_MANY_BYTES], [DACL_ERROR_DUPLICATE_ENTRY_FOUND], [DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE], [DACL_ERROR_ENTRY_TYPE_TOO_LARGE], [DACL_ERROR_REQUIRED_ENTRY_MISSING] or [DACL_ERROR_PARAMETER_ERROR].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_FlattenAclWithModeBits( ). This function uses the following routines — *ntoh_epi_uuid*( ) and *Epi_PrinId_toUuid*( ). Also, see Appendix B on page 229 for a list of return values and their meanings.

---

10. Refer to earlier footnote in *dacl_FlattenAclWithModeBits*( ) about *flattenForDisk* parameter.

NAME

dacl_ParseSyscallAcl — unpack a flattened **ACL**

SYNOPSIS

```
long  dacl_ParseSyscallAcl(
    char *        fileBufferP,
    int           bytesInBuffer,
    dacl_t *      aclBufferP,
    epi_uuid_t *  mgrUuidP
    );
```

ARGUMENTS

fileBufferPP	The buffer holding the packed **ACL**.
bytesInBufferP	Size of the packed **ACL**.
aclBufferP	The in-core **ACL**.
mgrUuidP	The UUID for the **ACL** manager.

DESCRIPTION

This function parses the given packed byte-stream format **ACL** and constructs an in-core **ACL**. The packed **ACL** is assumed to store principal ids in the 128 bits format.

It uses the function, *dacl_ParseAclDiskOption*( ) with *parseFromDisk* set to 0 (parse from the syscall buffer).

RETURN VALUE

[DACL_ERROR_BUFFER_ALLOCATION], [DACL_ERROR_INCORRECT_MGR_UUID], [DACL_ERROR_TOO_FEW_BYTES], [DACL_ERROR_TOO_MANY_BYTES], [DACL_ERROR_DUPLICATE_ENTRY_FOUND], [DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE], [DACL_ERROR_ENTRY_TYPE_TOO_LARGE], [DACL_ERROR_REQUIRED_ENTRY_MISSING], or [DACL_ERROR_PARAMETER_ERROR].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_FlattenAclWithModeBits( ) and *dacl_ParseAclDiskOption*( ). Also, see Appendix B on page 229 for a list of return values and their meanings.

NAME

dacl_ParseAcl — unpack a flattened **ACL**

SYNOPSIS

```
long  dacl_ParseAcl(
    char *        fileBufferP,
    int           bytesInBuffer,
    dacl_t *      aclBufferP,
    epi_uuid_t *  mgrUuidP
    );
```

ARGUMENTS

fileBufferPP	The buffer holding the packed **ACL**.
bytesInBufferP	Size of the packed **ACL**.
aclBufferP	The in-core **ACL**.
mgrUuidP	The UUID for the **ACL** manager.

DESCRIPTION

This function parses the given packed byte-stream format **ACL** and constructs an in-core **ACL**. The principal id format used in the packed **ACL** is determined at runtime.

It uses the function, *dacl_ParseAclDiskOption*( ) with *parseFromDisk* set to 1 (parse from the disk buffer).

RETURN VALUE

[DACL_ERROR_BUFFER_ALLOCATION], [DACL_ERROR_INCORRECT_MGR_UUID], [DACL_ERROR_TOO_FEW_BYTES], [DACL_ERROR_TOO_MANY_BYTES], [DACL_ERROR_DUPLICATE_ENTRY_FOUND], [DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE], [DACL_ERROR_ENTRY_TYPE_TOO_LARGE], [DACL_ERROR_REQUIRED_ENTRY_MISSING], or [DACL_ERROR_PARAMETER_ERROR].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_FlattenAclWithModeBits( ) and *dacl_ParseAclDiskOption*( ). Also, see Appendix B on page 229 for a list of return values and their meanings.

NAME

dacl_ExtractPermBits — extract UNIX permission bits from an **ACL**

SYNOPSIS

```
long  dacl_ExtractPermBits(
    dacl_t *     daclP,
    u_int32 *    permBitsP
    );
```

ARGUMENTS

daclP The **ACL**.

permBitsP The UNIX permission bits corresponding to the **ACL**.

DESCRIPTION

This function determines the UNIX permission bits corresponding to the **ACL**. Refer to Section 8.6 on page 160, for the correspondence between UNIX permission bits and the **ACL** access rights.

RETURN VALUE

[DACL_ERROR_PARAMETER_ERROR], if a parameter error is found, which is also logged.

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

The correspondence between UNIX permission bits and **ACL** access rights can be found in Section 8.6 on page 160 in the chapter, *Access Control List Overview*.

NAME

dacl_ChmodAcl — synchronize **ACL** with UNIX permission bits

SYNOPSIS

```
long  dacl_ChmodAcl(
    dacl_t *   daclP,
    u_int32    permBits,
    int        forDirectory
    );
```

ARGUMENTS

daclP	The **ACL** .
permBits	The UNIX permission bits.
forDirectory	Flag to specify a directory object **ACL**.

DESCRIPTION

This function modifies the **ACL** entries to correspond to the specified UNIX permission bits. Refer to Section 8.6 on page 160 in the chapter, *Access Control List Overview*, for the correspondence between UNIX permission bits and the **ACL** access rights. If the *forDirectory* flag is set, UNIX *write* permission results in *write*, *insert* and *delete* access rights being granted in the corresponding **ACL** entry.

RETURN VALUE

[DACL_ERROR_PARAMETER_ERROR], if a parameter error is found (no **ACL** is passed), which is also logged.

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

NAME

dacl_FreeAclEntries — free memory allocated for **ACL** entries

SYNOPSIS

```
long  dacl_FreeAclEntries(
    dacl_t *    theAclP
    );
```

ARGUMENTS

theAclP The **ACL**.

DESCRIPTION

This function releases the memory allocated for the complex entries in the **ACL**.

RETURN VALUE

[DACL_ERROR_PARAMETER_ERROR], if a parameter error is found (no **ACL** is passed), which is also logged.

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

NAME

dacl_PrintAclEntry — print an **ACL** entry

SYNOPSIS

```
long  dacl_PrintAclEntry(
    FILE *          stream,
    dacl_entry_t *  aclEntryP,
    epi_uuid_t *    defaultRealmUuidP,
    int             local
    );
```

ARGUMENTS

stream	Output file stream.
aclEntryP	The **ACL** entry.
defaultRealmUuidP	The native cell UUID.
local	Specifies whether ids should be interpreted locally or via DCE security registry.

DESCRIPTION

This function outputs an ascii representation of the **ACL** entry to the specified file stream. The various principal/group/organization ids present in the **ACL** entry are translated into their symbolic representations. Translations are done either locally or via the security service depending on the *local* flag.

RETURN VALUE

[DACL_ERROR_UNRECOGNIZED_USER_OR_GROUP], [DACL_ERROR_SEC_RGY_PGO_ERROR] or [DACL_ERROR_NONLOCAL_ENTRY_TYPE].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

This function also calls *dacl_NameAndTypeStringsFromEntry*( ), *dacl_EntryType_ToString*( ), and *dacl_Permset_ToString*( ).

NAME

dacl_PrintAcl — print an **ACL**

SYNOPSIS

```
long  dacl_PrintAcl(
    FILE *      stream,
    dacl_t *    aclP,
    int         local
    );
```

ARGUMENTS

stream	Output file stream.
aclEntryP	The **ACL**.
local	Specifies whether ids should be interpreted locally or via DCE security registry.

DESCRIPTION

This function outputs an ascii representation of the **ACL** to the specified file stream. The various principal/group/organization ids present in the **ACL** are translated into their symbolic representations. Translations are done either locally or via the security service depending on the *local* flag.

RETURN VALUE

[DACL_ERROR_SEC_RGY_PGO_ERROR], if the cell name cannot be found.

[DACL_ERROR_UNRECOGNIZED_USER_OR_GROUP] or
[DACL_ERROR_NONLOCAL_ENTRY_TYPE].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

This function also calls *dacl_PrintAclEntry*( ).

NAME

dacl_WriteToDisk — store an **ACL** into a file

SYNOPSIS

```
long  dacl_WriteToDisk(
    dacl_t *  aclP,
    char *    filenameP
    );
```

ARGUMENTS

aclP The **ACL**.

filenameP The file.

DESCRIPTION

This constructs a packed version of the specified in-core **ACL** and stores it in the specified file. There are no UNIX permission bits considered in This function. The principal id format for the packed **ACL** is determined at run time.

If a buffer was allocated for packing and a failure occurs in processing, the buffer is freed.

RETURN VALUE

[DACL_ERROR_FS_OPEN], [DACL_ERROR_FS_WRITE], [DACL_ERROR_FS_CLOSE], [DACL_ERROR_TOO_FEW_BYTES] or [DACL_ERROR_PARAMETER_ERROR].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

This function also calls *dacl_FlattenAcl*( ) to flatten for disk, *dacl_epi_FlattenAcl*( ) to flatten into the caller's buffer, *dacl_ValidateBuffer*( ) to validate the **ACL** for correctness.

NAME

dacl_CreateAclOnDisk — construct an **ACL** and store it

SYNOPSIS

```
long  dacl_CreateAclOnDisk(
    char *            filenameP,
    epi_uuid_t *      mgrUuidP,
    epi_uuid_t *      defaultRealmUuidP,
    dacl_permset_t *  userObjPermsP,
    dacl_permset_t *  groupObjPermsP,
    dacl_permset_t *  otherObjPermsP,
    int               useEpisodeFile,
    long              epiSyscallFlags
    );
```

ARGUMENTS

filenameP	The file for the **ACL** for non-filesystem **ACL**s; otherwise the filesystem object for the **ACL**.
mgrUuidP	The UUID for the **ACL** manager.
defaultRealmUuidP	The native cell UUID.
userObjPermsP	The permission set for the *user_obj* entry.
groupObjPermsP	The permission set for the *group_obj* entry.
otherObjPermsP	The permission set for the *other_obj* entry.
useEpisodeFile	Filesystem **ACL** indicator flag.
epiSyscallFlags	Flags used in a system call if its a filesystem **ACL**.

DESCRIPTION

This function constructs an **ACL** having only *user_obj*, *group_obj* and *other_obj* simple entries and converts it into a packed byte-stream. If *useEpisodeFile* is non-zero, it indicates that the **ACL** is a filesystem **ACL** and *filenameP* specifies the filesystem object for which the **ACL** is being set. Otherwise, its a non-filesystem **ACL** and *fileNameP* specifies the file to store the packed **ACL**.

RETURN VALUE

[DACL_ERROR_ACL_FILE_EXISTS] or [DACL_ERROR_FS_STAT].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

This function also calls *dacl_WriteToDisk*( ) if this is a non-filesystem **ACL**, *dacl_epi_FlattenAcl*( ) to flatten for filesystems if this is a filesystem object, *dacl_FreeAclEntries*, to free memory allocated for **ACL** entries.

NAME

dacl_ReadFromDisk — read an **ACL** from disk and unpack it

SYNOPSIS

```
long  dacl_ReadFromDisk(
    dacl_t *      aclP,
    char *        filenameP,
    epi_uuid_t *  mgrUuidP
    );
```

ARGUMENTS

aclP	The resultant in-core **ACL**.
filenameP	The file storing the packed **ACL**.
mgrUuidP	UUID for the **ACL** manager.

DESCRIPTION

This function reads the packed **ACL** from the specified file and unpacks it into the in-core **ACL** format. The format of principal id in the packed **ACL** is determined at runtime.

RETURN VALUE

[DACL_ERROR_FS_READ], if the **ACL** could not be read from disk.

[DACL_ERROR_BUFFER_ALLOCATION], [DACL_ERROR_FS_STAT], if the **ACL** could not be found.

[DACL_ERROR_FS_CLOSE], if the file descriptor for the **ACL** could not be closed,

[DACL_ERROR_ACLFILE_NOT_FOUND], [DACL_ERROR_FS_OPEN], as well as those listed for *dacl_ParseAcl*( ).

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_ParseAcl( ) which unpacks the **ACL** read from disk,

NAME

dacl_AddEntryToAcl — add an entry to an **ACL**

SYNOPSIS

```
long  dacl_AddEntryToAcl(
    dacl_t *      aclP,
    char *        typeStringP,
    char *        nameStringP,
    char *        permStringP,
    char *        dataStringP,
    epi_uuid_t *  mgrUuidP,
    int           local
    );
```

ARGUMENTS

aclP	The **ACL**.
typeStringP	**ACL** entry type, in symbolic format.
nameStringP	Entry name, in symbolic format for complex **ACL** entries.
permStringP	**ACL** entry access permissions, in symbolic format.
dataStringP	Information for extended **ACL** entry type.
mgrUuidP	The **ACL** manager UUID.
local	Specifies whether the entry name should be interpreted locally or via DCE security registry.

DESCRIPTION

This function constructs an **ACL** entry with the specified characteristics and adds it to the specified **ACL**. The *nameStringP* parameter is only used for a complex **ACL** entry type and the *dataStringP* parameter only for an extended **ACL** entry type.

RETURN VALUE

[DACL_ERROR_ENTRY_EXISTS], [DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE] or [DACL_ERROR_ENTRY_TYPE_TOO_LARGE].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_InitAclEntryFromStrings( ).

NAME

dacl_ModifyAclEntry — modify an **ACL** entry

SYNOPSIS

```
long  dacl_ModifyAclEntry(
    dacl_t *  aclP,
    char *    typeStringP,
    char *    nameStringP,
    char *    permStringP,
    char *    dataStringP,
    int       local
    );
```

ARGUMENTS

aclP	The **ACL**.
typeStringP	**ACL** entry type, in symbolic format.
nameStringP	Entry name, in symbolic format for complex **ACL** entries.
permStringP	**ACL** entry access permissions, in symbolic format.
dataStringP	Information for extended **ACL** entry type.
local	Specifies whether the entry name should be interpreted locally or via DCE security registry.

DESCRIPTION

This function modifies the specified **ACL** entry to have the specified characteristics.

RETURN VALUE

[DACL_ERROR_ENTRY_NOT_FOUND], [DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE], [DACL_ERROR_ILLEGAL_ENTRY], [DACL_ERROR_UNRECOGNIZED_USER_OR_GROUP], [DACL_ERROR_NONLOCAL_ENTRY_TYPE], [DACL_ERROR_SEC_RGY_PGO_ERROR] or [DACL_ERROR_MISSING_NAME].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_InitAclEntryFromStrings( ).

NAME

dacl_DeleteAclEntry — delete an **ACL** entry

SYNOPSIS

```
long  dacl_DeleteAclEntry(
    dacl_t *      aclP,
    char *        typeStringP,
    char *        nameStringP,
    epi_uuid_t *  mgrUuidP,
    int           local
    );
```

ARGUMENTS

aclP	The **ACL**.
typeStringP	**ACL** entry type, in symbolic format.
nameStringP	Entry name, in symbolic format for complex **ACL** entries.
mgrUuidP	The **ACL** manager UUID.
local	Specifies whether the entry name should be interpreted locally or via DCE security registry.

DESCRIPTION

This function deletes the specified entry from an **ACL**.

RETURN VALUE

[DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE], [DACL_ERROR_ILLEGAL_ENTRY], [DACL_ERROR_UNRECOGNIZED_USER_OR_GROUP], [DACL_ERROR_NONLOCAL_ENTRY_TYPE], [DACL_ERROR_SEC_RGY_PGO_ERROR], [DACL_ERROR_MISSING_NAME], [DACL_ERROR_ENTRY_NOT_FOUND] or [DACL_ERROR_ENTRY_REQUIRED].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_InitAclEntryFromStrings( ).

NAME

dacl_DeleteAllEntries — invalidate all **ACL** entries

SYNOPSIS

```
long  dacl_DeleteAllEntries(
    dacl_t *    aclP
    );
```

ARGUMENTS

aclP The **ACL**.

DESCRIPTION

This function invalidates all entries in an **ACL** and releases any allocated memory for complex entries.

RETURN VALUE

[DACL_ERROR_PARAMETER_ERROR], if a parameter error is found (no **ACL** is passed), which is also logged.

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_FreeAclEntries( ).

NAME

sec_acl_FlattenAcl — convert a DCE **ACL** into a flat DFS **ACL**

SYNOPSIS

```
long  sec_acl_FlattenAcl(
    sec_acl_t *    secAclP,
    char **        byteBufferPP,
    unsigned int * bytesInBufferP
    );
```

ARGUMENTS

secAclP	The DCE **ACL**.
byteBufferPP	The buffer to hold the packed DFS **ACL**.
bytesInBufferP	Size of the buffer.

DESCRIPTION

This function converts a DCE **ACL** into a packed DFS **ACL**. No UNIX permission are used as DCE objects have no such construct. The 128 bits principal id format is used in the packed DFS **ACL**.

RETURN VALUE

[DACL_ERROR_BUFFER_ALLOCATION], [DACL_ERROR_DUPLICATE_ENTRY_FOUND], [DACL_ERROR_INCORRECT_MGR_UUID] or [DACL_ERROR_PARAMETER_ERROR].

The following can be returned from called functions. [DACL_ERROR_TOO_FEW_BYTES].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_FlattenAclWithModeBits( ) and *dacl_FreeAclEntries*( ).

NAME

sec_acl_ParseAcl — convert a flat DFS **ACL** into a DCE **ACL**

SYNOPSIS

```
long  sec_acl_ParseAcl(
    sec_acl_t *   secAclP,
    char *        byteBufferP,
    unsigned int  bytesInBuffer
    );
```

ARGUMENTS

secAclP The DCE **ACL**.

byteBufferPP The buffer holding the packed DFS **ACL**.

bytesInBufferP Size of the packed DFS **ACL**.

DESCRIPTION

This function converts a packed DFS **ACL** into a DCE **ACL**. The packed DFS **ACL** is assumed to store the principal ids in the 128 bit format.

RETURN VALUE

[DACL_ERROR_PARAMETER_ERROR], if a parameter error is found (no **ACL** is passed), which is also logged.

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_FreeAclEntries( ).

NAME

dacl_InitAclEntryFromStrings — construct an **ACL** entry

SYNOPSIS

```
long  dacl_InitAclEntryFromStrings(
    dacl_entry_t *  aclEntryP,
    epi_uuid_t *    defaultRealmUuidP,
    char *          typeStringP,
    char *          nameStringP,
    char *          permStringP,
    char *          dataStringP,
    epi_uuid_t *    mgrUuidP,
    int             local
    );
```

ARGUMENTS

aclEntryP	The resultant **ACL** entry.
defaultRealmUuidP	The native cell UUID.
typeStringP	**ACL** entry type, in symbolic format.
nameStringP	Entry name, in symbolic format for complex **ACL** entries.
permStringP	**ACL** entry access permissions, in symbolic format.
dataStringP	Information for extended **ACL** entry type.
mgrUuidP	The **ACL** manager UUID.
local	Specifies whether the entry name should be interpreted locally or via DCE security registry.

DESCRIPTION

This function constructs an **ACL** entry with the specified characteristics.

RETURN VALUE

[DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE], [DACL_ERROR_ILLEGAL_ENTRY], [DACL_ERROR_UNRECOGNIZED_USER_OR_GROUP], [DACL_ERROR_NONLOCAL_ENTRY_TYPE], [DACL_ERROR_SEC_RGY_PGO_ERROR] or [DACL_ERROR_MISSING_NAME].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_EntryType_FromString( ).

NAME

dacl_NameAndTypeStringsFromEntry — generate ascii representations of complex **ACL** entry name and type

SYNOPSIS

```
long  dacl_NameAndTypeStringsFromEntry(
    dacl_entry_t *  aclEntryP,
    epi_uuid_t *    defaultRealmUuidP,
    char **         typeStringPP,
    sec_rgy_name_t  globalName,
    int             local
    );
```

ARGUMENTS

aclEntryP	The **ACL** entry.
defaultRealmUuidP	The native cell UUID.
typeStringP	Resultant symbolic type of entry type.
globalName	Resultant symbolic name of entry principal.
local	Specifies whether the entry id should be interpreted locally or via DCE security registry.

DESCRIPTION

This function generates the ascii string representations of the name and type of the complex **ACL** entry. The various principal/group/organization ids present in the **ACL** entry are translated into their symbolic representations. Translations are done either locally or via the security service depending on the *local* flag.

RETURN VALUE

[DACL_ERROR_UNRECOGNIZED_USER_OR_GROUP], [DACL_ERROR_SEC_RGY_PGO_ERROR] or [DACL_ERROR_NONLOCAL_ENTRY_TYPE].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

Also, see Appendix B on page 229 for a list of return values and their meanings.

NAME

dacl_EntryType_ToString — generate symbolic representation of **ACL** entry type

SYNOPSIS

```
char *  dacl_EntryType_ToString(
    dacl_entry_type_t    probeType
    );
```

ARGUMENTS

probeType The **ACL** entry type.

DESCRIPTION

This function generates an ascii string representation from the specified internal representation of the **ACL** entry type.

RETURN VALUE

None.

ERRORS

None.

NAME

dacl_EntryType_FromString — generate internal representation of symbolic **ACL** entry type

SYNOPSIS

```
long  dacl_EntryType_FromString(
    dacl_entry_type_t *  probeTypeP,
    char *               probeString
    );
```

ARGUMENTS

probeType The **ACL** entry type.

probeStringP Symbolic representation of the **ACL** entry type.

DESCRIPTION

This function generates an internal representation of an **ACL** entry type from the specified ascii string representation.

RETURN VALUE

[DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE] or [DACL_SUCCESS].

ERRORS

None.

NAME

dacl_Permset_ToString — generate a symbolic representation of **ACL** entry permissions

SYNOPSIS

```
void  dacl_Permset_ToString(
    dacl_permset_t *  thePermSetP,
    char *            stringBufferP
    );
```

ARGUMENTS

thePermSetP The **ACL** entry permissions bitset.

stringBufferP The buffer to hold the symbolic representation.

DESCRIPTION

This function generates the symbolic string representation of the specified **ACL** entry permissions bitset.

RETURN VALUE

None.

ERRORS

None.

NAME

dacl_Permset_FromString — generate internal representation of symbolic **ACL** entry permissions

SYNOPSIS

```
void  dacl_Permset_FromString(
    dacl_permset_t * thePermSetP,
    char *            permStringP
    );
```

ARGUMENTS

thePermSetP The generated permission bitset.

permStringP Symbolic **ACL** entry permissions.

DESCRIPTION

This function generates the internal representation of **ACL** entry permissions, a bitset, from its symbolic ascii string representation.

RETURN VALUE

None.

ERRORS

None.

NAME

dacl_ValidateBuffer — parse and validate an **ACL**

SYNOPSIS

```
long  dacl_ValidateBuffer(
    char *          byteBufferP,
    unsigned int *  bytesInBufferP,
    epi_uuid_t *    mgrUuidP,
    int             makeMinorRepairs,
    dacl_t **       daclPP
    );
```

ARGUMENTS

byteBufferP	The packed format of the **ACL**.
bytesInBufferP	Size of the packed **ACL**.
mgrUuidP	The UUID of the **ACL** manager.
makeMinorRepairs	Flag that causes correction of minor errors detected during validation.
daclPP	The parsed and validate **ACL**.

DESCRIPTION

This function unpacks the given packed **ACL** into its in-core format. The 128 bit principal ids format is assumed in the packed **ACL**. The resultant in-core **ACL** is then validated for correctness. If *makeMinorRepairs* argument is set, any minor errors detected are corrected.

This function has a filesystem specific version called *dacl_epi_ValidateBuffer*( ) with the same interface.

RETURN VALUE

[DACL_ERROR_PARAMETER_ERROR], which is also logged, [DACL_ERROR_VALIDATION_FAILURE], if the *user_obj* entry does not have **c** permission, or if the *mask_obj* entry is missing. It can also indicate that an entry specifies *foreign_user* within the default realm.

The following values can be returned from the called functions, below:
[DACL_ERROR_BUFFER_ALLOCATION], [DACL_ERROR_INCORRECT_MGR_UUID],
[DACL_ERROR_TOO_FEW_BYTES], [DACL_ERROR_TOO_MANY_BYTES],
[DACL_ERROR_DUPLICATE_ENTRY_FOUND],
[DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE],
[DACL_ERROR_ENTRY_TYPE_TOO_LARGE] or
[DACL_ERROR_REQUIRED_ENTRY_MISSING].

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_ParseSyscallAcl( ), *dacl_AreObjectEntriesRequired*( ) and *dacl_FreeAclEntries*( ).

NAME

dacl_InitEpiAcl — initialize a filesystem **ACL**

SYNOPSIS

```
long  dacl_InitEpiAcl(
    dacl_t *     theAclP,
    u_int32 *    modeP,
    int          forDirectory,
    epi_uuid_t * aclRealmP
    );
```

ARGUMENTS

theAclP	The **ACL**.
modeP	The UNIX permission bits.
forDirectory	Flag to specify a directory object **ACL**.
aclRealmP	Default realm of **ACL** (optional).

DESCRIPTION

This function initializes the **ACL** to be a filesystem **ACL** in the local cell. The **ACL** only has 3 simple object entries *user_obj*, *group_obj* and *other_obj*. If the UNIX permission bits argument *modeP* is non-NULL, the simple object **ACL** entries permissions are set to have corresponding access rights, otherwise *user_obj* is set to have only *control* access right and *group_obj* and *other_obj* to have none. If the *forDirectory* flag is set, UNIX *write* permission results in *write*, *insert* and *delete* access being granted in the corresponding **ACL** entry.

If the realm (*aclRealmP*) is not null, it is used as the default realm of the **ACL**. Otherwise, for the kernel, the realm of the **ACL** is set to the local cellID; and for a non-kernel user, the realm is set to zero.

RETURN VALUE

[DACL_ERROR_PARAMETER_ERROR], if a parameter error is found (no **ACL** is passed), which is also logged.

Otherwise, [DACL_SUCCESS] is returned.

ERRORS

None.

SEE ALSO

dacl_ChmodAcl( ).

NAME

dacl_AreObjectEntriesRequired — determine if simple object entries are needed in the **ACL**

SYNOPSIS

```
int  dacl_AreObjectEntriesRequired(
    epi_uuid_t *    mgrUuidP
    );
```

ARGUMENTS

mgrUuidP The UUID for the **ACL** manager.

DESCRIPTION

This function determines if the specified **ACL** requires the *user_obj*, *group_obj* and *other_obj* simple entries.

RETURN VALUE

[1], if the query determines the requirement must be met.

Otherwise, [0] is returned.

ERRORS

None.

NAME

dacl_AreObjectUuidsRequiredOnAccessCheck — determine if object UUIDs are required for authorization check

SYNOPSIS

```
int  dacl_AreObjectUuidsRequiredOnAccessCheck(
    epi_uuid_t *    mgrUuidP
    );
```

ARGUMENTS

mgrUuidP The UUID for the **ACL** manager.

DESCRIPTION

This function determines if the the UUIDs corresponding to *user_obj* and *group_obj* are required during the authorization check procedure.

RETURN VALUE

[1], if the query determines the requirement must be met.

Otherwise, [0] is returned.

ERRORS

None.

NAME

dacl_ArePermBitsRequiredOnAccessCheck — determine if UNIX permission bits are required for authorization check

SYNOPSIS

```
int  dacl_ArePermBitsRequiredOnAccessCheck(
    epi_uuid_t *    mgrUuidP
    );
```

ARGUMENTS

mgrUuidP The UUID for the **ACL** manager.

DESCRIPTION

This function determines if UNIX permission bits are required during the authorization check procedure.

RETURN VALUE

[1], if the query determines the requirement must be met.

Otherwise, [0] is returned.

ERRORS

None.

NAME

dacl_AclMgrName — determine symbolic name of **ACL** manager

SYNOPSIS

```
char *  dacl_AclMgrName(
    epi_uuid_t *    mgrUuidP
    );
```

ARGUMENTS

mgrUuidP The UUID for the **ACL** manager.

DESCRIPTION

This function determines the symbolic name for the **ACL** manager specified by the given **ACL** manager UUID.

RETURN VALUE

The following symbolic names are returned: [epsiode], [bosserver], or [unknown].

ERRORS

None.

NAME

Epi_PrinId_ToUuid — generate an UUID from a principal id

SYNOPSIS

```
void  Epi_PrinId_ToUuid(
    epi_principal_id_t *  epiPrinIdP,
    epi_uuid_t *          epiUuidP
    );
```

ARGUMENTS

epiPrintIdP	The principal id.
epiUuidP	The UUID.

DESCRIPTION

This is a macro that generates an UUID corresponding to the specified principal id.

RETURN VALUE

None.

ERRORS

None.

NAME

Epi_PrinId_FromUuid — generate principal id from an UUID

SYNOPSIS

```
void  Epi_PrinId_FromUuid(
    epi_principal_id_t *  epiPrinIdP,
    epi_uuid_t *          epiUuidP
    );
```

ARGUMENTS

epiPrintIdP	The principal id.
epiUuidP	The UUID.

DESCRIPTION

This is a macro that generates the principal id corresponding to the specified UUID.

RETURN VALUE

None.

ERRORS

None.

NAME

Epi_PrinId_Cmp — compare 2 principal ids or UUIDs

SYNOPSIS

```
int  Epi_PrinId_Cmp(
    (epi_principal_id_t|epi_uuid_t) *    idBlob1P,
    (epi_principal_id_t|epi_uuid_t) *    idBlob2P
    );
```

ARGUMENTS

idBlob1P	The first principal id or UUID.
idBlob2P	The second principal id or UUID.

DESCRIPTION

This is a macro that compares the 2 specified principal ids or the 2 specified UUIDs for equality. It returns zero if they are the same, else it returns a non-zero value.

RETURN VALUE

This routine returns zero if the comparison is the same, otherwise it returns a non-zero value.

ERRORS

None.

NAME

hton_epi_uuid — convert an UUID from host to network byte format

SYNOPSIS

```
void  hton_epi_uuid(
    epi_uuid_t *    epiUuidP
    );
```

ARGUMENTS

epiUuidP The UUID.

DESCRIPTION

This function converts an UUID from the host byte order to the network byte order format.

RETURN VALUE

None.

ERRORS

None.

NAME

ntoh_epi_uuid — convert a UUID from network to host byte format

SYNOPSIS

```
void  ntoh_epi_uuid(
    epi_uuid_t *    epiUuidP
    );
```

ARGUMENTS

epiUuidP The UUID.

DESCRIPTION

This function converts an UUID from the network byte order to the host byte order format.

RETURN VALUE

None.

ERRORS

None.

NAME

hton_epi_principal_id — convert a principal id from host to network byte format

SYNOPSIS

```
void  hton_epi_principal_id(
    epi_principal_id_t *    epiPrinIdP
    );
```

ARGUMENTS

epiPrintIdP The principal id.

DESCRIPTION

This function converts a principal id from host byte order to network byte order format.

RETURN VALUE

None.

ERRORS

None.

NAME

ntoh_epi_principal_id — convert a principal id from network to host byte format

SYNOPSIS

```
void  ntoh_epi_principal_id(
    epi_principal_id_t *    epiPrinIdP
    );
```

ARGUMENTS

epiPrintIdP The principal id.

DESCRIPTION

This function converts a principal id from network byte order to host byte order format.

RETURN VALUE

None.

ERRORS

None.

Appendix A

Mapping DFS ACLs to UNIX Mode Bits

This appendix discusses the relationship between **ACL**s and UNIX mode bits and also the file creation mask and how it relates to **ACL**s. More detailed information on **ACL**s can be found under the topic, *Access Control List Overview*, in Chapter 8 on page 155, relating to the **ACL** types discussed in this appendix.

A.1 Relationship Between ACLs and UNIX Mode Bits

The following summarizes the relationship between **ACL**s and mode bits:

- *owner* mode bits correspond to **ACL** *user_obj*
- *group* mode bits correspond as follows:
  - — *group* mode bits correspond to **ACL** *group_obj* if no *mask_obj* is present
  - — UNIX group mode bits correspond to the *mask_obj* if the **ACL** has a *mask_obj*
- *other* mode bits correspond to **ACL** *other_obj*.

This relationship will be demonstrated in the next set of examples. They start out with a basic set of characterists and then are cumulatively changed. This should be kept in mind as the examples are examined.

A.1.1 ACLs and UNIX Mode Bits

Example A-1 ACLs and UNIX Mode Bits

In this example, file **foo** is owned by rajesh@minyan.dce.transarc.com, **foo**'s owning group is transarc@transarc.dce.com. The prompt has been shortened to consist of the text, `[test]`.

The **acl_edit** command is shown being used with the list option (**-l**) to display the file's **ACL**.

```
[test] pwd
/.../minyan.dce.transarc.com/fs/test
[test] acl_edit foo -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
user_obj:rw-c--
group_obj:r-----
other_obj:r-----
[test] ls -l foo
-rw-r--r--   1 rajesh    transarc       0 Sep 15 11:51 foo
```

A.1.2 Changing other Mode Bits

Example A-2 Changing *other* Mode Bits

This example starts with the results demonstrated in he=1 .ds ;p Example A-1 on page 221as It demonstrates that *other* mode bits correspond to **ACL** *other_obj* according to the relationship stated in Section A.1 on page 221.

```
[test] chmod o+x foo
[test] ls -l foo
-rw-r--r-x   1 rajesh   transarc       0 Sep 15 11:51 foo
[test] acl_edit foo -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
user_obj:rw-c--
group_obj:r-----
other_obj:r-x---
```

Notice *other_obj* now has execute rights.

A.1.3 Changing group Mode Bits on a File with ACL with No mask_obj

Example A-3 Changing *group* Mode Bits on a File with **ACL** and No *mask_obj*

According to the relationship stated in Section A.1 on page 221, this example demonstrates that *group* mode bits correspond to the **ACL** *group_obj* entry since no *mask_obj* is present in the object. The base characteristics of the **ACL** are as they ended in the previous example, Example A-2.

```
[test] chmod g+w foo
[test] ls -l foo
-rw-rw-r-x   1 rajesh   transarc       0 Sep 15 11:51 foo
[test] acl_edit foo -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
user_obj:rw-c--
group_obj:rw----
other_obj:r-x---
```

Notice that *group_obj* now has **w** rights.

A.1.4 Changing the user_obj Entry

Example A-4 Changing the *user_obj* Entry

In this example, *owner* mode bits will exhibit the changed *user_obj*, as they correspond to the **ACL** *user_obj* according to Section A.1 on page 221.

```
[test] acl_edit foo -m user_obj:rwxc
[test] acl_edit foo -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
user_obj:rwxc--
group_obj:rw----
other_obj:r-x---
[test] ls -l foo
```

```
-rwxrw-r-x   1 rajesh    transarc       0 Sep 15 11:51 foo
   ^
   ^
```

Notice *owner* (mode bits) show **x** rights in the **ls -l** output.

A.1.5 Changing the group_obj Entry

Example A-5 Changing the *group_obj* Entry

In this example, since the **ACL** has no *mask_obj*, the change will affect the *group* mode bits. The starting setup for this example is from the ending of Example A-4 on page 222.

```
[test] acl_edit foo -m group_obj:rwx
[test] acl_edit foo -l
ls -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
user_obj:rwxc--
group_obj:rwx---
other_obj:r-x---
[test] ls -l foo
-rwxrwxr-x   1 rajesh transarc       0 Sep 15 11:51 foo
```

The *group_obj* entry change impacts *group* mode bits.

A.1.6 Changing the group_obj Entry in an ACL with a mask_obj Entry

Example A-6 Changing the *group_obj* Entry in an ACL with a *mask_obj* Entry

At this point, we introduce a *mask_obj* entry for the **ACL**. Now lets look at what happens if the **ACL** has an *mask_obj* entry.

Note: A *mask_obj* entry in an **ACL** is required if it has entries other than *user_obj*, *group_obj* and *other_obj*.

According to the relationship stated in Section A.1 on page 221, we should expect that a change to the *group_obj* entry in the **ACL** object will not affect the UNIX *group* mode bits.

```
[test] acl_edit foo -m user:ashok:r
[test] acl_edit foo -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
mask_obj:rwx---
user_obj:rwxc--
user:ashok:r-----
group_obj:rwx---
other_obj:r-x---
[test] ls -l foo
-rwxrwxr-x   1 rajesh    transarc       0 Sep 15 11:51 foo
```

A.1.7 The Results of chmod of the group Bits

Example A-7 The Results of **chmod** of the *group* Bits

Starting with where we left off in the previous example, (Example A-6 on page 223,) the following scenario demonstrates the results of **chmod** of the *group* bits. According to the relationship established in Section A.1 on page 221, we should expect to see a corresponding change to the *mask_obj* in the **ACL**, and no effect upon the *group_object* entry.

```
[test] chmod g-x foo
[test] ls -l foo
-rwxrw-r-x   1 rajesh transarc       0 Sep 15 11:51 foo
[test] acl_edit foo -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
mask_obj:rw----
user_obj:rwxc--
user:ashok:r-----
group_obj:rwx---        #effective:rw----
other_obj:r-x---
```

Notice that changing the group mode bits affected the *mask_obj* entry and not the *group_obj* entry.

A.1.8 The Results of acl_edit of the mask_obj Bits

Example A-8 The Results of **acl_edit** of the *mask_obj* Bits

This example starts with the setup from the previous example, Example A-8, as the base. The following scenario shows the result of modification of the *mask_obj* entry.

According to Section A.1 on page 221, modifying the *mask_obj* entry in the **ACL** object directly affects the UNIX *group* mode bits:

```
[test] acl_edit foo -m mask_obj:r
[test] acl_edit foo -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
mask_obj:r-----
user_obj:rwxc--
user:ashok:r-----
group_obj:rwx---        #effective:r-----
other_obj:r-x---
[test] ls -l foo
-rwxr--r-x   1 rajesh    transarc       0 Sep 15 11:51 foo
```

Notice that changing the *mask_obj* to **r** changed the group bits shown by **ls** to **r**.

A.1.9 Changing the group_obj Entry for an ACL with a mask_obj

Example A-9 Changing the *group_obj* Entry for an ACL with a *mask_obj*

According to Section A.1 on page 221, for an **ACL** object with a *mask_obj*, changing the *group_obj* entry for an **ACL** with *mask_obj* should have no impact on the *group* mode bits. The following scenario demonstrates this.

This example's setup starts with the results of the previous example, Example A-8 on page 224.

```
[test] acl_edit foo -m group_obj:- -l
# SEC_ACL for foo:
# Default cell = /.../minyan.dce.transarc.com
mask_obj:r-----
user_obj:rwxc--
user:ashok:r-----
group_obj:------
other_obj:r-x---
[test] ls -l foo
-rwxr--r-x   1 rajesh transarc       0 Sep 15 11:51 foo
```

Notice that there is no change in *group* mode bits.

A.2 File Creation Mask and ACLs

The following rules apply to the file creation mask. The file creation mask refers to the **umask**. In this appendix, an **Initial Object ACL** will be referred to as an **IO ACL**.

A. In DFS if a directory has an appropriate **Initial Object ACL**, then the *umask* setting is not taken into account when creating a file in that directory. Only the mode bits specified to the **creat** syscall and the **IO ACL** determine the resulting **ACL** on the file.

B. If the directory does not have an **IO ACL**, then the *umask* setting of the process creating the file is taken into account. In this case the mode bits on the resulting file are determined using the mode bits specified to **creat** syscall and the *umask*.

A.2.1 File Creation with no Initial Object ACL

Example A-10 File Creation with no **IO ACL**

Let's say we are in a directory with no **IO ACL** called **simple_dir** and we create a file using **touch**. In this example, **touch** specifies **0666** creation mode bits to the **creat** syscall.

First, let's set a *umask*, and then display it.

```
[simple_dir] umask 77
[simple_dir] umask
77
```

Now, let's *touch* the file, **foo** (and display it):

```
[simple_dir] touch foo
[simple_dir] ls -l foo
-rw-------   1 rajesh   transarc       0 Sep 15 12:08 foo
```

touch **foo** specified **creat(***foo, 0666***)**. DFS determined that the parent directory did not have an **IO ACL**, so case B applies.

The algorithm to determine the mode bits on the resulting file is the same as in traditional UNIX systems (as in POSIX.1):

```
resulting file mode bits = syscall creation mode bits & ~umask =
```

Thus, in this case:

```
resulting file mode bits = 0666 & ~077
                         = 0600
                         = owner:rw group:- other:-
```

as shown in the **ls -l** output.

A.2.2 File Creation with an Initial Object ACL

Example A-11 File Creation with an **IO ACL**

Let's say we are in a directory with an **IO ACL** called **acl_dir**. Case A should apply here on both file and directory creation. First, let's display the **Initial Object ACL**. This is done by using **acl_edit** with the **-io -l** options:

```
[acl_dir] acl_edit . -io -l
# Initial SEC_ACL for objects created under: .:
# Default cell = /.../minyan.dce.transarc.com
```

```
mask_obj:rwx---
user_obj:rwxc--
user:ashok:rwx---
group_obj:rw----
other_obj:r-----
```

Let's set the **umask** and display it, as in the previous example:

```
[acl_dir] umask 77
[acl_dir] umask
77
```

Now, let's *touch* the file, **bar** (and display it):

```
[acl_dir] touch bar
[acl_dir] ls -l bar
-rw-rw-r--   1 rajesh   transarc       0 Sep 15 15:21 bar
```

Notice that the permissions **664** on the new file,**bar**, mean that the *umask* was not applied, else the *group* and *other* permissions would be zero.

A.2.3 File ACL Creation with an initial object ACL

Example A-12 File **ACL** Creation with an **initial object ACL**

Let's see what the **ACL** for **bar** is. This can be done by using **acl_edit** with the **-l** option, as follows:

```
[acl_dir] acl_edit bar -l
# SEC_ACL for bar:
# Default cell = /.../minyan.dce.transarc.com
mask_obj:rw----
user_obj:rw-c--
user:ashok:rwx---       #effective:rw----
group_obj:rw----
other_obj:r-----
```

Under this case (A), the permissions on the new file's **ACL** are derived by intersecting the parent directory's **IO ACL** permissions with the mode bits specified to the **creat** syscall, as follows.

Note: The parent directory's **IO ACL** permissions are those from the **ACL** shown in Example A-11 on page 226.

Also, recall that *touch* uses **creat** mode bits as **0666**. Thus, the respective *owner*, *group*, and *other* creation mode bits are **rw**.

```
user_obj perms = parent dir IO ACL user_obj perm &
                 owner creation mode bits
```

Thus, in this case:

```
user_obj perms   = rwx & rw
                 = rw.
```

Also *user_obj* is always granted **c** rights.

Similarly for *other_obj*, intersect **IO ACL** *other_obj* permissions with *other* **creat** mode bits:

```
other_obj perms = parent dir IO ACL other_obj perm &
                  other creation mode bits
```

Thus:

```
other_obj perms  = r & rw = r
```

Recall from Section A.1 on page 221, that since the **IO ACL** has a *mask_obj*, the *IO mask_obj* will be intersected with the group **creat** mode bits to determine the *mask_obj* on the file **ACL** as follows:

```
mask_obj perms = parent dir IO ACL mask_obj perm &
                 group creation mode bits

mask_obj perms  = rwx & rw = rw.
```

Note: If the **IO ACL** did not have *mask_obj*, then the group **creat** mode bits would be intersected with the IO ACL group_obj permissions to determine the file ACL's group_obj permissions.

Appendix B

Access Control List Package Error List

The following sections contain the return values that are returned from the *Access Control List Interface Functions* found in Chapter 10 on page 171.

B.1 Access Control List Return Values

[DACL_SUCCESS]
The operation was successful or no errors were found.

[DACL_ERROR_TOO_MANY_BYTES]
Too many bytes in acl byte buffer.

[DACL_ERROR_TOO_FEW_BYTES]
Too few bytes in acl byte buffer.

[DACL_ERROR_ENTRY_NOT_FOUND]
ACL entry not found.

[DACL_ERROR_ENTRY_EXISTS]
ACL entry already exists.

[DACL_ERROR_UNRECOGNIZED_ENTRY_TYPE]
Unrecognized entry type.

[DACL_ERROR_UNRECOGNIZED_ENTRY_CLASS]
Unrecognized entry class.

[DACL_ERROR_UNIMPLEMENTED_ENTRY_TYPE]
Unimplemented entry type.

[DACL_ERROR_SEC_RGY_PGO_ERROR]
Error returned from security service.

[DACL_ERROR_MISSING_NAME]
Name required for specified entry type.

[DACL_ERROR_ENTRY_REQUIRED]
Attempt to remove required ACL entry.

[DACL_ERROR_BUFFER_ALLOCATION]
Buffer allocation error.

[DACL_ERROR_ACCESS_DENIED]
Requested access (implicitly) denied by ACL.

[DACL_ERROR_ACCESS_EXPLICITLY_DENIED]
Requested access explicitly denied by ACL.

[DACL_ERROR_ACL_FILE_EXISTS]
File in which ACL creation was requested already exists.

[DACL_ERROR_INCORRECT_MGR_UUID]
Attempt to parse ACL by incorrect ACL manager.

[DACL_ERROR_ILLEGAL_ENTRY]
ACL contains an entry not appropriate for the ACL manager type.

[DACL_ERROR_MGR_PARAMETER_ERROR]
ACL manager requires parameters not passed to routine.

[DACL_ERROR_REQUIRED_ENTRY_MISSING]
Required ACL entry missing from ACL.

[DACL_ERROR_PARAMETER_ERROR]
Required pointer parameter has NULL value.

[DACL_ERROR_UNRECOGNIZED_MGR_TYPE]
Unrecognized ACL manager type uuid.

[DACL_ERROR_ENTRY_TYPE_TOO_LARGE]
ACL entry type is too large to be processed by current code.

[DACL_ERROR_DUPLICATE_ENTRY_FOUND]
Duplicate ACL entry found.

[DACL_ERROR_ACLFILE_NOT_FOUND]
Specified ACL file not found.

[DACL_ERROR_VALIDATION_FAILURE]
Flat ACL buffer is of incorrect form.

[DACL_ERROR_UNRECOGNIZED_USER_OR_GROUP]
User or group not recognized by Registry Server.

[DACL_ERROR_NONLOCAL_ENTRY_TYPE]
Non-local entry type encountered running in local mode.

B.2 Filesystem Access Control List Return Values

The following are filesystem return values.

[DACL_ERROR_FS_OPEN]
File system open error.

[DACL_ERROR_FS_CLOSE]
File system close error.

[DACL_ERROR_FS_STAT]
File system stat error.

[DACL_ERROR_FS_READ]
File system read error.

[DACL_ERROR_FS_WRITE]
File system write error.

X/Open Preliminary Specification

Part 6:

The DCE DFS VFS+ Interface Specification

This part of the **DCE DFS** provides an overview of the VFS extensions provided by DFS, called the *DCE DFS VFS+ Interface* . This interface is an enhancement of the **DCE 1.1: Distributed File Service** specification that permits different file systems to coexist within one UNIX kernel.

Chapter 11

DCE DFS VFS+ Interface Introduction

This chapter provides an overview of the VFS+ interfaces and behaviors that a DFS-compliant physical file system needs to support. Such a file system will be fully interoperable with other implementations of **DCE DFS** file systems from the viewpoint of all network protocols and the vast majority of user-space software for extended filesystem fileset operations, known as **EFS** software.

11.1 Definition of Terms

DFS The DCE Distributed File System.

EFS The optional portion of DFS which deals with extended fileset operations (cloning, backup, replicas, and so on).

LFS Local File System: a physical file system, which in the context of this document, provides DFS semantics.

Episode is the name of the LFS that Transarc Corporation provides as part of the DFS EFS package.

UFS The **UNIX** File System. The name of the file system originally provided by vendors of **UNIX** systems.

Aggregate An expanse of physical disk which is managed by a DFS LFS. It is similar in context to a UNIX partition and is identified by both a device number and a device file. In many cases, the storage provided to an aggregate will be by a Logical Volume Manager (LVM).

Cell A collection of DCE machines administered as a single entity. In this context, the key point is that a cell is serviced by a single registry with a single, consistent set of user and group identifiers.

Fileset A set of related files, connected via a sub-tree with a single root point, that is administered as an entity by DFS. Although filesets cannot span aggregates, a single aggregate can hold many filesets. Filesets which have an associated maximum size or quota, grow independently of each other and compete for space within their aggregate. Individual filesets can be mounted locally, backed up and restored, cloned or transparently moved to another aggregate.

The terms fileset and volume are used interchangeably, although the term volume is becoming archaic.

Group A membership list, identified by a **uuid_t**, which contains a list of principals (by **uuid_t**) which are members of that group. Again, in a non-DCE environment, they are identified by a 32-bit (at best) **gid**.

PAC Privilege Attribute Certificate. A data structure, generated by the DCE Privilege Server, that contains the identity of a DCE authenticated principal. To a first approximation, it contains the principal's **uuid_t** along with the **uuid_t** of each group of which it is a member.

Principal An entity that is interacting with DFS. Although it usually refers to a person, it could as easily correspond to a DCE server or a computer within a DCE cell. In the

DCE environment, principals are identified by a **uuid_t**. In a non-DCE, standard UNIx, environment, they are identified by a 32- bit (at best) **uid**.

Realm Equivalent to a Cell. In security DCE security discussions, the term Realm is typically used instead of Cell.

11.2 VFS+ Interface Goals and Constraints

The **DCE DFS** VFS+ interface is an enhancement of the VFS interface. The VFS (Virtual File System) interface was introduced by Sun Microsystems to make it possible to implement different file systems within one **UNIX** kernel; specifically, to implement both a physical and a network file system. The original VFS interface, or variants of it used by other vendors of **UNIX** systems, appears in most **UNIX** kernels today. The VFS+ interface extends VFS in several ways, serving to generalize VFS and adapt it to fulfill the requirements of operation in a distributed computing environment.

The principal improvements offered by the VFS+ interface are:

- **Generalized credentials**. The credential structure passed to most of the VFS functions has been augmented to accommodate the use of various different authentication mechanisms. The new credential may thus carry the standard **UNIX** information as well as Kerberos tickets, or any other form of authentication information required. This revision is completely backwards-compatible.
- **Synchronization**. Most operations are redefined to use a synchronization package. In this way, protocol exporters can support notification guarantees to their clients (similar to AFS *callbacks*, for those familiar with the concept). When an object in any type of virtual file system is changed, the synchronization system notifies all interested parties. This allows a more general mechanism for providing appropriate semantics in a distributed file system.
- **Fileset/aggregate interface**. The VFS+ interface includes operations on **filesets**, a key functional enhancement and operability feature in the **DCE DFS** distributed environment. Filesets are located within a new form of data container called an **aggregate**, and the VFS+ interface also allows them to be manipulated.

  Historically, filesets were called **volumes**, but the name was changed to avoid confusion with features of other file systems.
- **Portability**. Several portability problems are addressed. Synchronization is equally effective for vendor file systems (such as the **UFS** file system in Sun's SunOS and other Berkeley-derived kernels) and for local file systems (such as the **DCE LFS** that is part of **DCE DFS**). Vendor file systems need not be enhanced to support the full functionality of filesets and aggregates, but they can support the full fileset/aggregate interface, with rudimentary functionality. And lastly, local file systems can be adapted to different kernels. It is necessary to do this without modifying the kernels above the VFS layer. A complication is that some vendor kernels, notably IBM's AIX and DEC's Ultrix, do not have simple variants of Sun's VFS interface, but have their own file system interfaces, with differing data structures (both AIX and Ultrix have a **gnode** structure, and Ultrix does not have a **vnode** structure), and different designs for the boundary between the file system and the higher levels of the kernel. Another complication is that the file system is expected to be integrated with the virtual memory system, and the different kernels have completely dissimilar and independent virtual memory systems.

An important motivation for these changes is compatibility among protocol exporters. For example, one protocol exporter (namely, a VFS+ server) may make guarantees about when an

exported file changes. At the same time, other servers (in particular, NFS), as well as the local **UNIX** kernel, may simultaneously modify these files.

In order to avoid modifying the file-system-independent layer of the **UNIX** kernel or the NFS server, the stock VFS vnode-level functions are redefined to call on a synchronization package that keeps track of all guarantees made by the various protocol exporter types. The updated function definitions are implemented by writing what is termed ''wrapper'' functions for an already-existing virtual file system. These wrappers perform the appropriate synchronization calls before and after the original VFS call.

The VFS+ performance issues are fairly straightforward. The principal additional overheads incurred by using the VFS+ interface are checks for concurrent whole-fileset and checks for conflicting references to vnodes. The check for concurrent use of a fileset is done by lookup in a *fileset registry* (see Section 11.6 on page 239). Checks for conflicting conflicting references to vnodes are done by calling a file-system-independent layer with the appropriate file ID.

This layer is necessary to ensure that no matter how a vnode is modified or otherwise accessed, all protocol exporters that may have granted incompatible promises to their clients will have an opportunity to revoke those promises. Promises are represented by *tokens*. The VFS+ interface must obtain *tokens* for files that it manipulates; in this respect it is no different from a DCE DFS client.

The *fileset registry* and the *token cache* are hash tables in which normal lookups, using the file ID, are relatively inexpensive. Optimizations allow the fileset synchronization and token obtaining steps to be skipped, wholly or in part, for those filesets that are not exported, or for filesets that are read-only. Thus, those file systems that may require inter-machine synchronization are the only ones that experience this modest performance penalty.

11.3 Overview of Interfaces to the LFS

There are three separate subdomains or areas in the VFS+ interface that are required. A fourth may be present:

1. **Vnode-level facilities**. The vnode functions appropriate to the particular kernel, accessible via a **struct vnodeops** function array or equivalent, are exported. Each type of LFS provides a vector of pointers to its vnode facilities, and a pointer to this function array is placed in vnodes initialized by that LFS. Thus, for vendor file systems, the array will generally not point to the original vendor functions, but will point perhaps to wrapper functions, which perform synchronization. The wrappers will need to call non-wrapped functions (which in the case of vendor file systems could be the original vendor functions). They will access the non-wrapped versions via the same function array, which therefore is extended to about twice the original length.

   That, however, isn't the only extension of the **struct vnodeops** function array. The methodology for achieving portability in **DCE LFS** has been to write a set of functions appropriate for Sun's original VFS interface, and to use wrapper functions to extend these for VFS-variant interfaces from other vendors (as well as for Sun's more recent VFS extensions). Another layer of wrapper functions means another extension of the function array, which is consequently about three times the original length.

   Of course, the kernel accesses this new triple-length array as if it were just a **struct vnodeops**. But components of the server may access other parts of the array. In particular, the Protocol Exporter accesses the non-synchronized non-portable functions. That is because it does its own synchronization, and it does not have to use the same VFS-variant as the kernel.

2. **Aggregate-level facilities**. An *aggregate* is a data container that holds filesets. Aggregates are a generalization of the **UNIX** concept of partition, while filesets are a generalization of the **UNIX** concept of a file system. That is, a fileset provides a single connected file hierarchy, while an aggregate provides a flat address space on disk. In conventional **UNIX** systems these concepts are identified so that each partition can contain exactly one file system, but in the DFS, an aggregate can contain any number of filesets. The VFS+ interface provides facilities for managing aggregates, including creating new filesets within the aggregate and enumerating existing filesets. Aggregate information is made available via a global *Aggregate Registry* table, where each entry describes the data container for filesets and exposes a function array containing the aggregate operations described above. This table is reached by means of an aggregate type, which is a small integer that is provided by the protocol exporter, that is used as an index to this table. (see Chapter 14 on page 303).
3. **Fileset-level facilities**. Structures and operations are provided to access and manage the contents of individual filesets situated within the above aggregates. Each type of LFS provides a vector of pointers to its vnode operations. A pointer to this vector is returned to DFS by the aggregate operation which attaches filesets. The global *Fileset Registry* table exports entries for each fileset hosted by available aggregates. These entries include a function array (vector of pointers) allowing use of the intra-fileset operations, such as creating, deleting, and getting status on files within a fileset (see Chapter 15 on page 325).
4. **LFS-specific facilities**. A set of facilities that are private or specific to a particular LFS may be present that are used to communicate between LFS-specific utilities and the LFS. These may consist of private system calls or other private routines which are perculiar to a specific vendor's LFS. Nothing else will be said about them other than as noted here - that they are not specifically excluded by this specification.

11.3.1 Locking

One of the most important VFS+ aspects is the manipulation of locks above the original VFS interface layer.

In the VFS+ interface, each VFS function is redefined to perform the following operations:

1. **Lock operands and declare the operations to be performed**. In this step, the VFS function calls a locking function common to all virtual file system types. Declarations are made of the file IDs that are about to be processed, as well as the operations that will be performed upon them. As part of this function, other virtual file systems may be invoked to revoke promises (such as *callbacks* and *tokens*) that they may have issued pertaining to the same files.
2. **Perform the desired original virtual file system operation**.
3. **Release the synchronization entries locked in the first step above**. This does not cause the revocation of any guarantees made by the call itself, but rather indicates to the local synchronization package that revocation is now permitted.

11.3.2 Credentials

The *credential structure* (**struct ucred**) passed to most VFS functions has also been generalized in the VFS+ interface. Each credential is augmented with a *magic cookie* value that is associated with a property list. This property list (and thus the credential containing the appropriate cookie) is then associated with an arbitrary set of user identification information. Examples of entries on this property list are NFS and Kerberos identity structures.

Using this generalized structure, file systems that understand the credential *magic cookie* will obtain the specific information (such as the user's Kerberos identity) that they need to use the relevant protection mechanisms.

File systems that have not been modified to understand the credential *magic cookie* will continue to use the basic **UNIX** uid and group information in order to make authorization decisions, oblivious of the extra information available.

11.3.3 Operations on Filesets and Aggregates

The final components of the new VFS system are two new virtual interfaces for performing operations on filesets and on aggregates. The functions in the fileset interface implement such operations as:

1. **Taking filesets off-line and putting them back on-line**. While a fileset is off-line, vnode operations on files stored in that fileset will either wait or fail with distinctive error codes, depending on the mode in which the fileset was taken off-line. When the fileset operations in question are completed, that fileset is once again functional, and can be accessed via protocol exporters or via system calls from the local **UNIX** kernel.

2. **Iterating through all files in a fileset**. Many operations require that some operation be performed on every file within that fileset. Examples are cloning a fileset and restoring a fileset from a dump.

3. **Operating on individual files in a fileset**. Cloning of a fileset makes use of an operation that clones an individual file. Restoring a fileset makes use of several operations on individual files: one to write the file's data; one to write its status; possibly one to create a file, and possibly one to delete a file; and others.

   The operations used to support dumping and restoring of filesets correspond roughly to various vnode operations used by the **UNIX** kernel. But there are important differences. The vnode operations implement **UNIX** semantics, while the fileset operations do not. For instance, the vnode operation **VOP_RDWR**, which reads or writes file data, will also set the file's atime (for reading) or mtime and ctime (for writing). The corresponding fileset operations **VOL_READ** and **VOL_WRITE**, on the other hand, are carefully coded to avoid modifying *any* of the file's meta-data, except its length. Another important difference is that, given a consistent file system on entry, a vnode operation will restore the file system to consistency at exit, while the volume operations do not maintain consistency. For instance, the above mentioned **VOL_WRITE** operation is used to restore directories as well as files. When it finishes writing a directory, the directory may well contain dangling references to files, which will be corrected only by subsequent volume operations to create those files. For comparison, vnode operations that modify directories, such as **VOP_CREATE** and **VOP_REMOVE**, also create or remove files in such a way that the directory hierarchy is correct by the time the operation returns.

4. **Accessing files on behalf of protocol exporters**. In normal operation, a DFS file exporter obtains a vnode for a file by passing the file's ID and uniquifier (generation number) to a fileset operation.

While a fileset is off-line, an operation on a vnode within that fileset should cause the corresponding virtual file system to either return a distinctive error code or simply wait for the fileset to come back on line.

The functions in the aggregate interface implement such operations as creating a new fileset on an aggregate and enumerating all filesets within an aggregate.

11.4 Organization of the VFS+ Switch

Examples of the following description can be found in Section 13.5 on page 290. They can be referred to for further information to gain an enhanced understanding of this description.

11.4.1 The O-*ops*

In the VFS+ interface, the *base* procedures (the ones already defined by the vendor-supplied kernel) are implicitly redefined. In addition to performing their basic functionality, they also call the DFS Token Manager in order to revoke any tokens conflicting with the operation the VFS function is about to perform. The token element will remain locked in the token database until the VFS operation completes.

The redefined procedures, as used by the vendor-supplied kernel, are referred to as the *O-ops*.

11.4.2 The N-*ops*

An additional set of procedures are provided, which represent the *extended* portion of the interface. They are similar to the base set, but assume the caller has performed all the required token synchronization. Since the extended procedures are new to the vnode operations vector, the only programs that will invoke them are components of DFS, such as the *O-ops*. Other clients of the VFS+ interface, such as the **UNIX** kernel's file system calls, do not know of the existence of these extended functions.

These procedures, having the same specifications as the *O-ops* but not performing synchronization, are referred to as the *N-ops*. If the file system is a vendor-supplied file system, these are generally the original vendor vnode ops.

11.4.3 The X-*ops*

The last extension is a set of functions that do not conform to the vendor's VFS-variant specifications, but rather, conform to the original VFS specifications. These are referred to as the *X-ops*. Because the *X-op* interface is constant across platforms, it is convenient for use by the Protocol Exporter and other server components. For portability, the DCE LFS has been written as a set of *X-ops*, and the corresponding *N-ops* are wrappers for these *X-ops*. Vendor-supplied file systems, on the other hand, come as sets of *N-ops*, and the corresponding base *X-ops* are wrappers for the *N-ops*.

The *X-ops* include some extra operations, besides those in Sun's original VFS. Some of these extra operations facilitate integration with vendor virtual memory systems. Others facilitate communication with the Protocol Exporter, or the Fileset Registry, or allow manipulation of Access Control Lists. A complete set of specifications for the *X-ops* may be found in Section 16.7 on page 424 and Section 16.8 on page 460.

11.5 Basic Operation

Because of the extension of the vnode operation vector, there are some basic bootstrapping problems in getting a VFS+ interface working in a kernel. It cannot be assumed that any file system's code is written with any knowledge of the VFS+ interface, or even that the VFS+ interface is present when the kernel is first booted or at any time soon after boot. In other words, the VFS+ interface may be loaded as a kernel extension, after a system has been running for some time.

Pathologically, some kernels may rely upon knowledge of the addresses of vnode operation vectors. For instance, a kernel may determine whether a vnode will require certain resources, by comparing the address of its vnode op vector with the known address of the vnode op vector for a particular file system. This sort of comparison will not work if old vnode ops vectors are replaced by new ones. Thus, such file systems cannot be exported. Appendix C on page 245 discusses typical components in a typical VFS+ package in a typical VFS kernel.

11.6 The Fileset Registry

In-memory structures describing filesets local to the host are kept in a hash table called the *Fileset Registry*. The **struct volume** fileset descriptor records kept there (see are hashed by volume ID. One fileset descriptor exists for each fileset served by that host. There are various ways to generate references to these fileset descriptor structures. Section 11.7 shows how to determine the proper fileset pointer given an already existing vnode. Section 15.14 on page 375 describes the *vol_vget*( ) call, which converts a file ID to a vnode containing the proper fileset descriptor reference.

Protocol exporters will need to have the *Fileset Registry* pre-loaded by an application program which iterates over the filesets on the server's disk(s) and loads fileset descriptors for each.

Each fileset descriptor stored in the *Fileset Registry* contains a pointer to an array of operations (**struct volumeops**) that may be performed on the associated fileset. Thus, given a fileset ID, acquiring the *Fileset Registry* entry for that fileset allows the caller to perform such operations as attach, destroy, and open, as well as operations on the individual files contained within that fileset. The **struct volumeops** operation array is used by such agents as the Fileset Server to perform higher-level activities such as cloning (replicating), moving, and dumping filesets. Refer to Chapter 15 on page 325 (in particular, the overview in Section 15.1 on page 325) for a more detailed examination of the fileset portion of the VFS+ interface.

11.7 The Fileset/Vnode Interfaces

These sections describe the interface between the fileset and vnode layers in more detail. They primarily define how, given a vnode, one finds the corresponding fileset.

A new VFS+ function, **VOPX_GETVOLUMEa**, is defined that takes a vnode pointer and returns the corresponding fileset pointer.

The UFS implementation of this function looks up the VFS pointer of the vnode in a hash table. The table associates VFS pointers with *fileset registry* entry pointers. In a file system for which exportation would make no sense, such as NFS, the implementation of this function would simply return a NULL pointer, indicating that the fileset concept does not apply to this file system type.

In **DCE LFS**, this function obtains a handle for the fileset from the file-system-specific portion of the vnode structure. From this handle, it obtains the fileset's ID, and looks up that ID in the *fileset*

registry in the normal way.

11.8 References from the LFS back into DFS

There is a set of core functionality that is shared between **DCE DFS** (herein called DFS) and an **DCE LFS**, herein called an LFS. These are provided by a set of DFS functions which LFS calls. This section will list this functionality. In the interest of minimizing the interactions between DFS and an LFS, only those calls which are truly needed (or seem potentially useful) and cannot be implemented by code within the LFS itself are actually listed.

11.8.1 LFS Introducing Itself to DFS

An LFS makes itself known to DFS via an *ag_setops*( ) call which supplies DFS with an aggregate operations vector to be used for the given aggregate type (AG_TYPE_xxx). These are listed in Section 13.2.4 on page 270 and are used in several places such as Section 13.2.1 on page 269.

Note: In Section 13.2.4 on page 270, *Valid Aggregate Types*, there is also a size definition. If one were to add a new type (or types), one would also increase the size of the aggregate operations structures by increasing this size appropriately.

Additionally, the aggregate operations vector whose structure is defined in Section 13.2.6 on page 270, *Aggregate Operations Vector*, is supplied. The following signature defines the *ag_setops*( ) function call.

```
/* agtype:   AG_TYPE_xxx encoding for the LFS
 * ops:      supplied aggregate operations vector
 */
extern void ag_setops(
    /* IN */ int    agtype,
    /* IN */ struct aggrops    *ops
);
```

11.8.2 Operating on DFS Lock Structures

The next set of functions are used by the *vol_lock*( ), *vol_unlock*( ), *ag_lock*( ) and *ag_unlock*( ) operations to operate on DFS lock structures.

The following locks are defined in the DFS header file **src/file/osi/lock.h** .

```
#define READ_LOCK       1
#define WRITE_LOCK      2
#define SHARED_LOCK     4
```

The following lock structure is also defined in this same header. It is the standard "data lock". All locks wait on excl_locked except for READ_LOCK, which waits on readers_reading

```
struct lock_data {
    unsigned char wait_states;     /* type of lockers waiting */
    unsigned char excl_locked;     /* boosted, shared or write lock? */
    unsigned char readers_reading; /* readers actually with read locks */
    unsigned char num_waiting;     /* probably need this soon */
};
```

The following locking functions are defined in this header as well. As previously mentioned, these functions are used to operate on the DFS lock structure, **lock_data**, defined above.

```
extern lock_ObtainRead(lock_data);
extern lock_ObtainWrite(lock_data);
extern lock_ObtainShared(lock_data);
extern lock_ReleaseRead(lock_data);
extern lock_ReleaseWrite(lock_data);
extern lock_ReleaseShared(lock_data);
```

11.8.3 Allocating and Freeing Memory

The next two calls are used to allocate and free kernel memory. They are specifically needed by the *ag_attach*() and *ag_detach*() functions. They are defined in the DFS header file **src/file/osi/osi.h** . See also Appendix F on page 323 for additional information.

```
/*
 * typedef needed for the Alloc and Free functions
 */

   typedef unsigned int size_t;

/*
 * Generic allocation routine
 * Returns pointer to allocated storage, else panics ;
 * CAN block
 *
 * asize:    allocation size wanted, in bytes
 */
   extern opaque osi_Alloc(
       size_t    asize
       );

/*
 * Generic memory deallocation routine
 *
 * x:  pointer to whatever is being freed
 * asize:    ... and its size (in bytes)
 */
   extern void osi_Free(
       opaque     x,
       size_t    asize
       );
```

11.8.4 Fileset Creation Assistance

The following call is needed by the *ag_volCreate*() operation. Its use is described in the chapter,*Aggregate Operations Interface*, under that function (see Chapter 14 on page 303). There is also more information in Appendix D on page 317. It is defined in the DFS header file **src/file/xvolume/vol_init.h** .

```
/* Returns 0 on success, else an error code */

   extern int vol_Attach(
       afsHyper            *volId,
       struct vol_status   *statusp,
       struct aggr         *aggrp,
```

```
        struct volumeops    *volopsp
        );
```

11.8.5 Obtaining a Fileset Structure

The following call is used by several **fileset** and **vnode** operations to determine if a given fileset (by ID) is exported to DFS and, if so, to obtain a held fileset (volume) structure for it. It is defined in the DFS header file **src/file/volreg/volreg.h** . It's signature can be found in Section 15.15 on page 399 in *Fileset Registry Array Functions*.

```
/* Returns 0 on success, else an error code
 *
 * fidp:    fileset ID
 * volp:    if non-NULL, pointer to held volume
 *          structure is returned here
 */
   extern int volreg_Lookup(
       struct afsFid *fidp,
       struct volume **volp
       );
```

11.8.6 Releasing a Fileset

The following call is made by the *vol_rele*( ) call in Section 15.14 on page 340 when the usage count on a volume structure goes to zero.

It is defined in the DFS header file **src/file/xvolume/vol_init.h** .

```
/* Always returns 0 (success)
 *
 * volp:    pointer to volume structure whose usage
 *          count has gone to 0.
 */
   extern int vol_VolInactive(
       struct volume *volp
       );
```

11.8.7 Conversion of Operations Vectors

The following calls assist in the "conversion" of **vnode** and **vfs** operations vectors. Consult the portion of this document on **vnode** operations, Chapter 16 on page 415.

These don't appear to exist in any DFS header files. Their source resides in the DFS file **src/xvnode/xvfs_vnode.c** .

```
/* afuns:    pointer to LFS extended vnode ops vector
 * axfuns:   pointer to enhanced (3-part) ops vector
 *           that is constructed by this call
 */
   extern void xvfs_InitFromXOps(
       struct xvfs_xops     *afuns,
       struct xvfs_vnodeops *axfuns
       );

/* aofuns:   pointer to LFS vfs operations vector
 * afuns:    pointer to enhanced (2-part) vfs vector
```

```
 *           that is constructed by this call
 * getvolfn: pointer to supplied vfs_getvolume
 *           function
 */
   extern void xvfs_InitFromVFSOps(
       struct osi_vfsops    *aofuns,
       struct xvfs_vfsops   *afuns,
       int                 (*getvolfn)()
       );
```

11.8.8 Getting the Local Cell ID

The following call returns the DCE local Cell ID (consult Chapter 12 on page 251 for details on its use).

```
extern dacl_GetLocalCellID(
    afsUUID       *localCellID
    );
```

11.8.9 Getting the Administrative Group ID

The following call returns the uuid of the DFS administrative group. It is the general function provided for determining if the caller is the member of the DFS administrative group.

```
extern dacl_GetSysAdminGroupID(
    afsUUID       *groupID
    );
```

11.8.10 Obtaining the Identity of the Principal

As described in the *Obtaining a Principal's Identity* section of the ACL document (see Chapter 12 on page 251), the following calls might be necessary.

Note: As part of its support, a vendor could instead provide a wrapper function within DFS that performs the full task of extracting a PAC from a set of credentials. Their signatures can be found in Section 17.1 on page 469, *The xcred Package*, in Section 17.3 on page 477, and Section 17.3 on page 480, respectively.

```
/* aucredP:      input unix credentials
 * axcredPP:     pointer to returned extended credentials
 */
   extern xcred_UCredToXCred(
       struct ucred  *aucredP,
       cred_t        **axcredPP
       );

/* axcredP:           input extended credentials
 * aattributeP:       desired attribute
 * aattributeLength:  ... and its length
 * avaluePP:          returned attribute value
 * areaLengthPP       ... and its length
 */
   extern xcred_GetProp(
       xcred_t       *axcredP,
       char          *aattributeP,
```

```
long          aattributeLength,
char          **avaluePP,
long          *areaLengthP
);
```

Appendix C

Components of a Typical VFS+ Package

C.1 The VFS Vector

The first item to be considered in providing a VFS+ package is providing a new VFS vector for the file system to be exported. Typically, the new vector is like the old vector, but some of the functions are wrappers. In particular, the *vfs_root* and *vfs_vget* functions are wrappers; they call the original function to obtain a vnode, and then *convert* that vnode, by replacing its old vnode ops vector with a new one.

Converted (VFS+) vnodes can be distinguished from normal (VFS) vnodes by a status flag in the **v_flag** field, **V_CONVERTED**. (In IBM's AIX, the flags field is the **gn_flags** field of the gnode; in DEC's Ultrix, the flags field is the **g_flag** field of the gnode.) If this flag is set, the vnode is a VFS+ vnode, and the functions pointed to by the **v_op** field have their VFS+ definitions. Otherwise, the functions are basic VFS functions and have their usual definition.

The conversion process works by checking for the **V_CONVERTED** flag. If present, there is nothing to be done, and the conversion process terminates. Otherwise, an updated set of vnode functions is generated from the existing vnode functions, and the updated function array is placed into the vnode's **v_op** field. Finally, the **V_CONVERTED** flag is set in the **v_flag** field in the converted vnode.

When an updated set of vnode functions has been generated for one vnode in a file system, it can be used for all the other vnodes in the same file system. So to save conversion effort, an association list can maintained, in which each pair consists of an old vnode ops array, and a new one. When a vnode is to be converted, the address of its vnode ops array can be looked up in the list, and if it is found (as an ''old'' array), the appropriate ''new'' array replaces it.

C.2 Naming Conventions

In this document, naming conventions are used for both the macros defined for vnode functions as well as the functions themselves.

- **Vnode functions**. One naming convention identifies the ''family'' to which a given vnode function belongs. Let us take the example of vnode functions implemented for the **UNIX** BSD file system, or **UFS**. Furthermore, let *fname* be the suffix portion of the function name pointed to from the **struct vnodeops** array, such as *rename*, *lookup*, *open*, and *mkdir*. In this appendix, *ufs_fname*( ) refers to the version of *fname* as it normally appears in the **struct vnodeops** function array straight from the vendor. The *glue* version of the same function, whose address is overwritten into this array (as an *O-op*) and which performs the additional synchronization operations transparently, is referred to as *xglue_fname*( ). Finally, the extended version of the same function, the one that is normally used by the **DFS** class of exporters directly (as an *X-op*) and for which the caller has already performed all the synchronization work is named *xufs_fname*( ). See Section 11.4 on page 238 for more information on *O-ops*, *N-ops*and *X-ops*.
- **Vnode macros**. There are a standard set of vnode macros defined by the file system. As in the previous paragraph, let *fname* be the suffix portion of the function name pointed to from the **struct vnodeops** array. Then, in Sun's original VFS, **VOP_FNAME** is the macro that indirects through the **struct vnodeops** array in the vnode descriptor and calls the appropriate

base function there. For clarity in DFS, an equivalent macro, **VOPO_FNAME**, is defined which accesses the same slot in the array but indicates clearly that an *O-op* is expected to be there. A second set of macros, **VOPX_FNAME**, used by DFS exporters, is also defined which indirects through the same array but calls the extended version of the same function (that is, the *X-op*). Similarly a third set of macros, **VOPN_FNAME**, is defined to indirect through the *N-op* slots in the same array.

C.3 Glue Functions

Most of these *glue functions* are derived from the corresponding vnode functions by writing a new function that calls the old one as part of its operation. Here is pseudo-code for the *xglue_rename*( ) function. It is written with an old-style Sun VFS interface in mind; the versions for other VFS-like interfaces, such as that used in IBM's AIX, would be somewhat different.

C.3.1 xglue_rename( ) Function

Example C-1 xglue_rename( ) Function

```
xglue_rename(vnode, name, tovnode, toname, cred)
struct vnode *vnode, *tovnode;
char *name, *toname;
struct ucred *cred; {
    struct vnode *tvp1, *tvp2;
    struct afs_tokenSet *tset;
    struct volume *vold;

    if (code = ReferenceCorrespondingVolume(vnode, &vold))
        return(code);

    if (NoTokensRequired (vold)) {
        code = VOPN_RENAME(vnode, name, tovnode, toname, cred);
        return(code);
    }

    VOPN_LOOKUP(vnode, name, &tvp1, cred);
    ConvertVnode(tvp1);
    VOPN_LOOKUP(tovnode, toname, &tvp2, cred);
    ConvertVnode(tvp2);

    tset = NewTokenSet();
    AddTokenSet(tset, vnode, DATA_WRITE+STATUS_WRITE);
    AddTokenSet(tset, tovnode, DATA_WRITE+STATUS_WRITE);
    AddTokenSet(tset, tvp1, STATUS_WRITE);
    AddTokenSet(tset, tvp2, STATUS_WRITE);
    ObtainTokenSet(tset);

    code = VOPN_RENAME(vnode, name, tovnode, toname, cred);

    ReturnTokenSet(tset);
    VN_RELE(tvp1);
    VN_RELE(tvp2);
```

```
    ReleaseCorrespondingVolume(vold);
    return(code);
}
```

The **Token Manager** is called to acquire and release tokens on all of the vnode-class operands, whether they are explicit or implicit in the VFS interface. Thus tokens must be obtained for the object being renamed, and the object (if any) that it will replace, though their vnodes are not explicit parameters of the function.

C.3.2 xglue_lookup() Function

Example C-2 xglue_lookup( ) Function

Calls like those done via the *VOP_LOOKUP* macro that potentially return new vnodes must also call the conversion function on the returned vnode parameters. For instance, the *xglue_lookup*( ) call looks like this:

```
xglue_lookup(vp, name, vpp, cred)
struct vnode *vp, **vpp;
char *name;
struct ucred *cred; {
    long code;
    struct afs_tokenSet *tset;
    struct volume *vold;

    if (code = ReferenceCorrespondingVolume(vp, &vold))
        return(code);

    if (NoTokensRequired (vold)) {
        code = VOPN_LOOKUP(vp, name, vpp, cred);
        return(code);
    }

    tset = NewTokenSet();
    AddTokenSet(tset, vp, STATUS_READ);
    ObtainTokenSet(tset);
    code = VOPN_LOOKUP(vp, name, vpp, cred);
    ReleaseTokenSet(tset);
    if (*vpp)
        ConvertVnode(*vpp);    /*Convert to VFS+ vnode*/

    ReleaseCorrespondingVolume(vold);
    return(code);
}
```

Note that the returned vnode (***vpp**) is converted into a VFS+ vnode. Ensuring that all vnodes in the system are converted is accomplished by intercepting the *vfs_vget*( ) and *vfs_root*( ) VFS functions as well as all returns of newly-created vnodes. The result is that after the interception code is installed, all VFS vnodes will be converted to VFS+ vnodes, and will thus call the VFS+ code instead of the normal VFS code from then on.

In other words, after the VFS+ interception code is enabled, all processing that starts with a mount point and continues down the file system tree via *lookup*( ) and *create*( ) calls will deal exclusively with VFS+ vnodes, rather than the generic VFS vnodes.

Notice the calls to *ReferenceCorrespondingVolume*( ) and *ReleaseCorrespondingVolume*( ) in the code above. These functions are responsible for ensuring that whole-fileset operations synchronize properly with the basic vnode operations. If the fileset concept is meaningless for a particular type of file system, these functions need do nothing but return zero (in which case the vnode operations will never wait), or return *inaccessible* when referring to data on a particular file system.

If there is a meaningful fileset concept for a particular file system, then there is a reference-counted fileset structure in one of three states:

Normal File operations may proceed on this fileset.

Busy File operations should wait until the fileset is no longer marked as *busy*.

Error File operations should fail with the specified error code.

C.4 Extended Functions

Pointers to the *xglue_rename*( ) and *xglue_lookup*( ) functions replace those of native (base) functions in the vnode function array. The extended functions, namely the ones that do *not* make calls to the Token Manager, must also be defined. *In most cases, the extended version of the vnode function is identical with the original, vendor-supplied version.* It is assumed for all extended functions that the caller has already dealt with all the synchronization requirements, so it is safe to call the original vendor code. However, all functions in the extended set that return new vnodes must also make sure that these resulting vnodes are the VFS+ versions (that is, they point to the augmented **struct vnodeops** function array). Thus, one may simply use the *ufs_rename*( ) function to implement *xufs_rename*( ) since it doesn't return any new vnode references. This cannot be said for *xufs_lookup*( ), which must be implemented as follows (old Sun VFS version):

C.4.1 xufs_lookup( ) Function

Example C-3 xufs_lookup( ) Function

```
xufs_lookup(vp, name, vpp, cred)
struct vnode *vp, **vpp;
char *name;
struct ucred *cred; {
    long code;

    code = VOPN_LOOKUP(vp, name, vpp, cred);
    if (!code)
        ConvertVnode(*vpp);    /*Convert to VFS+ vnode*/
    return(code);
}
```

This function, like all extended vnode functions, does not attempt fileset synchronization or token acquisition, but simply performs the native version of the lookup operation and then converts the resulting vnode, if one was generated.

C.5 Extended Vnode Attributes

The vnode attribute structure (**struct vattr**) used in UNIX kernels is extended for purposes of exporting files. The extended structure (**struct xvfs_attr**) is given in Section 13.6.3 on page 294. The X-ops, VOPX_GETATTR and VOPX_SETATTR, take an extra argument, which is a flag specifying whether or not the attribute structure argument is extended.

Chapter 12

The DCE DFS ACL Model for an LFS

This chapter discusses ACLS, security and protection checking as they relate to a DFS physical file system.

12.1 Overview

DCE DFS ACLs are are an extension of POSIX ACLs. The specific algorithms presented here are based on draft 13 (/D13) of the POSIX 1003.6.1 spec. This is a recent draft which is simplified in several respects from the earlier ones which are reflected in some of the existing implementations of the DCE DFS. It is assumed that the reader has access to this document.

Note: Since it is not clear what POSIX will finally decide on here, this document also describes (clearly annotated) the older behaviors, as implemented currently in Episode.

12.2 Definition of Terms

The terms here are defined in Section 11.1 on page 233. They are reproduced here for convenience as they are used extensively in the descriptions that follow.

Cell A collection of DCE machines administered as a single entity. In this context, the key point is that a cell is serviced by a single registry with a single, consistent set of user and group identifiers.

Group A membership list, identified by a **uuid_t**, whicn contains a list of principals (by **uuid_t**) which are members of that group. Again, in a non-DCE environment, they are identified by a 32-bit (at best) **gid**.

PAC Privilege Attribute Certificate. A data structure, generated by the DCE Privilege Server, that contains the identity of a DCE authenticated principal. To a first approximation, it contains the principal's **uuid_t** along with the**uuid_t** of each group of which it is a member.

Principal An entity that is interacting with DFS. Although it usually refers to a person, it could as easily correspond to a DCE server or a computer within a DCE cell. In the DCE environment, principals are identified by a **uuid_t**. In a non-DCE, standard UNIX, environment, they are identified by a 32- bit (at best) **uid**.

Realm Equivalent to a Cell. In security DCE security discussions, the term Realm is typically used instead of Cell.

12.3 Primitive Data Types

There are two primitive data types used in this chapter, **sec_id_pac_t** and **permissions_t**. Their definitions can be found in Section 13.1 on page 267.

12.4 Local Realm

Several of the **ACL** algorithms detailed below require the identity of the local DCE Realm (Cell). During **DCE DFS** and LFS (**DCE LFS**) initialization, this value (a **uuid_t**) is supplied to the system via a mechanism discussed elsewhere.

The simplest mechanism is to use the *dacl_GetLocalCellID*( ) call detailed in *Getting the Local Cell ID*, Section 11.8.8 on page 243. In this document, this Cell (by **uuid_t**) is referred to as the LOCAL-REALM.

12.5 DFS Administrator

DFS has the notion of a special System Administrative group which is granted the right to perform various control functions. (In many regards, this group enjoys the same file system privileges that ROOT on a typical UNIX system possesses.)

The simplest mechanism is to use the *xvfs_IsAdminGroup*( ) or *dacl_GetSysAdminGroupID*( ) call detailed in *Getting the Administrative Group ID*, Section 11.8.9 on page 243. If DFS is not active, there is no notion of an administrator.

In this document, this group (by **uuid_t**) is referred to as the DFS-SYSADMIN-GROUP.

12.6 DFS vs non-DFS requests

In certain situations, the desired behavior depends upon whether or not the operation being performed originated from DFS. Although a number of short cuts are possible (say, if DFS is not installed at all), the *logical* determination of this fact is described in *Obtaining the Identity of the Principal*, Section 11.8.10 on page 243.

12.7 ACL Contents

A DFS **ACL** consists of a certain amount of fixed information along with a variable number of ACL entries. The fixed information is the following:

Manager Type A UUID defining the type of manager that supports this **ACL** type. Alternatively, one can view it as defining an **ACL** type. In order to be interoperable with other physical file system implementations, the constant picked for the Episode LFS should be used.

Default. Realm A UUID identifying the realm that non-qualified (by an explicit realm) entries in this **ACL** are interpreted relative to. Usually, this will be the local cell within which this file system resides.

The definition for **ACL** structure, of type **dacl_t**, can be seen in *ACL Structure*, Section 9.8 on page 169.

The remainder of the **ACL** consists of some number of **ACL** entries. **ACL** entries come in several different types, each of which contains a type field, a set of permission rights and some amount of information identifying the principal(s) that this entry applies to. The different types of **ACL** entries are listed below. They are an amplification from an LFS point of view, of the information found in *ACL Entry Types*, Section 8.2 on page 156.

user_obj Contains: permission set. The rights granted to principals within the default realm that belong to the user file class of the file. (That is, the file *oid* field matches the principal's id field.)

group_obj Contains: permission set. The rights granted to principals within the default realm that belong to the group file class of the file. (That is, the file *gid* field matches either the principal's *gid* field or one of groups in its group list.)

other_obj Contains: permission set. The rights granted to any principals from the default realm not matched by any of the user or group types.

Note: A DFS **ACL** is required to contain exactly one of each of the above three entry types (*user_obj*, *group_obj* and *other_obj*). Likewise, the *user_obj* entry is required to contain the **perm_control** right. If either of these restrictions are violated, the **ACL** is corrupt and should be rejected by a *vn_setacl*( ) operation.

user Contains: user UUID, permission set. The rights granted to the specified user in the default realm. No two user entries can contain the same user UUID.

group Contains: user UUID, permission set. The rights granted to the specified group in the default realm. No two group entries can contain the same group UUID.

foreign_user Contains: realm UUID, user UUID, permission set. The rights granted to the specified user from the specified realm. No two foreign_user entries can contain the same realm UUID and user UUID pair.

foreign_group Contains: realm UUID, group UUID, permission set. The rights granted to the specified group from the specified realm. No two foreign_group entries can contain the same realm UUID and user UUID pair.

Note: The default realm in the above two entries (*foreign_user* and *foreign_group*) is required to be different than the ACL's default realm. This requirement might not be necessary or desirable given *ACL* inheritance.

foreign_other Contains: realm UUID. The rights granted to any principals from the specified realm that are not matched by any of the *foreign_user* or *foreign_group* entries. No two *foreign_other* entries can contain the same realm UUID.

any_other Contains: permission set. The rights granted to any principals not matched by any of the other entries. At most one *any_other* entry can appear in an ACL.

extended Contains: an uninterpreted string of bytes (and length). Ignored (but preserved) by DFS.

Note: This is for future extensions.

mask_obj Contains: permission set. This entry type corresponds to functionality dropped in the POSIX. The safest behavior, from an interoperability viewpoint, is to ignore but preserve this entry type. At most one **mask_obj** entry can appear in an **ACL**.

Note: This is an optional entry. Not all **ACL**s contain this entry.

12.8 Define the External ACL Representation

Although there are no requirements on **ACL** representation on disk and in memory, the *external* representation as seen outside the LFS is fixed.

The only limit on **ACL** size is that its external representation must fit within a 8192-byte **dfs_acl** structure.

```
struct dfs_acl {
    long        acl_len;
    char        acl_val[8188];
};
```

The *acl_len* field gives the number of bytes in *acl_val[]* that are valid. For an invalid or empty **ACL**, this will be 0.

The first few components stored in acl_val[] are fixed as follows:

uuid_t A manager UUID which gives type of **ACL** this is (literally, it identifies the manager which interprets it). This field's only use is to be checked for validity when an **ACL** is presented to the file system. If general interoperability is desired, LFS implementations should use the same UUID value that was selected for its Episode LFS. Specifically:

```
uuid_t LFSMgrUuid = { 0xd076c532, 0x0a1d, 0x11ca,
    {0x95, 0x3d, 0x02, 0x60, 0x2e, 0xa9, 0x6e, 0x00}
    };
```

uuid_t The default realm that this **ACL** applies to.

int32 The number of **ACL** entries which follow.

Following this is some number of **ACL** entries in an unspecified order. **ACL** entry types are identified by a type field with the values shown in the following table. See *ACL Entry Type Format*, Section 9.5 on page 167, for their definitions.

ACL Entry Type	Value
user_obj	0
group_obj	1
other_obj	2
user	3
group	4
mask_obj	5
foreign_user	8
foreign_group	9
foreign_other	10
unauth_mask	11
extended	12
anyother	13

Table 12-1 ACL Entry Types

12.8.1 Formats of ACL Types

The formats for the various types of **ACL** entries are shown below. These formats are defined by the **ACL** entry formats in *Complex ACL Entry Format*, Section 9.7 on page 168, and *Simple ACL Entry Format*, Section 9.6 on page 168, for those that are not foreign or extended. That is to say, the simple entries contain the permissions, *perms*, and the value from the above table (to determine if the entry is good), while the foreign types also have a realm uuid. This distinction can be seen in the definitions shown in Section 9.9.1 on page 170 and Section 9.9.2 on page 170.

- *user_obj*, *group_obj*, *other_obj*, *mask_obj*, *any_other*, *unauth_mask*

  permissions_t Associated permissions.

  int32 Type value from above table.

- *user*, *group*

  permissions_t Associated permissions.

  int32 Type value from above table.

  uuid_t *user* or *group* **uuid**.

- *foreign_user*, *foreign_group*

  permissions_t Associated permissions.

  int32 Type value from above table.

  uuid_t *user* or *group* **uuid**.

  uuid_t Realm **uuid**.

- *foreign_other*

  permissions_t Associated permissions.

  int32 Type value from above table.

  uuid_t Realm **uuid**.

- *extended*

  uuid_t Extension type.

  u_int16[3] Miscellaneous stuff.

  char[2]

  U_int32 The number of bytes following this.

  char[] (Whatever).

Note: All **ACL** entries must begin on an integral long (32-bit) boundary in memory. For the purpose of computing the total size of the **ACL** and placing additional entries after it, the size of an extended entry is logically rounded up to a multiple of 4 bytes in size. (Its internal size field can contain an arbitrary value, however.)

In this external representation, all fields are stored in network canonical order. Fields of type *long*, *int32* and **permissions_t** are run through *htonl*( ). Fields of type **short** (int16) are run through *htons*( ). Fields within a **uuid_t** structure are converted as follows:

- Fields with *.time_low* are run through *ntohl*( ).
- Fields with *.time_mid* are run through *ntohs*( ).

- Fields with *.time_hi_and_version* are run through *ntohs*( ).

Consult your **<stds.h>** header file for definitions of the **integer**, **long** and **ntoh** functions.

12.9 Relationship — ACLs and UNIX Protections

There is a direct equivalence between the *user_obj*, *group_obj* and *other_obj* required entries in an **ACL** and the standard UNIX protection bits (the mode bits). If an object does contain an **ACL** (as always, optional), these must remain consistent. P Although there are any number of implementation possibilities, the most obvious ones are listed below.

Approach 1 Don't associate the standard UNIX protection bits with a file. Instead, require that all objects have an associated **ACL** with at least the required *user_obj*, *group_obj* and *other_obj* entries.

From a performance and disk utilization point of view, it seems most reasonable to always hold these **ACL** entries in an object's inode (anode).

With this approach, changing the mode bits on a file automatically adjusts the **ACL** in a corresponding fashion.

Approach 2 Associate both the UNIX protection bits and an optional **ACL** with an object. When the object's mode bits are modified as a result of a **chmod** operation or file creation, the required entries in the **ACL** are changed.

Likewise, when an object's **ACL** is explicitly changed as a result of a set **ACL** operation, its mode bits are changed as well (to agree with the **ACL**).

With this approach, an object's **ACL** is always consistent with its UNIX permission bits. However, it may suffer from performance and disk utilization problems.

Approach 3 Associate both the UNIX protection bits and an optional **ACL** with an object.

When an object's **ACL** is explicitly changed as a result of a set **ACL** operation, its mode bits are changed as well (to agree with the **ACL**).

When the object's mode bits are modified as a result of a **chmod** operation or file creation, its **ACL** is left unchanged.

When an **ACL** is returned to user space during a get **ACL** operation, the read-write-execute permissions in its *user_obj*, *group_obj* and *other_obj* entries are changed to agree with potentially more up-to-date mode bits on the object.

Note: This does not occur for initial-file and initial-directory **ACL**s on a directory.

When this step is performed for a directory's **ACL**, the **perm_insert** and **perm_delete** rights will additionally be turned on if the **perm_write** one ends up being set (regardless of its initial state). Note the assymetry here: although these **perm_insert** and (or) **perm_delete** rights may be turned on during this process, they will not be turned off -- even if the **perm_write** right is removed.

When permission checking is done against an **ACL**, the permissions from the mode bits are used instead of what is found in the *user_obj*, *group_obj* and *other_obj* entries.

This approach has the same advantageous performance and disk utilization behavior that **Approach 1** enjoys.

Note: The second approach is not recommended at all. Either of the other two schemes seems reasonable. Episode follows the third approach.

The access check algorithm in *Access Rights Algorithm*, Section 12.16 on page 262, assumes **Approach 3**.

12.9.1 Episode Visibility above VFS+

As pointed out in the **fileset** and **vnode** operations chapters, under *vol_getacl*( ), (see Section 15.14 on page 365 and so forth) *vol_setacl*( ), *vol_copyacl*( ), *vn_setacl*( ) and *vn_getacl*( ) (in Section 16.8 on page 465 and so forth), Episode's choice of **Approach 3** above appears to make itself visible above the VFS+ layer. Specifically:

[1] Episode follows approach 3. Although an object's mode bits and regular **ACL** can disagree, the mode bits are the **truth** used during **ACL** checking algorithms.

[2] The *vn_getacl*( ) operation coerces (depending upon **ACL** type) the **ACL**'s required entries to agree with the object's mode bits, while *vol_getacl*( ) doesn't.

[3] The *vn_setacl*( ) operation coerces (depending upon **ACL** type) the object's mode bits to agree with an **ACL**'s required entries, while *vol_setacl*( ) doesn't.

[4] The *vol_copyacl*( ) operation, along with *vn_setacl*( ) when invoked in its copy mode, totally ignore an object's mode bits.

Items [2] and [3] don't seem to be that major since *vol_getacl*( ) and *vol_setacl*( ) are used during backup/restore/move operations. The important point here is that an object backed up by a *vol_getattr*( ) and *vol_getacl*( ) (in that order, although it shouldn't matter) appear UNCHANGED after it is restored by a *vol_copyacl*( ) or *vol_setacl*( )and *vol_setattr*( ) (in that order). A major concern is guaranteeing interoperability of the dump/restore format with other LFS implementations such as Episode.

Item [4] is more difficult. However, it has the virtue of allowing for maximum **ACL** sharing if the application (**acl_edit**, for instance) takes care of the mode bits itself.

Taking everything into account, the recommendation is to take the safest course and do as Episode does (**Approach 3**).

12.10 ACL Creation From Mode Bits Algorithm

There are several circumstances in which it is necessary to construct an **ACL** from a set of UNIX mode bits. One example is when a get **ACL** operation is performed on an object which does not contain an **ACL**. In such cases, the **ACL** is built as follows.

a. The manager type field (**uuid_t**) is set to the correct value (LFSMgrUuid from above).

b. The default realm field (**uuid_t**) is set to the LOCAL-REALM.

c. The *user_obj*, *group_obj* and *other_obj* entries (the required ones) all have their permissions taken from the mode bits in question. The **perm_read**, **perm_write** and **perm_execute** rights are copied in directly. If the object in question is a directory, **perm_insert** and **perm_delete** rights are granted whenever **perm_write** rights are. For the *user_obj* entry, the **perm_control** right is added as well.

d. The *number-of-entries* field within the ACL is set to 3.

12.11 Initial ACL and File Creation Algorithm

Any file system object can have an attached **ACL** which governs access to that object. Directories may additionally have two associated optional **ACL**s that are used during object creation.

The *Initial Object (Creation)* **ACL**, if present, is applied to any files created under that directory. (Initial **ACL**.)

The *Initial Container (Creation)* **ACL**, if present, is applied to any sub-directories created under that directory.

Note: Within this chapter, an unqualified use of the term **ACL** usually refers to the regular **ACL** used for access checking.

When an object is created, an initial **ACL** may be applied to or created for it as follows.

1. When a file is created, any *Initial Object* **ACL** of its parent directory is given to the object. If there is no *Initial Object* **ACL**, proceed to step [4].
2. When a directory is created, any *Initial Container* (Directory) **ACL** of its parent directory is given to the new directory. Additionally, the directory's *Initial Object* and *Initial Container* ACLs are inherited from the parent directory as well. If there is no *Initial Container* directory **ACL**, proceed to step [4].
3. The object's mode bits (0777) are replaced by a value computed from the *user_obj*, *group_obj* and *other_obj* entries of its **ACL** (**perm_read**, **perm_write**, **perm_execute**). If any of these entries are invalid, the corresponding mode bits are left 0. (In other words, the initial **ACL** overrides the initial mode bit setting)
4. If the create operation occurs via DFS, the remote principal's PAC indicates the realm from which the request originates (the *.realm* field of the **sec_id_pac_t** type). For a non-DFS operation (there is no PAC), the request is viewed as originating from the LOCAL-REALM.
5. If the originating realm is the LOCAL-REALM and the newly created object has no **ACL** (no inherited **ACL**), proceed to step [6]. If the originating realm agrees with the default realm in the object's **ACL**, proceed to step [6].

   Otherwise, the existing **ACL** (if any) must be modified to reflect the realm from which the create request actually originated.

   If the object has no **ACL**, one is synthesized for it from the file's mode bits as described in **ACL** *Creation From Mode Bits Algorithm*, Section 12.10 on page 257.

   The resultant **ACL**, however it was obtained, is modified as listed below.

   - If the originating request originated from DFS and is not authenticated (PAC *.authenticated* field = 0), an *unauth_mask* entry is added to the **ACL** with a permissions field of all-ones.

     Note: Currently, PACs presented to the LFS by DFS are always authenticated. Refer to *Obtaining a Principal's Identity*, Section 12.13 on page 259.

   - The default realm in the **ACL** is set equal to the originating realm (step [4]).
   - Any user or group entries in the **ACL** are changed to *foreign_user* and *foreign_group* ones. The realm field in these *foreign_xxx* entries is set equal to the ORIGINAL realm that was associated with the **ACL**.
   - Any *foreign_user* or *foreign_group* entries with a realm field equal to the NEW **ACL** realm (the originating realm) are changed to *user* and *group* ones, respectively.

6. If the new object is a directory that posesses an *Initial Object (Creation)* **ACL**, perform the same operations detailed in step [5] on THAT initial file **ACL** (instead of on the object's regular **ACL**).

   If the new object is a directory that possesses an *Initial Container* **ACL**, perform the same operations detailed in step [5] on THAT initial directory **ACL** (instead of on the object's regular **ACL**).

12.12 User and Group Identities

(This is the saga of the **uuid_t** versus the 32 bit UNIX ID.) When a DCE registry assigns accounts, it guarantees that a 32 bit UNIX **user** or **group** ID agrees with the first 32 bits of the corresponding DCE **uuid_t**. Although this behavior could in theory change at some time in the future, it is unlikely to as long as the underlying operating systems upon which DCE and DFS are layered deal natively with these 32s-bit values instead of **uuid_t'**s.

Because of this fact, the access right algorithm given later is only required to take into account the first 32-bits of *user* and*group* **uuid_t** values. Likewise, only these first 32 bits are taken into account when comparing a **uuid_t** against 32-bit file **oid** or **gid** values as well as fields from a non-DFS **ucred** authentication structure.

12.13 Algorithm for Obtaining a Principal Identity

During a DFS access, a principal's identity is contained in a PAC structure (**sec_id_pac_t**). In an ideal world, each of the extended **vnode** operations would have been enhanced to include an argument of this type which could then, as discussed in a following section, be used during access checks. However (from the point of view of the LFS), these extended **vnode** operations continue to take only the standard **struct ucred** argument from which, in the DFS case, an appropriate PAC must be obtained.

Before going into how this can be achieved, it might be best to discuss what happens when a remote DFS request arrives at a server. Although the actual specifics are not in themselves relevant, a rough understanding of what is happening is useful.

a. We initially start out with an rpc-handle, an **afsFid** (object identifier) and the call's arguments.

b. A call to *px_AdjustCell*( ) is made to force the *afsFid.Cell* field into a known state (.*high* = 0, .*low* = 1). (Presumably, certain code relies upon this behavior.)

c. A call to *fshs_InqContext*( ) is made in order to obtain the context (identity, and so forth) of the remote caller. This is followed by a call to *fshs_GetPrincipal*( ) to obtain information regarding the principal at the remote site, including a PAC (**sec_id_pac_t**) and an appropriate **ucred** structure.

d. A series of *xcred_xxx*( ) calls are made in order to embed this **ucred** and PAC within an extended credentials **xcred** structure and also plant within the **ucred** a PAG (Process Authentication Group) field which can later be used to "track down" this **xcred** and, hence, the PAC.

e. A call is made to *volreg_LookupExtended*( ) with the object **afsFid** in order to locate the exported fileset in question (indicated by the **afsFid** .*Volume* field). This call returns a pointer to a **struct volume**. After this volume is held (*vol_hold*( )), the **vnode** of the desired file is obtained by a call to *vol_vget*( ).

f. A call is made into the file system to perform the operation via a *VOPX_xxx*( ) call. From within the file system code, the path back to the PAC structure required for access checks is the PAG which was left in the **ucred** structure.

g. The volume and vnode are released via *vol_rele*( ) and *vn_rele*( ) calls. (The principal and context structures are released as well.)

From within the LFS (step (f) above), the desired PAC is obtained as follows. This description is presented as actual C code since it is so dependent upon calls into the DFS portion of the system.

Note: The precise declarations for these *xcred_xxx*( ) calls can be found in *DCE DFS Credential Design*, Section 17.1 on page 469.

```
/* Call xcred_UcredToXcred to use a PAG embedded within
 * the ucred, if any, to locate an extended credential
 * structure (xcred) which will eventually lead to the
 * PAC. It takes a ucred argument and, if successful,
 * returns a pointer to an xcred structure.
 */
rtn_code = xcred_UcredToXcred(&ucred, &xcred_ptr);
if (rtn_code == 0)
    /* Call xcred_GetProp to actually obtain the PAC.
     * It takes an xcred structure argument and, if
     * successful, returns a pointer to a PAC
     * (* sec_id_pac_t) along with the size of the PAC
     * - which we don't care about.
     */
    rtn_code = xcred_GetProp(xcred_ptr, "DCE_PAC", 7,
               &pac_ptr, &pac_size);

if (rtn_code == 0) {
    /* We have a PAC: execute the access check
     * detailed below with with this.
     */
}
else {
    /* There is no PAC.  Presumably, this is a non-DFS or
     * local access.  Either execute an access check
     * algorithm optimized to work off a ucred or "create"
     * a PAC from the ucred via the rules given below and
     * execute the PAC-based access check algorithms.  In
     * the latter case, the space for such a PAC (a
     * sec_id_pac_t along with a group list) is allocated
     * and freed in a file system dependent manner.
     */
}
```

If the DFS request in question is unauthenticated, the PAC returned by *xcred_GetProp*( ) is set to an unauthenticated identity as shown below. Because even these PACs appear to be authenticated, normal **ACL** entries can be established to match unauthenticated users.

.authenticated	non-zero (true)
.principal.uuid	-2 (1st 32 bits: used in access check)

.group.uuid	-2 (1st 32 bits: used in access check)
.realm.uuid	-2 (1st 32 bits)
remaining fields	set to 0

12.14 Algorithm for Generating PAC from Ucred Structure

As was mentioned above, (Section 12.13 on page 259), it may be necessary during a on-DFS access to synthesize a PAC from the contents of a standard ucred structure. The simplest way to accomplish this is to start out with both a **sec_id_pac_t** PAC structure and a maximum sized (OSI_MAXGROUPS_1) **sec_id_t** group list (array of **sec_id_t**, that is). The PAC is initialized as follows with fields not mentioned set to 0).

.pac_type	= *sec_id_pac_format_v1* (= 0)
.authenticated	= true (actually, to, -1)
.realm.uuid	= LOCAL-REALM
.principal.uuid.time_low	= **uid** from **ucred** (by *osi_GetUID*( ) call)
.group.group.uuid.time_low	= **gid** from **ucred** (by *osi_GetGID*( ) call)
.num_groups	= number of groups in the ucred group list (via *osi_GetNGroups*( ))
.groups	= pointer to the **sec_id_t** *group* array

For each of these groups, the .groups[] entry is zeroed and then filled in as follows:

.groups[i].uuid.time_low	= *group-id* from **ucred** *.cr_groups[i]*

Note that a 32-bit UNIX ID-to-**uuid_t** conversion is performed via a copy into the first 32-bits of the **uuid_t**.

Note: Although this initialization could also be accomplished via a call to the DFS function *dacl_PacFromUcred*( ), a manual description is presented above in the interest of minimizing the actual interactions between an LFS and DFS.

12.15 perm_control Access Right

Operations such as *chmod*( ) which, would in a traditional UFS file system, require the caller to own a file in question, instead require **perm_control** access. The owner of a file will always have this right.

12.16 Access Rights Algorithm

The access check algorithm presented below uses the following parameters.

desired-access A **permset_t** detailing the desired access.

file-mode-bits The object's mode protection bits.

file-oid The object's owner-ID value (32-bit UNIX **uid**).

file-gid The object's group-ID value (32-bit UNIX **gid**).

When the file-oid or file-gid are compared against a **uuid_t** below, the comparison actually occurs against the first 32-bits of the **uuid_t**.

ucred The **ucred** structure of the principal on whose behalf the request is being made.

ACL The object's **ACL**, if any.

PAC The PAC identifying the principal on whose behalf the request is being made. For a DFS access, a PAC is always available (as discussed above). For non-DFS access (local, NFS access), the following algorithm assumes that a PAC is built from the available **ucred** structure as discussed above. Alternatively, one could provide two bodies of code; one working off of a PAC (DFS access) and another working off of a **ucred** (non-DFS access).

Associated with this PAC is an indication of whether this is a DFS access or a non-DFS access.

Note: A PAC is only available for requests that arrive via DFS. In the situation where a local user has performed a **dce_login**, there is no PAC that an LFS (or DFS) has access to.

For the purpose of user and group comparisons, only the first 32-bits of **uuid_t** values need to be considered. This is the size of **file-oid** and **file-id** fields as well as the that of the relevant **ucred** fields that we started with in the event of a non-DFS access. All realm comparisons are performed on the full **uuid_t** .

Note: This is the behavior of Episode LFS.

Several of the steps below test for an authenticated user. Although they are present in Episode, they are never triggered since PACs presented by DFS to the LFS are always (currently, at least) authenticated. Refer to *Algorithm for Obtaining a Principal's Identity*, Section 12.13 on page 259, and also **ACL** *Entry Types for Unauthenticated Users*, Section 8.2.3 on page 157.

The algorithm shown below can, in general, be executed in one of two situations:

a. The set of rights that the caller has to an object is being computed (as returned by *vn_getattr*( ) or *vol_getattr*( ) in the *.callerAccess* field).

b. A determination is being made as to whether the caller has a specific access to an object (as performed by the *vn_acess*( ), *vn_lookup*( ), types of vnode operations)

In the Episode implementation, several of the individual steps differ depending upon which of the two cases is being handled. These are called out below.

If there is no object **ACL**, the rights that the principal has to the object are determined as in Section 12.16.1 on page 263.

Note: If the implementation were willing to construct an **ACL** in this case, the "there is an **ACL**" algorithm in Section 12.16.2 on page 263 could presumably be followed.

In either case, the desired access can finally be compared against the granted rights computed in Section 12.16.2 to determine if the access is allowed.

12.16.1 No Object ACL Exists Rights Algorithm

1. If the principal in question is root, full rights are granted: **perm_control**, **perm_read**, **perm_write** and**perm_execute**. If the object in question is a directory, **perm_insert** and **perm_delete** rights are granted as well. Root access is indicated by an all-zero *.principal.uuid* field in an authenticated PAC (in the non-DFS situation, from a **uid** of 0). Continue with step [7].

   Note: Episode doesn't perform this step in the case of a *vn* (or *vol_getattr*( )) operation obtaining a remote client's allowed access (*.callerAccess*) to an object.

2. If the PAC *.authenticated* field is 0 (an unauthenticated DFS access), no rights are granted. Exit the algorithm.

3. If the the *.realm.uuid field* in the PAC differs from the LOCAL-REALM (a DFS request), no rights are granted. Exit the algorithm.

4. If the PAC *.principal.uuid* field matches the *file-oid* field, the **owner** bits in the file-mode-bits are granted. Additionally, **perm_control** access is granted as well. Continue at step [7].

5. If the *file-gid* field matches either the *.group.uuid* field or one of the *.groups[].uuid* fields in the PAC, the *group* bits in the file-mode-bits are granted. Continue at step [7].

6. If all else fails, the *other* bits in the file-mode-bits is granted.

   Note: In the above UNIX mode-to-**permset_t** rights conversion, a straight forward **read-write-execute** to **perm_read-write-execute** mapping is performed.

7. If the object is a directory and the **perm_write** right has been granted, the **perm_insert** and **perm_delete** ones are as well.

8. Continue with step [k] in the ACL case algorithm in Section 12.16.2 (the next section).

12.16.2 Object ACL Exists Rights Algorithm

If there is an object ACL, the rights are determined instead of in the previous section, as follows.

a. If the PAC *.realm.uuid* is all zeroes, the steps below behave as if it were equal to the default **realm** field from the **ACL**.

b. If the principal in question is root, full rights are granted: **perm_control**,**perm_read**, **perm_write** and **perm_execute**. If the object in question is a directory, **perm_insert** and **perm_delete** rights are granted as well. Root access is indicated by an all-zero *.principal.uuid* field in an authenticated PAC (in the non-DFS situation, from a uid of 0). Continue with step [l] to perform a Read-only (RO) fileset check.

c.

Note: This step starts *User* entry checking. It matches from the **file-uid**

If the PAC *.realm.uuid* field agrees with the default **realm** from the **ACL** and the PAC *.principal.uuid* field matches the **file-uid** (or oid), the rights from the *user_obj* entry are granted and continue with step [k].

As discussed earlier, an implementation may need to replace the **ACL perm_read**, **perm_write** and **perm_execute** rights with the corresponding bits from the file-mode bits.

d. If the PAC *.realm.uuid* field agrees with the default realm from the **ACL**:

The **ACL** is searched for an entry of type *user* whose user **uuid** matches the PAC *.principal.uuid* field.

Otherwise, the **ACL** is searched for an entry of type *foreign_user* whose user uuid matches the PAC *.principal.uuid* field and whose **realm uuid** matches the PAC *.realm.uuid* field.

If such an entry is found, the rights in it are granted and continue with step [k].

e.

Note: This step starts *Group* entry checking. It matches from the **file-gid**

If the PAC *.realm.uuid* field agrees with the default **realm** from the **ACL**:

If the **file-gid** matches the PAC *.group.uuid* field or one of the PAC *.groups[].id.value* fields, the rights from the *group_obj* entry are granted and continue with step [k].

As discussed earlier, an implementation may need to replace the **ACL perm_read**, **perm_write** and **perm_execute** rights with the corresponding bits from the file-mode bits.

f.

Note: This step matches against some **ACL group** entry.

If the PAC *.realm.uuid* field agrees with the default **realm** from the **ACL**:

The **ACL** is searched for entries of type *group* whose *.group.uuid* matches either the *.group.uuid* or one of the *.groups[i].id.uuid* fields in the PAC.

Otherwise, the **ACL** is searched for entries of type *foreign_group* whose *.realm.uuid* matches the PAC *.realm.uuid* and whose *.group.uuid* matches either the *.group.uuid* or one of the *.groups[i].id.uuid* fields in the PAC.

The rights from ALL such **ACL** entries found are "or'd" together to form the granted access.

If at least one such **ACL** entry is found, continue with step [k].

g.

Note: This step matches against the **ACL other** entry.

If the PAC *.realm.uuid* field agrees with the default **realm** from the **ACL**, the rights from the *other_obj* entry are granted and continue with step [k].

As discussed earlier, an implementation may need to replace the **ACL perm_read**, **perm_write** and **perm_execute** rights with the corresponding bits from the file-mode bits.

h. The **ACL** is searched for an entry of the type *foreign_other* whose *.realm.uuid* matches the PAC *.realm.uuid*. If such an entry is found, the rights in it are granted and continue with step [k].

i. If an **ACL** entry of the type *any_other* exists, the rights in it are granted and continue with step [k].

j. If this step is reached, no rights are granted thus far.

k.

Note: This step begins finishing up, and is arrived at from several previous places in the processing algorithm.

If the PAC *.authenticated* field is zero (0), this is an unauthenticated request and the rights computed thus far are masked with the *any_other* entry if present, or if not, a mask of zero

is used.

Note: See **ACL** *Entry Types for Unauthenticated Users*, Section 8.2.3 on page 157. If there is an *unauth_mask* entry in the **ACL**, it should be ignored. If the version of the system is DCE 1.1 or newer, these entries should be removed.

l. If the object resides on a read-only fileset and is not a device file, the **perm_write** permission is removed if it has been granted thus far.

Note: This allows device files to reside on read-only (RO) filesets.

Note: Episode doesn't perform this step in the case of a *vn_getattr*( ) or *vol_getattr*( ) operation obtaining a remote client's access (.callerAccess) to a file.

m. If the object is a directory:

The **perm_insert** and **perm_delete** rights are removed (if present) if **perm_write** is not present.

Otherwise, the **perm_insert** and **perm_delete** rights are not meaningful and can be removed.

n.

Note: This is for DFS-SYSADMIN-GROUP processing.

If the PAC *.realm.uuid* field equals the LOCAL-REALM (the request originates from the local cell) and the DFS_SYSADMIN_GROUP is equal to one of the groups indicated in the PAC *.group.uuid* or *.groups[].uuid* fields, the **perm_control** right is added as well.

12.17 mask_obj ACL Entry Algorithm Impact

As was mentioned earlier, no mention was made of the *mask_obj* **ACL** entry in the above algorithms since the functionality provided by it has been dropped from the POSIX **ACL** proposals. The impacts that this *mask_obj* field would have on the above algorithms are listed below for a number of reasons.

1. Episode and DCE in general, implements to an older POSIX specification that does contain the *mask_obj* functionality.
2. POSIX has not as yet accepted the new proposal.
3. Complete interoperability with Episode and the rest of DCE might require that an LFS implement this functionality.

The general idea is that during a *chmod*( ), the group mode bits are used as a mask (the *mask_obj* ACL entry) that weakens all *user* and *group* type **ACL entries**. In order to accomodate this *mask_obj* functionality, the above algorithms are modified as follows.

- When an **ACL** is being validated during a *vn_setacl*( ) operation, an **ACL** is considered corrupt if it there are any entries of type *user*, *foreign_user*, *group* or *foreign_group* and there is not a *mask_obj* entry.

  Note: Episode also requires a *mask_obj* entry if there are any *foreign_other* entries.

- During a *chmod*( ) operation to an object with an **ACL**, the **read-write-execute** group mode bits being set are copied into the *mask_obj* entry if it exists. (If there is no *mask_obj* entry, they are copied into the *group_obj* entry instead.)

Note: As discussed earlier, an implementation may leave the **ACL** alone and instead force this coercion at the time of a latter *vn_getacl*( ) or access rights check.

- During a *vn_setacl*( ) operation, the **read-write-execute** rights from the **ACL** *mask_obj* entry are copied into the group file mode bits. (If there is no *mask_obj* entry, rights from the *group_obj* entry are instead copied into the group file mode bits.)
- During the access rights checking algorithm presented earlier, the rights in a *mask_obj* entry (if present) are used to weaken ("and operation") the rights in any **ACL** entries of type *user*, *group_obj*, *group*, *foreign_user* and *foreign_group*.

  Additionally, the presence of an *mask_obj* entry causes step [f] in the previous section, Section 12.16.2 on page 263, to be performed even if step [e] triggers a hit. In this case, the rights constructed by [f] and [e] are "or'd" together to construct the actual rights granted.

12.18 Miscellaneous Topics and Suggestions

1. If **ACL**s become prevalent, performance and disk utilization concerns will likely argue for **ACL** sharing (both on disk and in memory). As pointed out, most objects created by principals in a *foreign* realm have an **ACL**.
2. The DFS *vn_getattr*( ) and *vol_getattr*( ) requests are required to return the **minimum rights** granted by an object's **ACL** to ANY user (authenticated or not). Performance concerns might warrant "pre-computing" this value and storing it in a dedicated **ACL** or anode (inode) field.

Chapter 13

VFS+ Data Types

13.1 Primitive Data Types

The following are the primitive data types needed by a **DCE LFS** .

afsFid A DFS file identifier. See Section 4.11.8 on page 60 for its definition.

afsHyper A 64-bit identifier for objects such as cells, volumes, and so on. See Section 4.11.7 on page 60 for its definition.

afsTimeval A time value. See Section 4.11.6 on page 60 for its definition.

afsUUID Equivalent to an uuid_t. See Section 4.11 on page 59 of Chapter 4 on page 57 for its definition.

permissions_t DCE permissions are stored within a 32-bit word. Within this long, DFS is cognizant of the rights listed below.

```
#typedef u_int32 permissions_t

#define perm_read       0x1  /* Read rights */
#define perm_write      0x2  /* Write rights */
#define perm_execute    0x4  /* file: Execute rights */
                             /* directory: Search rights */
#define perm_control    0x8  /* the right perform various */
                             /* control operations (for example, */
                             /* changing the mode bits) */
#define perm_insert     0x10 /* right to add dir entries */
#define perm_delete     0x20 /* right to remove dir entries */
```

The remaining bits, although not interpreted by DFS, are fully preserved in **ACL**s.

sec_id_pac_t The PAC (Privilege Attribute Certificate) identifying an authenticated principal.

```
struct sec_id_t {
    uuid_t                  uuid;
    idl_char                *name;
};
struct sec_id_foreign_t {
    sec_id_t                id;
    sec_id_t                realm;
};

struct sec_id_pac_t {
    sec_id_pac_format_t     pac_type;
    long                    authenticated;
    sec_id_t                realm;
    sec_id_t                principal;
    sec_id_t                group;
    short                   num_groups;
```

```
    short                   num_foreign_groups;
    sec_id_t                *groups;
    sec_id_foreign_t        *foreign_groups;
};
```

The following are the definitions of the **sec_id_pac_t** fields.

.pac_type A format type which can be safely ignored (currently, always *sec_id_pac_format_v1* which is 0).

.authenticated Non-zero implies that the principal is authenticated.

.realm The administrative realm with which the user and group identities in this PAC are associated.

.principal Principal's identifier.

.group Principal's primary group.

.num_groups Number of entries in *.groups*.

.num_foreign_groups Number of entries in *.foreign_groups*.

.groups List of groups associated with the default *.realm* field above to which the principal is a member; each entry consists of a group **uuid_t** and an ignored name.

.foreign_groups List of groups associated with other realms to which the principal is a member; each entry consists of both a realm (**uuid_t** and ignored name) and group (**uuid_t** and ignored name).

uuid_t The globally unique name for entities (objects, interfaces, principals, and so on) known by DCE. The DCE variant of the Universal Unique Identifier is listed in Appendix A of the **X/Open DCE: Remote Procedure Call** specification which lists it in the form of table entries. Its code structure is listed here for convenience, and is as follows:

```
typedef struct uuid_t {
    u_int32         time_low;
    u_int16         time_mid;
    u_int16         time_hi_and_version;
    unsigned char   clock_seq_hi_and_reserved;
    unsigned char   clock_seq_low;
    unsigned char   node[6];
    };
```

13.2 Aggregates and Aggregate Registry Data Types

```
#include <dcedfs/osi.h>
#include <dcedfs/common_def.h>
#include <dcedfs/lock.h>

#define    MAX_AGGRNAME    64        /* XXXX */
```

13.2.1 Aggregate Static Status

Aggregate's general static status information: the piece that sits in the **struct aggr** defined in Section 13.2.6 on page 270. These fields are all maintained by software above the LFS and therefore do not need to be stored on disk. For instance, many are taken from or are derived from information in the dfstab table.

```
struct ag_status_st {
    long aggrId;               /* Aggregate Id */
    long nVolumes;             /* Number of attached filesets */
    long spare1, spare2, spare3, spare4;/* Some spares */
    dev_t device;              /* major/minor aggr device number */
    char aggrName[MAX_AGGRNAME]; /* Aggregate name */
    char devName[32];          /* Aggr device's name */
                               /*(One of the AG_TYPE_xxx codes)*/
    char type;                 /* UFS, PFS, AIX3, etc. */
    char spares[16];           /* More spares */
};
```

13.2.2 Aggregate Dynamic Status

The dynamic aggregate status information: the piece that is computed on the fly. Fields within this structure are maintained (or supplied) by the file system dependent code.

The *minFree* field is the number of blocks on the aggregate held in reserve for the exclusive use of root. If this field is 0 (as it is for Episode), then the *totalUsable* field will be equal to the *blocks* field.

```
struct ag_status_dy {
    long blocks;        /* # blks (of size fragsize) in aggr */
    long blocksize;     /* block size of aggregate */
    long totalUsable;   /* total available 1K blocks on aggr */
                        /* available to non-root users */
    long realFree;      /* free 1K blocks avail to non-root */
    long minFree;       /* min free 1K blocks */
    long fragsize;      /* fragment size of aggr */
    long spares[7];     /* Some spares */
};
```

13.2.3 Aggregate Status

This structure holds the status of an aggregate.

```
struct ag_status {
    struct ag_status_st    ag_st;
    struct ag_status_dy ag_dy;
};
```

13.2.4 Valid Aggregate Types

The following are valid aggregate types (ag_status.type). See Section 13.2.1 on page 269.

```
#define    AG_TYPE_UNKNOWN    0
#define    AG_TYPE_UFS        1
#define    AG_TYPE_EPI        2
#define    AG_TYPE_AIX3       3
#define    AG_TYPE_VXFS       4

#define MAX_AG_TYPE           5 /* size of array of agops structures */
```

13.2.5 Adding of New Filesystem Types

Macros to make addition of new file system types easier in the future.

```
#define AG_TYPE_SUPPORTS_EFS(ag)  (((ag) == AG_TYPE_EPI) \
              || ((ag) == AG_TYPE_VXFS))

#define AG_TYPE_TO_STR(ag)  ((ag) == AG_TYPE_EPI ? "LFS" : \
              (ag) == AG_TYPE_VXFS ? "VXFS" : \
              "Non-LFS")
```

13.2.6 Aggregate Structure

The **aggr** structure, below, is a memory resident structure maintained for each aggregate. In general, it is maintained by code above the LFS and is passed as an argument to each of the aggregate operations.

```
struct aggr {
    struct aggr *a_next;        /* next on linked list */
    struct aggrops *a_aggrOpsp; /* Ops on aggregates */
    long a_states;              /* Aggregate states */
    long a_refCount;            /* Reference counter */
    osi_dlock_t a_lock;         /* Lock variable for the structure */
    struct ag_status_st a_stat_st; /* static part of aggregate's status */
    struct vnode *devvp;        /* special vnode for device */
    opaque  a_fsDatap;          /* Pointer to File-system-dependent */
                                /* private data */
};
```

13.2.7 Aggregate Field Definitions as a_* Items

By defining all aggr's fields as a_* they can freely be moved around without changing the code all over the place where this is done.

```
#define    a_aggrId      a_stat_st.aggrId
#define    a_aggrName    a_stat_st.aggrName
#define    a_devName     a_stat_st.devName
#define    a_device      a_stat_st.device
#define    a_type        a_stat_st.type
#define    a_nVolumes    a_stat_st.nVolumes
```

The following fields are defined for compatibility.

```
#define    a_blocks      a_status.blocks
#define    a_blocksize   a_status.blocksize
#define    a_totalUsable a_status.totalUsable
#define    a_realFree    a_status.realFree
#define    a_minFree     a_status.minFree
#define    a_volId       a_status.volId
```

13.2.8 Aggregate States

The following state bits within *a_states* in the **aggr** structure in Section 13.2.6 on page 270 are defined. The second, **AGGR_DELETED**, is obsolete and no longer in use.

```
#define AGGR_EXPORTED    1   /* Exported aggregate */
#define AGGR_DELETED     2   /* Aggregate deleted */
```

13.2.9 Fileset Creation Flags

The following are flags for **volCreate**( ) in the **aggrops** structure in Section 13.2.12 on page 272.

```
#define AGGR_CREATE_ROOT        (0x1)
#define AGGR_CREATE_VALID_FLAGS (0x1)
```

13.2.10 Aggregate Attach Flags

The following are flags for **ag_attach**( ) in the **aggrops** structure in Section 13.2.12 on page 272.

```
#define AGGR_ATTACH_NOEXPORT (0x1)
#define AGGR_ATTACH_VALID_FLAGS (0x1)
```

13.2.11 Aggregate Sync Flags

The following are flags for **ag_sync**( ) in the **aggrops** structure in Section 13.2.12 on page 272.

```
#define AGGR_SYNC_FILESYS       1
#define AGGR_SYNC_COMMITMETA    2
#define AGGR_SYNC_COMMITALL     3
```

13.2.12 Aggregate Operations Vector

The following is the aggregate operations vector.

```
struct aggrops {
    int (*ag_hold)();
    int (*ag_rele)();
    int (*ag_lock)();
    int (*ag_unlock)();
    int (*ag_stat)();
    int (*ag_volCreate)();
    int (*ag_volInfo)();
    int (*ag_detach)();
    int (*ag_attach)();    /* NOT an AG_ATTACH; Accessed explicitly */
    int (*ag_sync)();
};
```

13.2.13 Exported Aggregate Registry Items

The following are the exported globals and functions for dealing with the *Aggregate Registry*.

```
extern struct aggr *ag_root;
extern osi_dlock_t ag_lock;
extern long ag_attached;

#ifdef KERNEL
extern struct icl_set *xops_iclSetp;
#endif /* KERNEL */

extern void ag_Init(void);
extern int ag_PutAggr(struct aggr *), ag_RemoveAggr();
extern struct aggr *ag_GetAggrByDev(dev_t);
extern struct aggr *ag_GetAggr(long);
extern int ag_NewAggr(
    char *aggrNamep,
    long aggrId,
    char *aggrDevnamep,
    char aggrType,
    struct aggrops *aggrOpsp,
    dev_t dev,
    opaque fsdata,
    struct aggr **aggrpp,
    unsigned flags
);

struct aggrops *agOpvec[MAX_AG_TYPE];
```

13.2.14 Vnode Ops Classification

The following are broad classifications of the vnode ops types.

```
#define VNOP_TYPE_NOOP        5  /* noop */
#define VNOP_TYPE_READONLY   10  /* read only vnode op   */
#define VNOP_TYPE_READWRITE  15  /* read write vnode op */
#define VNOP_TYPE_INVALID    30
```

13.2.15 Concurrency for vol_stat_st

The following define the concurrency levels for concurrency field in vol_stat_st. They are used to control concurrency between volops and vnops.

```
#define    VOL_CONCUR_ALLOPS    1
#define    VOL_CONCUR_READONLY 10
#define    VOL_CONCUR_NOOPS    20
```

The following is a macro to determine if a vnode op can execute through, even if its fileset is busy.

```
#define VOL_VNOP_COMPAT(concurr, type) (((concurr == VOL_CONCUR_ALLOPS) || \
    (type == VNOP_TYPE_NOOP) || \
    ((concurr == VOL_CONCUR_READONLY) && \
    (type == VNOP_TYPE_READONLY))) ? \
    1 : 0)
```

13.2.16 Aggregate Table Entry Format

The following defines the **/opt/dcelocal dfstab** file and the sizes of the strings it contains.

```
#define    ASTAB_INFIX    "/var/dfs/"
#define    ASTAB_SFX      "dfstab"

/* Sizes of strings */

#define    ASTABSIZE_SPEC    512
#define    ASTABSIZE_DIR     512
#define    ASTABSIZE_TYPE    64
```

This is the actual **dfstab** table entry structure, defined as a **struct astab**. This structure identifies a **DCE LFS** aggregate that can be exported. For non-LFS partitions only, an additional entry is present - the **Fileset ID**. See Section 13.2.17 on page 274, below for the definition. The **OSF DCE DFS Administration Guide and Reference** provides pertinent information about the **dfstab** file.

```
struct astab {
    char    as_spec[ASTABSIZE_SPEC];    /* block special device name */
    char    as_aggrName[ASTABSIZE_DIR]; /* Aggregate name */
    char    as_type[ASTABSIZE_TYPE];    /* type of physical file system */
    u_long  as_aggrId;                  /* Aggregate Id */
};
```

13.2.17 UFS Entry Extra Data

```
/* File system extra data:  UFS version */

struct ufs_astab {
    hyper  uas_volId;  /* 1st & only volume */
    char   uas_mountedon[ASTABSIZE_DIR]; /* name of mounted-on dir */
};
```

13.2.18 Values for Filesystem Type

```
#define    ASTABTYPE_UFS    "ufs"
#define    ASTABTYPE_EPI    "lfs"
#define    ASTABTYPE_AIX3    "aix3"
#define    ASTABTYPE_VXFS    "vxfs"
```

13.2.19 Adding New Filesystem Types

The following macros are for making the addition of new file system types easier than manual addition.

```
#define ASTABTYPE_SUPPORTS_EFS(atype) ((strncmp(atype, ASTABTYPE_EPI, \
    ASTABSIZE_TYPE) == 0) || (strncmp(atype, ASTABTYPE_VXFS,
        ASTABSIZE_TYPE) \
    == 0))

#define ASTABTYPE_TO_AGTYPE(astype, agtype)  \
    if (!strncmp(astype, ASTABTYPE_UFS, ASTABSIZE_TYPE)) \
         agtype = AG_TYPE_UFS; \
    else if (!strncmp(astype, ASTABTYPE_EPI, ASTABSIZE_TYPE)) \
         agtype = AG_TYPE_EPI; \
    else if (!strncmp(astype, ASTABTYPE_VXFS, ASTABSIZE_TYPE)) \
         agtype = AG_TYPE_VXFS; \
    else \
         agtype = AG_TYPE_UNKNOWN
```

13.3 Fileset Data Types

13.3.1 Define the Fileset Function Array

A description of the operations available within the fileset function array appears below. The array is an exported data type.

```
struct volumeops {
    /* per-fileset operations */
    int (*vol_hold)();
    int (*vol_rele)();
    int (*vol_lock)();
    int (*vol_unlock)();
    int (*vol_open)();
    int (*vol_seek)();
    int (*vol_tell)();
    int (*vol_scan)();
```

```
int (*vol_close)();
int (*vol_destroy)();        /* actually whole-fileset */
int (*vol_attach)();
int (*vol_detach)();
int (*vol_getstatus)();
int (*vol_setstatus)();

/* per-file operations */
int (*vol_create)();
int (*vol_read)();
int (*vol_write)();
int (*vol_truncate)();
int (*vol_delete)();
int (*vol_getattr)();
int (*vol_setattr)();
int (*vol_getacl)();
int (*vol_setacl)();

/* more whole-fileset operations */
int (*vol_clone)();
int (*vol_reclone)();
int (*vol_unclone)();

/* vnode lookup operations */
int (*vol_vget)();
int (*vol_root)();
int (*vol_isroot)();

/* more per-fileset operations */
int (*vol_getvv)();
int (*vol_setdystat)();
int (*vol_freedystat)();
int (*vol_setnewvid)();

/* another per-file operation */
int (*vol_copyacl)();

/* per-fileset operations */
int (*vol_concurr)();
int (*vol_swapids)();
int (*vol_sync)();
int (*vol_pushstatus)();

/* per-file operations */
int (*vol_readdir)();
int (*vol_appenddir)();

/* per-fileset operations */
int (*vol_bulksetstatus)();
int (*vol_getzlc)();

/* another per-file operation */
```

```
    int (*vol_getnextholes)();

    /* another per-fileset operation */
    int (*vol_deplete)();
};
```

13.3.2 Volume Operations Definitions

The following are the definitions of the system calls for the volume operations in the **struct volumeops** Function Array. They can be found in the header file, **/src/file/xvolume/vol_init.h** .

```
#define    VOLOP_OPEN           4    /* Slots 0-3 are reserved */
#define    VOLOP_SEEK           5
#define    VOLOP_TELL           6
#define    VOLOP_SCAN           7
#define    VOLOP_CLOSE          8
#define    VOLOP_DESTROY        9
#define    VOLOP_GETSTATUS     10
#define    VOLOP_SETSTATUS     11
#define    VOLOP_CREATE        12
#define    VOLOP_READ          13
#define    VOLOP_WRITE         14
#define    VOLOP_TRUNCATE      15
#define    VOLOP_DELETE        16
#define    VOLOP_GETATTR       17
#define    VOLOP_SETATTR       18
#define    VOLOP_GETACL        19
#define    VOLOP_SETACL        20
#define    VOLOP_CLONE         21
#define    VOLOP_RECLONE       22
#define    VOLOP_VGET          23
#define    VOLOP_ROOT          24
#define    VOLOP_ISROOT        25
#define    VOLOP_UNCLONE       26
#define    VOLOP_FCLOSE        27        /* XXXX */
#define    VOLOP_SETVV         28
#define    VOLOP_SWAPVOLIDS    29
#define    VOLOP_COPYACL       30
#define    VOLOP_AGOPEN        31
#define    VOLOP_SYNC          32
#define    VOLOP_PUSHSTATUS    33

#define    VOLOP_PROBE         34
#define    VOLOP_LOCK          35
#define    VOLOP_UNLOCK        36

#define    VOLOP_READDIR       37
#define    VOLOP_APPENDDIR     38

#define    VOLOP_BULKSETSTATUS 39
#define    VOLOP_GETZLC        40
#define    VOLOP_GETNEXTHOLES  41
#define    VOLOP_DEPLETE       42
```

13.3.3 The Fileset NextHole Structure

A description of the operations available within the fileset NextHole structure appears below. It can be found in the header file, **/src/file/xvolume/vol_init.h** .

```
/* Parameter for VOLOP_GETNEXTHOLES */
#define    VOLHOLE_MAX_HOLES   10

struct vol_NextHole {
    struct afsHyper startPoint; /* Return all holes starting here or later */
                                /* Zero this to start */
    unsigned long flags;
    unsigned long outCount;     /* count of returned holes */
    unsigned long spare1;
    unsigned long spare2;
    struct vol_holeDesc {
    struct afsHyper holeStart;  /* byte address in file where hole starts */
    struct afsHyper holeLen;    /* byte count of the hole */
    } holes[VOLHOLE_MAX_HOLES];
};
/* IN flags: */
             /* none yet */
/* OUT flags: */
#define VOLHOLE_FLAG_LAST  0x1  /* no more holes in file past these */
```

13.3.4 Define the vol_stat_st Structure

The following structure defines the static portion of a fileset's status.

```
struct vol_stat_st {
    afsHyper volId;           /* on-disk     */
    afsHyper parentId;        /* on-disk     */
    afsTimeval cloneTime;     /* on-disk     */
                              /* when fileset was made via clone, reclone */
    afsTimeval vvCurrentTime; /* when lazy replica's VV was known current */
    afsTimeval vvPingCurrentTime;
                              /* when last tried for VV of lazy replica */
    long type;
    u_long accStatus;   /* volops we'll perform in this trans (bit mask) */
    u_long accError;    /* error to return for incompatible vnode ops */
    u_long states;      /* on-disk  lots of individual VOL_XXX  bits */
    u_long reclaimDally;/* for the R/W backing some lazy-rep R/O */
    long tokenTimeout;
    long activeVnops;   /* count of active vnode ops */
    long volMoveTimeout;
    long procID;        /* pid of process making fileset busy */
    long spare5;
    long spare6;
    long spare7;
    long spare8;
    long spare9;
    char volName[VOLNAMESIZE]; /* on-disk (VOLNAMESIZE is 112) */
    unsigned char concurrency; /* level of concurrency for vnode ops */
    char cspares[15];
```

```
};
```

The portion of a fileset's status which (for the most part) is changed by **DCE DFS** and not by the LFS. Fields tagged with an "/* on-disk .. */ must be stored in permanent, on-disk storage. Except as discussed in the chapter *Fileset (Volume) Operations Interface* or in the chapter *Aggregate Operations Interface*, these status fields are ignored by the LFS layer. Several of the fields appear to be obsolete (and unused). The above fields have the following meanings.

.volId	The ID of this fileset.
.parentId	For a *.backing* (VOL_BACKUP) or *.readonly* (VOL_READONLY) fileset, the ID of the fileset that it was originally built from (in other words, the "read-write" member of a fileset group). For the RW fileset, this field is set to **0,,0**.
.cloneTime	The time at which a cloned fileset (the backing half) is initially created or re-cloned.
.vvCurrentTime	Time that the lazy replica's volume version was known to be current.
.vvPingCurrentTime	Time when DFS tried to last obtain the lazy replica's volume version.
.type	Set to one of VOL_RW or VOL_READONLY; see Section 13.3.9 on page 282 for the encodings of these.
.accStatus	While a fileset is open, set by DFS to the open-type argument that was passed to the *vol_open*( ) call.
.accError	While a fileset is open, set by DFS to the error code to be returned to any conflicting opens that fail.
.states	Fileset state bits. See Section 13.3.9 on page 282. Controls the amount o time to wait (in other words, daily) before actually removing deleted files on read-only replica filesets.
.tokenTimeout	Set to the volume version token expiration time by DFS (at the same time that the *.vvCurrentTime* and *.vvPingCurrentTime* fields are set).
.activeVnops	The number of active vnode operations (from either the protocol exporter or the vfs and vnode glue code) currently operating on this fileset.
.volMoveTimeout	For filesets actively involved in a move operation (either the source or destination), the estimated time of completion. (Used to recover from failures during the operation.)
.procID	The identity of the process (thread, actually) that has the fileset open.
.volName	The name of the fileset.
.concurrency	For an open fileset, one of the VOL_CONCURR_xxx values determining the set of vnode operations that are permitted to proceed concurrently.
.states	The meanings of the state bits (VOL_xxx flags) within this field are listed in Section 13.3.4 on page 277.

13.3.5 Define the vol_stat_dy Structure

The following structure defines the dynamic portion of a fileset's status.

```
struct vol_stat_dy {
    afsTimeval creationDate; /* on-disk */
                             /* when this fileset was created */
    afsTimeval updateDate;   /* on-disk */
                             /* when any update happened in this fileset */
    afsTimeval accessDate;   /* on-disk */
                             /* when any access was made to this fileset */
    afsTimeval backupDate;   /* spare */
    afsTimeval copyDate;     /* on-disk */
                             /* when dump was made that was restored here */
    afsHyper volversion;     /* on-disk */
    afsHyper backupId;       /* on-disk */
    afsHyper cloneId;        /* on-disk */
    afsHyper llBackId;       /* on-disk */
                             /* low-level backing ID */
    afsHyper llFwdId;        /* on-disk */
                             /* low-level forward ID */
    afsHyper allocLimit;     /* on-disk */
    afsHyper allocUsage;     /* on-disk */
    afsHyper visQuotaLimit;  /* on-disk */
    afsHyper visQuotaUsage;  /* on-disk */
    long fileCount;
    long minQuota;
    long owner;
    long unique;             /* on-disk   */
                             /* necessary if ‘‘version’’ is afsHyper? */
    long index;              /* Redundant */
    long rwIndex;
    long backupIndex;
    long parentIndex;
    long cloneIndex;
    long nodeMax;
    long aggrId;
    long spare2;
    long spare3;
    long spare4;
    long spare5;
    long spare6;
    u_long tag;
    u_long msgLen;
    char statusMsg[128];      /* on-disk   */
    char cspares[16];
};
```

The portion of a fileset's status which (for the most part) is changed by **DCE DFS** and not by the LFS. Fields tagged with an "/* on-disk .. */ must be stored in permanent, on-disk storage. Except as discussed in the chapter *Fileset (Volume) Operations Interface* or in the chapter *Aggregate Operations Interface*, these status fields are ignored by the LFS layer. Several of the fields appear to be obsolete (and unused).

Note: For fields holding time values, Transarc's Episode only stores the *.sec* portions of these times on-disk. The *.usec* field is set to zero on get info requests.

The above fields have the following meanings.

.creationDate The time, set by the LFS, at which this fileset was created.

.updateDate Set by the LFS to the current time whenever a fileset's status is changed. This includes the creation and deletion of files as well as whenever the volume version is incremented.

Note: For performance reasons, there is freedom in the granularity of updates to this field (e,g,: every second or couple of seconds). Therefore, this field need not be absolutely accurate.

.accessDate Although the original intent of this field is unclear, under Transarc's Episode it is updated at exactly the same instants that the above *.updateDate* field is. (Exception: at fileset creation, via *ag_volCreate*( ), these two fields can initially be set to two different values, although they'll synchronize themselves almost immediately.) Unless or until the meaning of these fields changes in the future, it should be possible to implement them as a single on-disk field.

Note: Transarc considers it unlikely that the *.accessDate* field will ever be given a distinct meaning.

.backupDate Appears to be currently unused.

.copyDate When a fileset is restored, this field is set to the time at which the dump (or backup) from which it is restored was originally taken.

.volversion This field is incremented by the LFS whenever any change is made to any file on the fileset (to either the status or data portion of the file).

Note: It appears this field is not updated for fileset status updates.

.backupId The ID of any .backup (VOL_BACKUP, created via "fts clone") fileset within the fileset group of which this one is a member.

.cloneId The ID of any .readonly (VOL_READONLY, release or scheduled replicas) filesets within the fileset group of which this one is a member.

.llBackId This field, and the one following are related. If non-zero, this field identifies the fileset backing that is backing this one.

.llFwdId This field and the preceeding one maintain the linked list that maintains the backing relationship between cloned filesets. If non-zero, this field identifies the immediate fileset that this one is backing. Refer to Section 15.3 on page 328 for further details if desired.

Note: The next four quota related fields are more thoroughly described in the section, *Fileset quotas*, Section 15.7 on page 333. When the Limit values are initially presented to the file system by a *vol_setattr*( ) or *ag_volCreate*( ) call, they should be rounded up to cover an integral number of internal "allocation units". The values returned by *vol_getattr*( ) should reflect these rounded up values and can, therefore, differ from what was originally specified.

If, to a *vol_setattr*( ) or *ag_volCreate*( ) operation, a quota limit is supplied that is larger than the maximum value that can be represented internally, the maximum value that can be held internally should be used instead.

Refer to the section *Fileset Quotas*, Section 15.7 on page 333.

The following continue to define the meanings of the fields in the **struct vol_stat_dy** structure.

.allocLimit	The maximum size, in bytes, that a fileset's "allocated" usage can grow to.
.allocUsage	The "allocated" measure, in bytes, of the amount of disk space being used by the fileset.
.visQuotaLimit	The maximum size, in bytes, that a fileset's "visible" usage can grow to.
.visQuotaUsage	The "visible" measure, in bytes, of the amount of disk space being used by the fileset.
.fileCount	Currently unused.
.minQuota	Currently unused.
.owner	Currently unused.
.unique	Episode uses this per- fileset counter for the maintenance of inode generation numbers. Other LFS implementations may or may not use this field. Consult the section on *Inode Generation Numbers*, Section 15.8 on page 334, for further information.
.index	The index of this fileset on the aggregate. This value has the same interpretation and significance as that used by the aggregate *ag_volInfo*( ) operation. See Section 14.4 on page 312 if desired.
.rwIndex	Currently unused.
.backupIndex	Currently unused.
.cloneIndex	Currently unused.
.parentIndex	This field is always returned as 0 by Transarc's Episode (although there appears to be latent support for it). **Note:** It appears that this field was or is intended to provide the index (see the *.index* field above) of the fileset that is backing this one. (Not the index of the *vol_stat_st .parentID* fileset.)
.nodeMax	The canonical (based on VOL_ROOTINO) index of the highest inode (its index, that is) allocated within the fileset. (This field indicates the size of the dynamically allocated inode table.) There does not appear to be any use of this field.
.aggrId	The Id of the aggregate on which this fileset resides.
.tag	Currently unused.
.msgLen	Currently unused.
.statusMsg	A status message -- *not* necessarily null terminated.

13.3.6 Define the vol_status Structure

The following structure defines the combined static and dynamic portions of a fileset's status.

```
struct vol_status {
    struct vol_stat_st vol_st;
    struct vol_stat_dy vol_dy;
};
```

13.3.7 Define the vol_statusDesc Structure

The following structure defines the status description structure used by such operations as *vol_bulksetstatus*( ).

```
#define VOL_MAX_BULKSETSTATUS    3

typedef struct vol_statusDesc {
    union {
    int            volDesc;
    struct volume  *volp;
    }              vsd_volId;
    u_long         vsd_mask;
    u_long         vsd_spare;
    struct vol_status  vsd_status;
} vol_statusDesc_t;
#define vsd_volDesc    vsd_volId.volDesc
#define vsd_volp       vsd_volId.volp
```

13.3.8 Define Transient Error

The following macros determine if an error is transient or persistent.

```
#define VOL_ERROR_IS_TRANSIENT(error) (error < VOLERR_TRANS_HIGHEST && \
     error > VOLERR_TRANS_LOWEST)
```

13.3.9 Volume States (for vol_stat_st)

The following general status flags are for the state field of the **struct vol_stat_st** defined in Section 13.3.4 on page 277. Following these definitions is an indication of which are stored in on-disk, permanent storage, in Section 13.3.10 on page 283.

```
/* These really belong to v_voltypes */
#define VOL_TYPEBITS      0x00003 /* low-level type of volume this is */
#define VOL_RW            0x00001 /* R/W volume */
#define VOL_READONLY      0x00002 /* ReadOnly Volume */

/* Some of these (that is, VOL_OK) are composites */
#define VOL_BUSY          0x00004 /* Volume is Busy */
/* The following three definitions are obsolete */
#define VOL_OFFLINE       0x00008 /* Volume is offline */
#define VOL_DELONSALVAGE  0x00010 /* Delete On Salvage */
#define VOL_OUTOFSERVICE  0x00020 /* Out Of service */
#define VOL_DEADMEAT      0x00040 /* About to be deleted */
#define VOL_LCLMOUNT      0x00080 /* Volume is mounted locally */

/* replication-specific state bits */
```

```
#define VOL_REPFIELD      0x00f00 /* which replication created this? */
#define VOL_REP_NONE      0x00000 /* none */
#define VOL_REP_RELEASE   0x00100 /* ''vos release'' */
#define VOL_REP_LAZY      0x00200 /* lazy replication */
#define VOL_IS_COMPLETE   0x01000 /* this volume instance is complete */
#define VOL_HAS_TOKEN     0x02000 /* our VVage should always be zero. */
#if 0
/* no longer used -- can be reclaimed for some other purpose */
#define VOL_KNOWDALLY     0x04000 /* know reclaimDally val-this vol */
#endif
#define VOL_NOEXPORT      0x08000 /* Do not export this volume! */

#define VOL_TYPEFIELD     0xf0000 /* what high-lvl type vol this is */
#define VOL_TYPE_RW       0x10000 /* read-write (ordinary) */
#define VOL_TYPE_RO       0x20000 /* ''.readonly'' */
#define VOL_TYPE_BK       0x30000 /* ''.backup'' */
#define VOL_TYPE_TEMP     0x40000 /* temporary use, dumps, moves, etc. */

#define VOL_GRABWAITING   0x100000 /* grab is pending for this volume */
#define VOL_LOOKUPWAITING 0x200000 /* lookup is pending for this volume */
#define VOL_REPSERVER_MGD 0x400000 /* managed by repserver */
#define VOL_MOVE_TARGET   0x800000 /* this vol is target for move op */
#define VOL_MOVE_SOURCE   0x1000000 /* this vol is source for move op */
#define VOL_ZAPME         0x2000000 /* delete vol with extreme prejudice */
#define VOL_CLONEINPROG   0x4000000 /* this vol is partially cloned */
#define VOL_IS_REPLICATED 0x8000000 /* this vol is replicated */
#define VOL_OPENDONE      0x10000000 /* back from VOL_OPEN procedure */
#define VOL_OK  ~(VOL_BUSY|VOL_OFFLINE|VOL_DELONSALVAGE| \
            VOL_OUTOFSERVICE| VOL_ZAPME|VOL_CLONEINPROG| \
            VOL_OPENDONE)
```

13.3.10 On-disk States

The following set of state definitions is stored in on-disk, permanent storage. All others are not stored on-disk for the fileset (and hence, not seen by the LFS layer).

VOL_RW

VOL_READONLY

VOL_DELONSALVAGE

VOL_IS_COMPLETE

VOL_NOEXPORT

VOL_REPSERVER_MGD

VOL_MOVE_TARGET

VOL_MOVE_SOURCE

VOL_ZAPME

VOL_CLONEINPROG

VOL_IS_REPLICATED

VOL_REPFIELD

VOL_REP_NONE

VOL_REP_RELEASE

VOL_REP_LAZY

VOL_TYPEFIELD

VOL_TYPE_RW

VOL_TYPE_RO

VOL_TYPE_BK

VOL_TYPE_TEMP

13.3.11 Kernel Maintained State Bits

These are the bits that the kernel maintains alone, not by VOLOP_SETSTATUS.

```
#define VOL_BITS_NOSETSTATUS (VOL_BUSY|VOL_DEADMEAT|VOL_GRABWAITING| \
            VOL_LOOKUPWAITING|VOL_DELONSALVAGE| \
            VOL_OPENDONE)

/* Don't turn off VOL_DELONSALVAGE when attaching a volume.  It is a    */
/* kernel-maintained bit that is also persistent, unlike the other bits */
/* in VOL_BITS_NOSETSTATUS                                              */

#define VOL_BITS_INITIALMASK (VOL_BITS_INITIAL | (VOL_BITS_NOSETSTATUS \
            & ~VOL_DELONSALVAGE))

/* During an identity swap, preserve these bits in their original struct
 * volume, because any waiters will be waiting on the address of that
 * structure's v_states field. */

#define VOL_BITS_NOSWAP  (VOL_GRABWAITING|VOL_LOOKUPWAITING)
```

13.3.12 Useful Macros in Volume Pointers

```
#define VOL_READWRITE(volp) ((volp)->v_stat_st.states & VOL_RW)
#define VOL_EXPORTED(volp)  (!((volp)->v_stat_st.states & VOL_NOEXPORT))
```

13.3.13 Mask Bits for VOL_SETSTATUS MASK Argument

The setstatus operations take a mask that specifies the set of status fields to be updated. Bits within this mask are defined as follows.

```
#define VOL_STAT_VOLNAME        0x00000001
#define VOL_STAT_VOLID          0x00000002
#define VOL_STAT_VERSION        0x00000004
#define VOL_STAT_UNIQUE         0x00000008
#define VOL_STAT_OWNER          0x00000010
#define VOL_STAT_TYPE           0x00000020
#define VOL_STAT_STATES         0x00000040
#define VOL_STAT_ALLOCLIMIT     0x00000080
#define VOL_STAT_BACKUPID       0x00000100
```

```
#define VOL_STAT_PARENTID       0x00000200
#define VOL_STAT_CLONEID        0x00000400
#define VOL_STAT_CREATEDATE     0x00000800
#define VOL_STAT_UPDATEDATE     0x00001000
#define VOL_STAT_ACCESSDATE     0x00002000
#define VOL_STAT_COPYDATE       0x00004000
#define VOL_STAT_NODEMAX        0x00008000
#define VOL_STAT_VISLIMIT       0x00010000
#define VOL_STAT_MINQUOTA       0x00020000
/* spare VOL_STAT_SIZE          0x00040000 */
#define VOL_STAT_INDEX          0x00080000
#define VOL_STAT_BACKVOLINDEX   0x00100000
#define VOL_STAT_STATUSMSG      0x00200000
#define VOL_STAT_CLONETIME      0x00400000
#define VOL_STAT_VVCURRTIME     0x00800000
#define VOL_STAT_VVPINGCURRTIME 0x01000000
/* spare VOL_STAT_ACCERROR      0x02000000 */
#define VOL_STAT_BACKUPDATE     0x04000000
#define VOL_STAT_RECLAIMDALLY   0x08000000
#define VOL_STAT_LLBACKID       0x10000000
#define VOL_STAT_LLFWDID        0x20000000
#define VOL_STAT_VOLMOVETIMEOUT 0x40000000
```

13.3.14 Define the Fileset (volume) Structure

This structure is a memory resident structure maintained for each fileset. It is allocated by the file system independent layer (in the xvolume manager) at fileset attach time (*vol_attach*( )0 and is passed as an argument to each of the fileset operations.

```
/*
 * In-core volume structure
 */

struct volume {
    struct squeue       v_lruq;    /* lruq + free list queue */
    struct volume       *v_next;
    struct aggr         *v_paggrp; /* points to aggregate this */
                                   /* fileset belongs to */
    struct volumeops    *v_volOps; /* points to volumeops vector */
    struct lock_data    v_lock;    /* lock variable
    long                v_count;   /* ref count for storage */
    opaque              v_fsDatap; /* points to file sys dependent */
                                   /* PRIVATE data */
    struct vol_stat_st  v_stat_st;
};
```

13.3.15 Define Volume Fields as v_*

By defining all volume's fields as v_* they can be freely moved around without changing the code all over.

```
/* stuff in vol_stat_st */
#define v_volId              v_stat_st.volId
#define v_parentId           v_stat_st.parentId
#define v_cloneTime          v_stat_st.cloneTime
#define v_vvCurrentTime      v_stat_st.vvCurrentTime
#define v_vvPingCurrentTime  v_stat_st.vvPingCurrentTime
#define v_voltype            v_stat_st.type
#define v_accStatus          v_stat_st.accStatus
#define v_accError           v_stat_st.accError
#define v_concurrency        v_stat_st.concurrency
#define v_states             v_stat_st.states
#define v_reclaimDally       v_stat_st.reclaimDally
#define v_volName            v_stat_st.volName
#define v_activeVnops        v_stat_st.activeVnops
#define v_procID             v_stat_st.procID
```

13.3.16 Define the vol_Dirent Structure

This is the structure used by *vol_readdir*( ) and *vol_appenddir*( ). See Appendix H on page 403 for information about the fields within this structure.

```
typedef struct vol_dirent {
    int32    offset;      /* Logical directory offset */
    int32    vnodeNum;    /* Canonical vnode number */
    u_int32  codesetTag;  /* Tag for the name's codeset */
    u_short  reclen;      /* Len of this rec, 4-byte boundary */
    u_short  namelen;     /* Length of 'name' */
    char     name[OSI_MAXNAMLEN+1]; /* File name, variable sized */
} vol_dirent_t;
```

13.3.17 Define the Fileset Error Codes

The following define the fileset related error codes. Currently these overlay the errno error space; in fact, they are returned from the **afs_syscall** interface via errno.

```
#define  VOL_ERR_EOF        1
#define  VOL_ERR_DELETED    2
#define  VOL_ERR_EOW        8
```

13.3.18 Define the Fileset Open Operation Types

When an administrative, user-space utility *open*s a fileset, it supplies a bitmask that indicates the types of operations that will be performed on the open fileset. This is used in order to determine the class of concurrent operations that will be permitted on the fileset while it is open. Bits within this mask are defined as follows.

```
/*
 * VOLOP bit definitions
 */

#define VOL_OP_SEEK          0x00000001
```

```
#define VOL_OP_TELL          0x00000002
#define VOL_OP_SCAN          0x00000004
#define VOL_OP_DESTROY       0x00000008
#define VOL_OP_GETSTATUS     0x00000010
#define VOL_OP_SETSTATUS     0x00000020
#define VOL_OP_CREATE        0x00000040
#define VOL_OP_READ          0x00000080
#define VOL_OP_WRITE         0x00000100
#define VOL_OP_TRUNCATE      0x00000200
#define VOL_OP_DELETE        0x00000400
#define VOL_OP_GETATTR       0x00000800
#define VOL_OP_SETATTR       0x00001000
#define VOL_OP_GETACL        0x00002000
#define VOL_OP_SETACL        0x00004000
#define VOL_OP_CLONE         0x00008000
#define VOL_OP_RECLONE       0x00010000
#define VOL_OP_UNCLONE       0x00020000
#define VOL_OP_SETNEWVID     0x00040000
#define VOL_OP_COPYACL       0x00080000
#define VOL_OP_SWAPIDS       0x00100000
/* now, for the piece of SETSTATUS that changes the fileset ID */
#define VOL_OP_SETSTATUS_ID  0x00200000
#define VOL_OP_SYNC          0x00400000
#define VOL_OP_PUSHSTATUS    0x00800000
#define VOL_OP_READDIR       0x01000000
#define VOL_OP_APPENDDIR     0x02000000
#define VOL_OP_GETZLC        0x04000000
/* Forcibly (architecturally) deny concurrent vnode ops */
#define VOL_OP_NOACCESS      0x08000000
#define VOL_OP_GETNEXTHOLES  0x10000000
#define VOL_OP_DETACH        0x20000000
#define VOL_OP_DEPLETE       0x40000000
```

13.3.19 Define the Fileset Operations VOL_SYS_XXX

```
/*
 * VOL syscall to VOLOP translation
 * Translates syscalls to volop collections;
 * Depends on xvolume/vol_init.c implementation.
 */

#define VOL_SYS_SEEK          (VOL_OP_SEEK)
#define VOL_SYS_TELL          (VOL_OP_TELL)
#define VOL_SYS_SCAN          (VOL_OP_SCAN)
#define VOL_SYS_DESTROY       (VOL_OP_DESTROY)
#define VOL_SYS_GETSTATUS     (VOL_OP_GETSTATUS)
#define VOL_SYS_SETSTATUS     (VOL_OP_SETSTATUS)
#define VOL_SYS_CREATE        (VOL_OP_CREATE)
#define VOL_SYS_READ          (VOL_OP_READ)
#define VOL_SYS_WRITE         (VOL_OP_WRITE)
#define VOL_SYS_TRUNCATE      (VOL_OP_TRUNCATE)
#define VOL_SYS_DELETE        (VOL_OP_DELETE)
#define VOL_SYS_GETATTR       (VOL_OP_GETATTR)
```

```
#define VOL_SYS_SETATTR       (VOL_OP_SETATTR)
#define VOL_SYS_GETACL        (VOL_OP_GETACL)
#define VOL_SYS_SETACL        (VOL_OP_SETACL)
#define VOL_SYS_CLONE         (VOL_OP_CLONE)
#define VOL_SYS_RECLONE       (VOL_OP_RECLONE)
#define VOL_SYS_UNCLONE       (VOL_OP_UNCLONE)
#define VOL_SYS_SETVV         (VOL_OP_GETSTATUS | VOL_OP_SETSTATUS)
#define VOL_SYS_SWAPVOLIDS    (VOL_OP_SWAPIDS)
#define VOL_SYS_COPYACL       (VOL_OP_COPYACL)
#define VOL_SYS_SYNC          (VOL_OP_SYNC)
#define VOL_SYS_PUSHSTATUS    (VOL_OP_PUSHSTATUS)
#define VOL_SYS_READDIR       (VOL_OP_READDIR)
#define VOL_SYS_APPENDDIR     (VOL_OP_APPENDDIR)
#define VOL_SYS_GETZLC        (VOL_OP_GETZLC)
#define VOL_SYS_GETNEXTHOLES  (VOL_OP_GETNEXTHOLES)
/* Forcibly (architecturally) deny concurrent vnode ops */
#define VOL_SYS_NOACCESS      (VOL_OP_NOACCESS)
#define VOL_SYS_DETACH        (VOL_OP_DETACH)
#define VOL_SYS_DEPLETE       (VOL_OP_DEPLETE)
```

13.3.20 Sync Type Values for vol_sync()

```
#define VOL_SYNC_COMMITSTATUS  1
#define VOL_SYNC_COMMITMETA    2
#define VOL_SYNC_COMMITALL     3
```

13.3.21 Define the Fileset Handle Structure

The following is the internal volume handle for all VOP_* calls. It is an in-kernel structure only; a volume descriptor (long) is passed to the user space to associate it with this structure.

There, as an administrative utility performs actions on an open fileset, certain information regarding its progress (namely which file on the fileset is being processed) is recorded in a **vol_handle** structure. This is passed as an argument to several of the fileset operations.

The *.fid*, *.index* and *.type* fields below are not interpreted outside of the LFS layer.

```
struct vol_handle {
    struct volume *volp;  /* pointer to incore volume */
    long opentype;        /* Open volume type */
                          /* Mask of VOL_OP_XXX bits spec- */
                          /* ified at vol_open(0 time and  */
                          /* gives the type ops to perform */
    struct afsFid fid;    /* Fid for "active" volume anode */
    long index;           /* active Anode's index */
                          /* set by vol_open(), vol_seek(),*/
                          /* vol_scan() and vol_create().  */
    long type;            /* active Anode's mode/type */
};
```

13.3.22 Define the Root Anode Index

The following represents the special index for the volume's root anode. The special dummy index is required since various physical file systems use different numbers for their root anode (ufs = 2, episode = 15, etctera). Do *not* change its definition since code depends on it (for instance, scanning -1, 0, 1 ... to maxino is assumed by various programs). Ordinary files are indicated by indices of 0, 1, 2, and so forth.

```
#define VOL_ROOTINO    -1
```

13.3.23 Define the Maximum Quota Size

Define the largest 64-bit constant that is evenly divisible by 64K and whose value will fit in 32 bits when divided by 1024. This is the maximum value allowed by LFS when the aggregate fragment size is 1K (the smallest frag. size allowed); although larger quota values are possible when the fragment size is increased. The value must also be evenly divisible by 64K to prevent rounding errors when converting to units of 64K (or less), the largest fragment size allowed by LFS.

The following 64 bit number is approximately 4.4 terabytes.

```
#define VOL_MAX_QUOTA_HIGH   0x3ff
#define VOL_MAX_QUOTA_LOW    0xffff0000
```

13.4 Extended Credential Data Types

This section will define the data types exported by the credential package described in *The xcred Package*, Section 17.1 on page 469.

13.4.1 Define the xcred_PListEntry_t Structure

This structure is used to build a list of all attribute-value pairs for an **xcred**, including derivation relationships. It is used by *xcred_DeleteEntry*( ). Section 17.3 on page 484.

```
/* Property list entry.  */

typedef struct xcred_PListEntry {
    struct xcred_PListEntry *nextP; /* Next entry in linear plist */
    struct xcred_PListEntry *prevP; /* Prev entry in linear plist */
    long attrBytes;   /* Size of attr (incl. null) */
    char *attributeP; /* Ptr to attr. component */
    long valueBytes;  /* Size of value component */
    char *valueP;     /* Ptr to value component */
    long flags;       /* Internal flags */
} xcred_PListEntry_t;
```

13.4.2 Define the xcred_t Structure

This refcounted and lockable structure contains all the extended credential info associated with a given user, stored in a property list. Also stored is information on the UNIX cred with which this structure is associated. Included is a generation number, incremented each time someone changes the property list.

```
/* Xcred structure.  */

typedef struct xcred {
    struct squeue lruq;    /* LRU queue */
    struct lock_data lock; /* Structure lock */
    long refcount;         /* Reference count */
    long changeCount;      /* # times plist changed */
    long flags;            /* Flags described above */
    long uflags;           /* More user-defined flags */
    xcred_PListEntry_t *propListP; /* Property list */
    struct ucred *assocUCredP; /* Assoc UNIX cred struct */
    long assocPag;         /* Unique identifier in ucred */
} xcred_t;
```

13.4.3 Reserved Attributes

The following parts of the attribute space are reserved for the following purposes. Other non-conflicting names may be defined by users.

- *NFSId* contains a long integer representing the user's identity according to an unauthenticated NFS call.

13.5 The VFS+ Switch

13.5.1 DCE LFS VNOPS Vector Organization

Example 13-1 Initializing the Extended VNOPS Vector

```
/*
 * The extended VNOPS vector is initialized as follows:
 *
 *    OOPS  point to xglue_*
 *    XOPS  point to the passed in 'axfuns'
 *    NOPS  point to the passed in 'ofuncs'
 *
 * The CM calls   with 'axfuns' pointing to the CM VNOPS;
 * EPISODE calls  with 'axfuns' pointing to the EFS VNOPS;
 * Everyone calls with 'ofuncs' pointing to n<os>_ops.
 *
 * This results in an extended VNOPS vector such that
 * VOP_xxx calls on vnodes in the AFS VFS generate a
 * call chain like this:
 *
 *      VOP_xxx
 *        xglue_xxx()     * does tkc synchronization *
```

```
 *          n<os>_xxx()   * maps VFS to VFS+         *
 *            cm_xxx()
 *
 * VOP_xxx calls on vnodes in the LFS VFS generate a
 * call chain like this:
 *
 *      VOP_xxx
 *        xglue_xxx()     * does tkc synchronization *
 *          n<os>_xxx()   * maps VFS to VFS+         *
 *            efs_xxx()
 */
```

13.5.2 LFS Generating Extended Vnode Operations

Example 13-2 Generating New Style Vnode Operations From Original

```
/*
 * Used by GetNewVnodeOpsFromOld (below) to generate new style vnode
 * operations from original-style vnode operations.
 * In other words, this is used by ufs (not episode or cm) to
 * get extended vnode structures for its vnodes.
 *
 * We initialize the extended VNOPS vector as follows:
 *
 *      OOPS  point to xglue_*
 *      XOPS  point to xufs_*
 *      NOPS  point to the passed in 'aofuns'
 *
 * The PX calls xvfs_convert(vp) to convert a vnode to a vnode
 * with the extended vnode ops.  This results in a call
 * to this routine, where 'aofuns' points to ufs vnops or
 * efs vnops, depending on which VFS was referenced by PX.
 *
 * This results in an extended VNOPS vector such that
 * VOP_xxx calls on vnodes in the AFS VFS generate a
 * call chain like this:
 *
 *      VOP_xxx
 *        xglue_xxx()     * does tkc synchronization *
 *          ufs_xxx()
 *
 * or
 *
 *      VOP_xxx
 *        xglue_xxx()     * does tkc synchronization *
 *          nosf_xxx()    * maps VFS to VFS+         *
 *            efs_xxx()
 *
 *
 * Calls via VOPX (for example, by PX) generate a call chain like this:
 *
 *      VOPX_xxx
```

```
 *        efs_xxx()
 *
 * or
 *
 *      VOPX_xxx
 *        xufs_xxx()      * maps VFS+ to VFS         *
 *          ufs_xxx
 */
```

13.5.3 Converting Old Vnode Operations

Example 13-3 GetNewVnodeOpsFromOld Routine

```
/*
 * Enter some old vnode ops into a conversion hash table, which is
 * just a linked list for now.
 */
static struct xvfs_vnodeops *GetNewVnodeOpsFromOld(
    struct osi_vnodeops *aoldOpsp)
{
    register struct xvfs_opspair *tp;
    register struct xvfs_vnodeops *np;

    /* Look for the vnode ops and see if we've already done a conversion
     * from this ops.  If so, return it.
     *
     * Also, we look to see if these vnode ops are already converted.
     * If they are, we just return the same ops pointer.  This
     * path is taken when UFS recycles an inode: it clears v_flag (which
     * clears the converted flag), but doesn't reset the vnode ops pointer.
     * We *really* don't want to convert the vnode a second time.
     */
    for (tp = xvfs_AllOpsPairs; tp; tp = tp->next) {
        if (aoldOpsp == (struct osi_vnodeops *) tp->newOps)
            return (struct xvfs_vnodeops *) aoldOpsp;

        if (aoldOpsp == tp->oldOps)
            return tp->newOps;
    }
```

13.6 Vnode Data Types

13.6.1 Preliminary Items

There is a preliminary check that must be made for Vnode Operations, as follows.

```
#ifndef  TRANSARC_XVNODE_H
#define  TRANSARC_XVNODE_H
#include <dcedfs/volume.h>
#include <dcedfs/common_data.h>
#include <dcedfs/osi.h>
#include <dcedfs/osi_cred.h>

#ifdef   AFS_DEFAULT_ENV
#error   "Check that xvfs_genvnode.h is right for you"
#include <dcedfs/xvfs_genvnode.h> /* std VFS systems */
#endif   /* AFS_DEFAULT_ENV */
```

13.6.2 Converted Vnode Indication

The status flag described in Section C.1 on page 245 is defined here, along with it's relatives, as follows.

```
/* Exported structures and functions are preceded by module
 * name xvfs_ */

/*
 * Define constant used in v_flag field to indicate that the vnode
 * has already been converted.
 */

#ifndef AFS_OSF_ENV
#define V_CONVERTED   0x1000    /* looks out of the way for now */
#endif

#ifdef  AFS_AIX31_ENV
#define IS_CONVERTED(vp)     ((vp)->v_gnode->gn_flags & V_CONVERTED)
#define SET_CONVERTED(vp)    (vp)->v_gnode->gn_flags |= V_CONVERTED
#define SET_UNCONVERTED(vp)  (vp)->v_gnode->gn_flags &= ~V_CONVERTED
#else
#define IS_CONVERTED(vp)     ((vp)->v_flag & V_CONVERTED)
#define SET_CONVERTED(vp)    (vp)->v_flag |= V_CONVERTED
#define SET_UNCONVERTED(vp)  (vp)->v_flag &= ~V_CONVERTED
#endif
#define XVFS_CONVERT(vp)     (IS_CONVERTED(vp) ? 0 : xvfs_convert(vp))
#define xvfs_PutVolume(volp) if (volp) VOL_RELE(volp)
```

13.6.3 Define the xvfs_attr Structure

The following defines additional attribute fields that have special meaning in DFS; used by both Episode and UFS. The Fields flagged with an "/* on disk */" must be stored in permanent, on-disk storage for the file.

```
struct Txvattr {
    afsHyper dataVersion;  /* on disk */
    afsHyper fileID;       /* on disk */
    afsHyper volVersion;   /* on disk */
    u_long author;
    u_long callerAccess;
    u_long anonAccess;
    u_long parentVnode;
    u_long parentUnique;
    afsTimeval serverModTime;
    u_long fstype;  /* type of the containing aggregate, from aggr.h */
    /* Values that are principally derived from what Episode stores
       for DFS */
    afsUUID    objid;      /* on disk */
    u_long  timeUncertainty; /* on disk */
    /* for Episode, the CFLAGS from epia_GetInfo (anode.h),
      including EPIA_CFLAGS_COPYONWRITE */
    u_long  representationFlags;
    /* Magic cookies usable to identify common ancestry. */
    u_long  backingIndex;    /* on disk */
    u_long  backingVolIndex; /* on disk */
    /*
     * These Ix values are cookies that are interpretable only by the
     * underlying representation.
     */
    u_long  aclIx;           /* on disk */
    u_long  initDirAclIx;    /* on disk */
    u_long  initFileAclIx;   /* on disk */
    u_long  plistIx;         /* on disk */
    u_long  uPlistIx;        /* on disk */

    u_long    clientOnlyAttrs; /* rsrvd for machines that never run FXs */
                               /* on disk */ /* clientOnlyAttrs */
    u_long    spare1;
    u_long    spare2;
    u_long    spare3;
    u_long    spare4;
    u_long    spare5;
    u_long    spare6;
};
```

These extended attributes have the following meanings.

.dataVersion This 64-bit version number increases monotonically whenever the file's contents change (whenever the **mtime** is updated). Its advantage over the **mtime** is that it will never go backwards.

Note: The value in this field is used by the DFS cache client (CM).

.fileID The inode index and inode generation numbers for the file, in that order. These are the same values that would reside in an **afsFid** naming the file.

.volVersion Whenever the volume version number is incremented due to a change to this file, that (incremented) volume version is stored with the file.

.author This unused field is set to –1 by the *vn_getattr*( ) and *vol_getattr*( ) calls.

.callerAccess This **permset_t** (see the DFS **ACL** specification, **ACL** *Storage Format*, Chapter 9 on page 165) is set to the rights that the caller of the *vn_getattr*( ) or *vol_getattr*( ) call has to the file.

.anonAccess This **permset_t** is equal to the set of rights that an object's **ACL** grants to everyone, even unauthenticated clients. If an object does not have an **ACL**, this field will be 0. See the DFS **ACL** specification mentioned above, and the description of *vn_getattr*( ) in Section 16.7 on page 432. This field is used by the client cache manager in some situations to avoid having to perform a remote **ACL** check.

.parentVnode This unused field is set to –1 by the *vn_getattr*( ) and *vol_getattr*( ) calls.

.parentUnique This unused field is set to –1 by the *vn_getattr*( ) and *vol_getattr*( ) calls.

.serverModTime This field should be set to the file's modified time (mtime) by the *vn_getattr*( ) and *vol_getattr*( ) calls.

.fstype This field is set to the appropriate AP_TYPE_XXX constant for the LFS by the *vn_getattr*( ) and *vol_getattr*( ) calls.

.objid The object UUID for this file.

.timeUncertainty This field is currently unused; the *vol_getattr*( ) and *vn_getattr*( ) calls should return 0 for it and the *vol_setattr*( ) and *vn_setattr*( ) calls should ignore it. It was intended to hold DTS time uncertainties to accompany the file's atime, mtime and ctime.

.representationFlags
This field is set to a set of internal, LFS-dependent state flags.

Note: These are a set of internal Episode flags. It is not known yet whether they have any DFS (non-diagnostic) use and hence require architecting.

The following two fields will only be non-zero for a cloned file. They identify the *backing* file for files in a copy-on-write relationship. For example: if F2 is a read-only clone of the read-write fileset F1 (F1 ==> F2), (meaning F1 is backed by F2), these fields would be zero for F2 and non-zero for F1.

Note: It is not known whether these fields serve any actual purpose or are there for diagnostic (or obsolete) reasons. If the former, it's not clear to what degree a particular LFS has to agree with what Episode puts here.

.backingIndex The index of the fileset which is holding the backing, copy-on-write file for this one. This index has the same interpretation (and significance) as the one used by the *ag_volinfo*( ) aggregate call. Note that filesets related by cloning must reside on the same aggregate.

.backingVolIndex The index of the file which is serving as the backing, copy-on-write file for this one. This field has the same interpretation (and significance) as the *.Vnode* field in a file's **afsFid** (or *.fileID*, see above).

In order for fileset moves and backups (both implemented via cloning) to be transparent, a file and its backing file will ALWAYS have the same index.

.clientOnlyAttrs A set of uninterpreted (by the LFS) flags maintained with the file on behalf of DFS.

The following five fields should be set to a non-zero value by the *vn_getattr*( ) and *vol_getattr*( ) calls if the file has an associated auxiliary object of the particular type. Although the precise interpretation of the fields is not architected, it is suggested that they somehow identify or name their auxiliary object such that sharing within a fileset (for example, shared **ACL**s) can be detected via a simple equality check. Obviously, the initial directory and file **ACL** fields are only required for directory objects.

.aclIx Object **ACL**.

.initDirAclIx For a directory: the *Initial Container* (directory) **ACL**.

.initFileAclIx For a directory: the *Initial File* **ACL**.

.plistIx The system property list.

.uPlistIx The user property list.

The "Enhanced" attribute structure: The standard "vattr" structure is simply an overlay (always first) in this structure. This structure consists of both the standard vnode and extended DFS attributes.

```
struct xvfs_attr {
    struct vattr vattr;
    struct Txvattr xvattr;
};
```

13.6.4 Define the Vnode Operations Function Aray

A description of the operations available within the vnode function array appears below. The array is an exported data type.

```
/* extended VOPX operations; note that all attr-taking dudes
 * take xvattrs, not vattrs.
 */
struct xvfs_xops {
    int (*vn_open)();
    int (*vn_close)();
    int (*vn_rdwr)();
    int (*vn_ioctl)();
    int (*vn_select)();
    int (*vn_getattr)();
    int (*vn_setattr)();
    int (*vn_access)();
    int (*vn_lookup)();
    int (*vn_create)();
    int (*vn_remove)();
    int (*vn_link)();
    int (*vn_rename)();
    int (*vn_mkdir)();
    int (*vn_rmdir)();
    int (*vn_readdir)();
    int (*vn_symlink)();
```

```
    int (*vn_readlink)();
    int (*vn_fsync)();
    int (*vn_inactive)();
    int (*vn_bmap)();
    int (*vn_strategy)();
    int (*vn_ustrategy)(); /* assuming stuff already mapped in */
    int (*vn_bread)();
    int (*vn_brelse)();
    int (*vn_lockctl)();
    int (*vn_fid)(); /* op for old style fid op */
    int (*vn_hold)(); /* maybe we don't need these; revisit when done */
    int (*vn_rele)();
    /*
     * new ones for us to provide, rather than just existing to
     * make writing the O functions easier (that is, porting).
     */
    int (*vn_setacl)();
    int (*vn_getacl)();
    int (*vn_afsfid)();
    int (*vn_getvolume)();
    int (*vn_getlength)();
    /*
     * Some new ops for AIX 3
     */
    int (*vn_map)(); /* also used for SunOS 5 */
    int (*vn_unmap)();
    /*
     * A new op for OSF/1
     */
    int (*vn_reclaim)();
    /*
     * Some new ops for SunOS 5
     */
    int (*vn_read)();
    int (*vn_write)();
    int (*vn_realvp)();
    void (*vn_rwlock)();
    void (*vn_rwunlock)();
    int (*vn_seek)();
    int (*vn_space)();
    int (*vn_getpage)();
    int (*vn_putpage)();
    int (*vn_addmap)();
    int (*vn_delmap)();
    int (*vn_pageio)();
#define vn_frlock  vn_lockctl /* overlay equivalent ops */
    /*
     * Ops for HP/UX
     */
    int (*vn_pagein)();
    int (*vn_pageout)();
};
```

13.6.5 Define the Enhanced Vnode Operations

A description of the operations available within the enhanced (converted) vnode function array appears below. The array is an exported data type.

```
/*
 * Operations on the Enhanced vnodes.
 */
struct xvfs_vnodeops {
    struct osi_vnodeops oops;  /* Glued vendor vnodeops */
    struct xvfs_xops xops;     /* Enhanced (Decorum) vnodeops */
    struct osi_vnodeops nops;  /* Original vendor vnodeops */
};
```

The **osi_vnodeops** structure is somewhat of a misnomer. It is the native Operating System (OS) **vnode** vector, which is anything but osi (Operating System Independent).

13.6.6 Define VOPX_XXX for Extended Vnodeops

```
/*
 * Macros for extended vnodeops
 */
#define VOPX_OPEN      efs_null
#define VOPX_CLOSE     efs_null
#define VOPX_RDWR      efs_rdwr
#define VOPX_IOCTL     efs_invalid
#define VOPX_SELECT    efs_invalid
#define VOPX_GETATTR   efs_getxattr
#define VOPX_SETATTR   efs_setxattr
#define VOPX_ACCESS    efs_access
#define VOPX_LOOKUP    efs_lookup
#define VOPX_CREATE    efs_create
#define VOPX_REMOVE    efs_remove
#define VOPX_LINK      efs_link
#define VOPX_RENAME    efs_rename
#define VOPX_MKDIR     efs_mkdir
#define VOPX_RMDIR     efs_rmdir
#define VOPX_READDIR   efs_readdir
#define VOPX_SYMLINK   efs_symlink
#define VOPX_READLINK  efs_readlink
#define VOPX_FSYNC     efs_fsync
#define VOPX_INACTIVE  efs_inactive
#define VOPX_BMAP      efs_bmap
#if defined(AFS_AIX31_VM)
#define VOPX_STRATEGY(VP,BP) efs_strategy (BP)
#else
#define VOPX_STRATEGY  efs_null
#endif
#define VOPX_USTRATEGY efs_panic
#define VOPX_BREAD     efs_panic
#define VOPX_BRELSE    efs_panic
#define VOPX_LOCKCTL   efs_lockctl
#define VOPX_FID       efs_fid
#define VOPX_HOLD      efs_hold
```

```
#define VOPX_RELE      efs_rele
#define VOPX_GETACL    efsx_getacl
#define VOPX_SETACL    efsx_setacl
#define VOPX_AFSFID    efs_afsfid
#define VOPX_GETVOLUME efs_getvolume
#define VOPX_GETLENGTH efs_getlength
#ifdef  AFS_AIX31_ENV
#define VOPX_MAP       efs_map
#define VOPX_UNMAP     efs_unmap
#endif
#ifdef  AFS_OSF_ENV
#define VOPX_RECLAIM   efs_reclaim
#endif
#ifdef  AFS_SUNOS5_ENV
#define VOPX_READ      efs_vmread
#define VOPX_WRITE     efs_vmwrite
#define VOPX_REALVP    efs_realvp
#define VOPX_RWLOCK    efs_rwlock
#define VOPX_RWUNLOCK  efs_rwunlock
#define VOPX_SEEK      efs_seek
#define VOPX_SPACE     efs_space
#define VOPX_GETPAGE   efs_getpage
#define VOPX_PUTPAGE   efs_putpage
#define VOPX_ADDMAP    efs_addmap
#define VOPX_DELMAP    efs_delmap
#define VOPX_PAGEIO    efs_pageio
#define VOPX_FRLOCK    efs_frlock
#endif
#ifdef  AFS_HPUX_ENV
#define VOPX_PAGEIN    efs_pagein
#define VOPX_PAGEOUT   vfs_pageout
#endif
```

13.6.7 Define VOPX_UPDATE Flags

```
/*
 * Flags argument for VOPX_UPDATE
 */
#define XVN_ACC    1    /* atime */
#define XVN_UPD    2    /* mtime */
#define XVN_CHG    4    /* ctime */
```

13.6.8 Define the Enhanced Operations Vector

This is the enhanced (converted) VFS operations vector.

```
/*
 * The extended VFS op vector
 */
struct xvfs_vfsops {
    struct vfsops xvfsops;
    struct vfsops vfsops;
    int (*vfsgetvolume)();
};
```

13.6.9 Define the VFS Operations

The following defines a VFS operations vector.

```
struct vfsops (
    int (*vfs_mount )();
    int (*vfs_unmount )();
    int (*vfs_root )();
    int (*vfs_statfs )();
    int (*vfs_sync )();
    int (*vfs_vget )();
    int (*vfs_getmount )();
};
```

13.6.10 Define the Vnode Operation Classifications

The following definitions classify the **vnode** operations into noop, read-only and read-write.

```
/* Macros to classify vnode ops into noop, read-only and read-write */
#define VNOP_LOCK     VNOP_TYPE_READWRITE
#define VNOP_LINK     VNOP_TYPE_READWRITE
#define VNOP_UNLINK   VNOP_TYPE_READWRITE
#define VNOP_MKDIR    VNOP_TYPE_READWRITE
#define VNOP_RMDIR    VNOP_TYPE_READWRITE
#define VNOP_RENAME   VNOP_TYPE_READWRITE
#define VNOP_SYNCGP   VNOP_TYPE_READWRITE
#define VNOP_TRUNC    VNOP_TYPE_READWRITE
#define VNOP_GETVAL   VNOP_TYPE_READONLY
#define VNOP_RWGP     VNOP_TYPE_READWRITE
#define VNOP_STAT     VNOP_TYPE_READWRITE
#define VNOP_UPDATE   VNOP_TYPE_READWRITE
#define VNOP_OPEN     VNOP_TYPE_READWRITE
#define VNOP_CLOSE    VNOP_TYPE_READWRITE
#define VNOP_READLINK VNOP_TYPE_READWRITE
#define VNOP_SYMLINK  VNOP_TYPE_READWRITE
#define VNOP_BMAP     VNOP_TYPE_READWRITE
#define VNOP_NAMEI    VNOP_TYPE_READWRITE
#define VNOP_MKNOD    VNOP_TYPE_READWRITE
#define VNOP_REMOVE   VNOP_TYPE_READWRITE
#define VNOP_LOOKUP   VNOP_TYPE_READONLY

/*
 * Change MAP and UNMAP to NOOP type vnodeops instead of READWRITE type
 * to prevent MAP and UNMAP vnodeops from blocking on busy filesets
 * as the AIX kernel global shared memory lock is held across these
 * calls. If these vnode ops block, any process trying to start or exit
 * will deadlock on the held shared memory lock
 */
#define VNOP_MAP      VNOP_TYPE_NOOP
#define VNOP_UNMAP    VNOP_TYPE_NOOP

#define VNOP_ACCESS   VNOP_TYPE_READONLY
#define VNOP_GETATTR  VNOP_TYPE_READONLY
#define VNOP_SETATTR  VNOP_TYPE_READWRITE
```

```
#define VNOP_FSYNC    VNOP_TYPE_READWRITE
#define VNOP_FTRUNC   VNOP_TYPE_READWRITE
#define VNOP_RDWR     VNOP_TYPE_READWRITE
#define VNOP_LOCKCTL  VNOP_TYPE_READWRITE
#define VNOP_READDIR  VNOP_TYPE_READWRITE
#define VNOP_GETACL   VNOP_TYPE_READONLY
#define VNOP_SETACL   VNOP_TYPE_READWRITE
#define VNOP_PGRD     VNOP_TYPE_READWRITE
#define VNOP_PGWR     VNOP_TYPE_READWRITE
#define VNOP_CREATE   VNOP_TYPE_READWRITE

/* SunOS relies on VFS_ROOT never blocking. If VFS_ROOT blocks, it holds
 * the vnode mutex which prevents anybody trying to obtain a reference
 * to the vnode to block that leads to a deadlock.
 * HP, AIX and Solaris hold a reference to a filesystems's root vnode always
 * and hence there should not be any problem if VFS_ROOT is unglued.
 */
#ifndef AFS_OSF_ENV
#define VFSOP_ROOT    VNOP_TYPE_NOOP
#else
#define VFSOP_ROOT    VNOP_TYPE_READWRITE
#endif

#define VFSOP_UNMOUNT VNOP_TYPE_READWRITE
#define VFSOP_FHTOVP  VNOP_TYPE_READWRITE
#define VFSOP_VGET    VNOP_TYPE_READWRITE
```

13.6.11 Directory Entry Formats

The following defines the native directory entries of various kinds as shown by the ifdef statements.

```
#ifdef AIX         /* Native dir entry has an offset */
struct dirent {
    u_long              offset;
    u_long              inode;
    u_short             recordlen;
    namelen;
    char                dir_name[];
};
#elif SunOS-5       /* Native dir entry has an offset */
struct dirent {
    u_long              inode;
    u_long              offset;
    u_short             recordlen;
    u_short             namelen;
    char                dir_name[];
};
#else               /* Native dir entries do NOT have an offset */
struct dirent {
    u_long              inode;
    u_short             recordlen;
    u_short             namelen;
```

```
    char                dir_name[];
    u_long              offset;
};
```

The fields have the following meaning:

.offset	An offset corresponding to whatever (empty space or another entry) follows this entry in the directory.
.inode	An LFS-dependent (interpreted only by the LFS) anode number.
.recordlen	Length of this entry, rounded up to a 4 byte boundary. Therefore, the offset from the start of this entry to the next one in a supplied buffer.
.namelen	Length of the file name, not including the terminating null char.
.dir_name	Variable sized file name: includes a terminating null char.

Chapter 14

Aggregate Operations Interface

This chapter describes the portion of the VFS+ interface which allows operations on aggregates, the **DCE DFS** data containers which house filesets, as introduced in Section 11.3 on page 235. A description of each aggregate known to the local host is kept in the *Aggregate Registry*, a global kernel table. An aggregate descriptor includes a pointer to a **struct aggrops** function array, namely those operations that may be performed on the aggregate. The corresponding aggregate operation is chosen from the function array and executed. The aggregate operation architecture is remarkably similar to that used to provide access to VFS operations. In fact, one implementation technique for these functions is to define an aggregate file system via a **struct vfs**, suitably extended, and use the standard VFS mechanisms to manipulate them.

A description of the operations available within the aggregate function array appears below. The array is presented as an exported data type, and the associated declaration may be found in Section 13.2.12 on page 272. It is reproduced here for convenience.

```
struct aggrops {
    int (*ag_hold)();
    int (*ag_rele)();
    int (*ag_lock)();
    int (*ag_unlock)();
    int (*ag_stat)();
    int (*ag_volCreate)();
    int (*ag_volInfo)();
    int (*ag_detach)();
    int (*ag_attach)();
    int (*ag_sync)();
};
```

14.1 Initialization

Each type or implementation of a DFS LFS file system is identified by a small integer constant within the file **aggr.h** within the DFS source directory **src/file/xaggr**. Section 13.2.4 on page 270, *Valid Aggregate Types* lists the currently defined aggregate types.

14.1.1 Identifying a New LFS to DCE DFS

When a new file system is implemented, a new AG_TYPE_xxx constant must be added to this file in order to identify its type. In this constant, the xxx is the actual identifier of the type of aggregate (and hence, LFS) that is being defined. Additionally, as previously mentioned in Section 11.8.1 on page 240, the constant MAX_AG_TYPE (in Section 13.2.4 on page 270) should be adjusted since this controls the size of the aggregate operations array discussed below.

These aggregate types need not be consistent across multiple vendor platforms. They do not cross the network in any protocols - the aggregate operations are local affairs. It would be beneficial if they were; however, there is no provision currently by which this can be accomplished.

Note: Re-using the aggregate type chosen by Transarc for Episode is not an option.

1. It would preclude running with both Episode and the vendor's LFS.

2. It would probably be illegal.
3. The **backup** and **restore** algorithms assume that when moving between aggregates of the same type, directory offsets can be preserved. Undoubtedly, there are other technical differences that would cause difficulties.

14.1.2 Registering Aggregate Operations

As previously mentioned in the first paragraph of Chapter 14 on page 303 , in the discussion of **Aggregate-level facilities** in Section 11.3 on page 235 and Section 11.8.1 on page 240, there is an *Aggregate Registry* table containing aggregate information that has been made available to **DCE DFS**, by the act of an LFS registering its aggregate operations at initialization time. Registration is made by use of the *ag_setops*( ) function call. The result is that a pointer to the aggregate operations vector just identified is inserted in the *Aggregate Registry* in the appropriate offset identified by the aggregate type identifier. This results in the user space DFS components being informed about a new aggregate type.

The aggregate operations for a particular file system are identified by a **struct aggrops** entrypoint vector identified in Section 13.2.12 on page 272.

14.2 Exporting volumes to DFS

The **dfsexport** command is used to export the filesets contained on an aggregate to DFS. Any aggregates being exported are required to have an entry in the dfstab file (**/opt/dcelocal/var/dfs/dfstab**). Each entry in this file identifies an aggregate and, for Transarc LFS file systems (as opposed to exported UFS file systems), contains the following fields:

Device Name Pathname to the device file for the aggregate.

Aggregate Name A 64-character (byte) aggregate name interpreted by neither DFS nor the LFS.

File System Type The **aggregate type**, or more succinctly, the **file system type** that is housing the aggregate. For an exported UFS, this is "**ufs**". For Transarc's Episode, this is "**lfs**".

For a new implementation of an LFS, a new type string should be declared in the DFS source file **src/file/xaggr/astab.h**. For example, the following type for an appropriate "xxx" value would be added to those defined in Section 13.2.18 on page 274:

```
#define ASTABTYPE_XXX "xxx"
```

Aggregate ID Each aggregate on a machine is identified by an **aggregate ID**. These IDs, which are selected by the system administrator as the dfstab file is constructed, are uninterpreted by DFS/LFS and are simply unique (for that local machine) integers starting from 1.

Note: The format of entries in this table can be different for different aggregate types. For example, an exported UFS has a **dfstab** entry that is different from what is shown above. If necessary (or beneficial), a new lfs could define private fields holding information to be passed to it at export time. Since DFS already has a mechanism for passing LFS-private data to *ag_attach*( ), only the **dfsexport** command itself would require changes in this event. (Specifically: **dfstab.c** and **export.c** in **src/file/xaggr**.)

Appendix G on page 401 lists the steps involved in exporting the filesets in an aggregate.

14.3 Aggregate Mounts

The **UNIX** operating system, in most implementations, only performs file system updates (syncs) to file systems that it finds in the local mount table. This immediately causes a problem for filesets which are exported to DFS but not mounted locally. To overcome this difficulty, DFS performs a special mount of aggregates as they are attached - just to get them into the mount table. Specifically:

- No provision is made for allowing access below these mount points since the only goal is to get an entry into the mount table.
- Aggregates - not individual filesets - are mounted. The mount points used are of the form:

  `/opt/dcelocal/var/dfs/aggrs/<aggregate-name>`

- The mount is performed with a new file system type, MOUNT_AGFS.
- DFS supplies code that implements this file system type (**vfs** and **vnode** operations). A few **vnode** operations are marginally implemented to allow **/bin/ls** operations within the mount directory listed above; most return ENOSYS since they should never be used. The **vfs** operations are implemented on top of the aggregate operations (*ag_xxx*( )) provided by the LFS. They are:

  _root Returns a "synthesized" vnode with the V_ROOT flag set.

  _statfs Calls *ag_stat*( ) to obtain a statfs structure that gives the space usage on the aggregate.

  _sync Calls *ag_sync*( ), with a synctype of AG_SYNC_FILESYS, to actually perform the *sync*.

  _unmount Updates internal tables to reflect the fact that this aggregate is no longer "mounted".

- These "pseudo-mounts" exist independently of any local mounts that might already exist or be created later for filesets on the aggregate.

14.4 Aggregate Array Functions

Descriptions of the members of the **struct aggrops** function array follow. File system independent code above the LFS enforces the requirement that with the exception of *ag_stat*( ), *ag_volInfo*( ) and *ag_sync*( ), these operations can only be issued by ROOT (the local super user).

NAME

ag_hold — Increment the aggregate reference count

SYNOPSIS

```
int ag_hold(
    /* IN */ struct aggr *aggd
    );
```

ARGUMENTS

aggd Aggregate descriptor pointer.

DESCRIPTION

Increment the reference count of the aggregate described by the *aggd* pointer.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

ag_rele — Decrement an aggregate's reference count

SYNOPSIS

```
int ag_rele(
    /* IN */ struct aggr *aggd
    );
```

ARGUMENTS

aggd Aggregate descriptor pointer.

DESCRIPTION

Decrement the reference count of the aggregate associated with the *aggd* pointer. The aggregate reference count is normally incremented by the *ag_hold*( ), *agg_Lookup*( ) and *agg_Attach*( ) functions.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

ag_lock — Lock access to an aggregate

SYNOPSIS

```
int ag_lock(
    /* IN */ struct aggr *aggd,
    /* IN */ int type
    );
```

ARGUMENTS

aggd Aggregate descriptor pointer.

type Type of lock to be obtained. Allowable values are **READ_LOCK**, **WRITE_LOCK** and **SHARED_LOCK**.

DESCRIPTION

Lock access to an aggregate.

DISCUSSION

This operation is currently unused by DFS. The semantics of this function are unspecified. It is implementation-dependent. If implemented, this operation could utilize the standard DFS lock primitives as follows:

```
{
    code = error_status_ok;
        {
        if (type == READ_LOCK) {
            lock_ObtainRead(&aggrp->a_lock);
        }
        else if (type == WRITE_LOCK) {
            lock_ObtainWrite(&aggrp->a_lock);
        }
        else if (type == SHARED_LOCK) {
            lock_ObtainShared(&aggrp->a_lock);
        }
        else code = EINVAL;
        }
    return code;
    }
```

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EINVAL] If *type* is neither READ_LOCK, WRITE_LOCK or SHARED_LOCK.

NAME

ag_unlock — Unlock access to an aggregate

SYNOPSIS

```
int ag_unlock(
    /* IN */ struct aggr *aggd,
    /* IN */ int type
    );
```

ARGUMENTS

aggd	Aggregate descriptor pointer.
type	Type of lock obtainedat call to *ag_lock*( ). Allowable values are **READ_LOCK**, **WRITE_LOCK** and **SHARED_LOCK**.

DESCRIPTION

Unlock access to an aggregate.

DISCUSSION

This operation is currently unused by DFS. The semantics of this function are unspecified. It is implementation-dependent. If implemented, this operation could utilize the standard DFS lock primitives as follows:

```
{
    code = error_status_ok;
        {
        if (type == READ_LOCK) {
            lock_ReleaseRead(&aggrp->a_lock);
        }
        else if (type == WRITE_LOCK) {
            lock_ReleaseWrite(&aggrp->a_lock);
        }
        else if (type == SHARED_LOCK) {
            lock_ReleaseShared(&aggrp->a_lock);
          }
        else code = EINVAL;
        }
    return code;
    }
```

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EINVAL]	If *type* is neither READ_LOCK, WRITE_LOCK, or SHARED_LOCK.

NAME

ag_stat — Returns statistics on the given aggregate

SYNOPSIS

```
int ag_stat(
    /* IN */ struct aggr *aggd,
    /* INOUT */ struct ag_status *astatd
    );
```

ARGUMENTS

aggd	Aggregate descriptor pointer.
astatd	Pointer to an aggregate status structure into which the information will be read.

DESCRIPTION

Provide the caller with the statistics block associated with the aggregate described by **aggd** pointer, placing the results at location **astatd**.

DISCUSSION

Any status fields not explicitely set should be zeroed.

The static portion of the status can be simply copied from the aggrp->a_stat_st to astatp->ag_st . (This static status is maintained in the agregate structure.)

The dynamic portion of the status in astap->ag_dy is computed as appropriate for the particular LFS implementation. Fields with no significance (namely, **minFree**, **spares**) should be zeroed.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

ag_volCreate — Create a new fileset within an aggregate

SYNOPSIS

```
int ag_volCreate(
    /* IN */  struct aggr *aggd,
    /* IN */  struct afsHyper *avolidp,
    /* IN */  struct vol_status *avolstatp,
    /* IN */  long aflags
    );
```

ARGUMENTS

aggd	Aggregate descriptor pointer.
avolidp	Pointer to the ID of the new fileset.
avolstatp	Pointer to a fileset status block with initial values in fields that require them.
aflags	Miscellaneous flag bits. The only bit which has significance is AGGR_CREATE_ROOT.

DESCRIPTION

Create and attach (for DFS export) a fileset within the aggregate associated with the descriptor located at **aggd**. The initial fileset header is derived from the image provided through the **avolstatp** pointer, which has various fields set.

DISCUSSION

The following bit is defined to have significance and is defined as:

```
/* defined bits within the flags argument   */
#define AGGR_CREATE_ROOT    1
```

See Appendix D on page 317 for further information about this function.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EINVAL]	If not exactly one of VOL_RW and VOL_READONLY are set in the initial status **.v_states** field. The new fileset has not been created in this case.
[ENOSPC]	There is not enough space to create the fileset. The fileset has not been created in this case.
[EDQUOT]	The disk quota has been exceeded. The fileset has not been created in this case.

NAME

ag_volInfo — Enumerate the filesets within the given aggregate

SYNOPSIS

```
int ag_volInfo(
    /* IN */  struct aggr *aggd,
    /* IN */  long index,
    /* OUT */ struct volume *vold
    );
```

ARGUMENTS

aggd	Aggregate descriptor pointer that is being enumerated.
index	Index of fileset within the aggregate that is to be examined for this call.
vold	Fileset descriptor into which to copy information about the fileset located at slot **index** within the aggregate.

DESCRIPTION

Determine if there is a fileset at position **index** within the aggregate described by the **aggd** pointer. If there is, put information about it in the fileset descriptor (**vold**). If there is no fileset at position **index**, return **ENOENT**. The first valid fileset index within an aggregate is zero. To enumerate all filesets within an aggregate, *ag_volInfo*( ) is called repeatedly, with **index** starting at zero and incremented at each subsequent call. The iteration completes when *ag_volInfo*( ) returns **VOL_ERR_EOF**.

ag_volInfo( ) can be compared with the fileset operation, *vol_getstatus*( ) (see Section 15.14 on page 355). Both return information about a fileset. But when the former is called, the aggregate is not necessarily attached, and consequently the aggregate op does not look up the fileset in the Fileset Registry, nor make any attempt to gather dynamic information about how the fileset is being used.

DISCUSSION

For information relative to the fields within the fileset structures, see Appendix E on page 321.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[ENOENT]	There is no fileset at the position selected by the *index* argument.
[VOL_ERR_EOF]	All filesets on the aggregate have index values smaller than the one supplied on this call. The calling software should stop looping through the filesets, as they have all been enumerated.

SEE ALSO

vol_getstatus( ).

NAME

ag_detach — Detach the given aggregate

SYNOPSIS

```
int ag_detach(
    /* IN */ struct aggr *aggd
    );
```

ARGUMENTS

aggd	Aggregate descriptor pointer.

DESCRIPTION

Detach the aggregate associated with the **aggd** pointer, taking all its filesets off-line.

DISCUSSION

This call should release, via *osi_Free*( ), any private storage that was allocated by the *af_attach*( ) function. In this event, the aggregate structure (**struct aggr** field *.a_fsDatap*) should be zeroed as well. See Section 13.2.6 on page 270 for information pertinent to the fields in this structure.

Other file system dependent clean up might be required as well, keeping in mind that filesets on the aggregage might still be mounted for local use.

Note: Although Transarc's Episode file system guarantees that an aggregate will not be *ag_detach*( )'d until it is no longer in use (by either DFS or locally mounted filesets), other implementations of the LFS and **mount** and **umount** commands need not behave in this manner.

Consult Appendix F on page 323 for further information.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[ENXIO]	Most failures that are I/O related should be returned with this code.

SEE ALSO

Section 11.8.3 on page 241 defines the signature for the *osi_Free*( ) fP function.

NAME

ag_attach — Attach the given aggregate

SYNOPSIS

```
int ag_attach(
    /* IN */  dev_t dev,
    /* IN */  struct vnode *bdevvP,
    /* IN */  u_long flags,
    /* IN */  caddr_t data,
    /* OUT */ opaque *fsdatap,
    /* OUT */ long *fsdatalen
    );
```

ARGUMENTS

dev	Device representing the partition on which the aggregate is located.
bdevvP	Vnode for the block device of this aggregate that is being attached. The device numer is obtained from this vnode.
flags	Various bits. If the flag AGGR_ATTACH_NOEXPORT is present, this indicates that the aggregate is not actually being exported to DFS (Instead, one of its filesets is being mounted locally).
data	A data area (in user space), holding filesystem-specific data about the aggregate. In **DCE LFS** this parameter is not used. For UFS aggregates it holds the fileset ID of the aggregate's one fileset, and the name of the directory on which the corresponding filesystem is mounted. If present, it should be copied in via *copyin*( ).
fsdatap	Pointer to a place to put a filesystem-specific control block.
fsdatalen	Length of the filesystem-specific control block, for use in freeing it, in case this is done in the course of aborting *ag_Attach*( ) rather than (as is usually the case) by *ag_Detach*( ).

DESCRIPTION

Attach the aggregate specified, putting all its filesets on-line.

DISCUSSION

Consult Appendix F on page 323 for further information.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[ENXIO]	Most failures that are I/O related should be returned with this code. For instance, this failure indicates that the aggregate is in an inconsistent (corrupt) state and needs to be run through a disk salvager (such as FSCK) before it can be accessed.

NAME

ag_sync — Sync the given aggregate

SYNOPSIS

```
int ag_sync(
    /* IN */ struct aggr *aggd,
    /* IN */ int syncType
    );
```

ARGUMENTS

aggd	Aggregate descriptor pointer.
syncType	Describes the type of sync to be done, choose from one of: **AGGR_SYNC_FILESYS**, **AGGR_SYNC_COMMITMETA** or **AGGR_SYNC_COMMITALL**.

DESCRIPTION

Sync the state of the aggregate to the permanent storage, according to the *syncType* parameter:

AGGR_SYNC_FILESYS

All dirty data on the aggregate (and possibly all aggregates of the same type) is (scheduled to be asynchronously) written to permanent storage, consistent with the behavior of **sync**(2). This is useful to periodically flush dirty data for filesets that have no local mount point, and thus, are not affected by the system's periodic calls to **sync**(2).

AGGR_SYNC_COMMITMETA

All dirty meta-data (for instance, file status) is written to permanent storage. The call does not return until the I/O has completed (It is synchronous.).

AGGR_SYNC_COMMITALL

All dirty data (both meta-data and user data) is written to permanent storage. The call does not return until the I/O has completed.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EINVAL]	The **syncType** argument is not one of AGGR_SYNC_FILESYS, AGGR_SYNC_COMMITMETA or AGGR_SYNC_COMMITALL.

Appendix D

Information Pertinent to ag_volCreate( )

In the process of creating and attaching a fileset on an aggregate, the following fields from the supplied initial volume status are stored on disk for the newly created fileset. On-disk fileset status fields not mentioned here should be set to zero.

D.1 Static Status

From the static status portion, statusp->vol_st:

.volName

.volId The volIdp argument is used instead of this.

.parentId

.cloneTime

.states The following fileset state bits are sampled and stored on-disk.

```
VOL_READONLY      \ exactly 1 of these
VOL_RW            /  should be set
VOL_IS_COMPLETE
VOL_DELONSALVAGE
VOL_ZAPME
VOL_CLONEINPROG
VOL_REPFIELD
VOL_TYPEFIELD
VOL_REPSERVER_MGD
VOL_NOEXPORT
VOL_IS_REPLICATED
VOL_MOVE_SOURCE
VOL_MOVE_TARGET
```

D.2 Dynamic Status

From the dynamic status portion, .vol_dy

.creationDate If the seconds (.sec) portion of this is 0, the fileset's creation time is set to the current time.

Observation: although the hi-level Episode code samples this and leaves it in the memory structure, the low-level code seems to ignore it and set the on-disk field to "now".

.updateDate

.accessDate

.copyDate

.volversion

.backupId

.cloneId

.llBackId

.llFwdId

.allocLimit

.visQuotaLimit

.unique Observation: although the hi-level Episode code seems to sample *.unique* and leave it in the memory structure, the low-level code seems to ignore it and set the on-disk field to 0.

.statusMsg

D.3 Other Items

The ID of the new fileset is taken from the volIdp argument.

If the AGGR_CREATE_ROOT flag is specified, a properly initialized root directory (containing "." and "..", owner and group taken from the callers credentials, without ACL or property lists) is created. Otherwise, the fileset is left in an inconsistent state (namely, waiting for a clone or restore) and the VOL_DELONSALVAGE volume state flag should be set in both the supplied volume status structure and the on-disk storage for this fileset.

The .vol_dy.index field in the supplied status structure is set to the value that, if passed as the index argument to the aggregate *ag_volInfo*( ) operation, would select this newly created fileset.

Observation: this .vol_dy.index field may in fact be obsolete or un-needed.

If, for some reason, the operation fails with the new fileset left in an intermediate state (for instance, cleanup was not possible), the VOL_DELONSALVAGE flag should be set in both the supplied status .vol_st.states word and in on-disk storage. (Such an intermediate state, while sometimes unavoidable, is obviously undesirable.)

D.4 Attaching the New Fileset

If a new fileset is created and left on the aggregate (even if in an inconsistent state due to an error), this operation must export it to DFS via the DFS function *vol_Attach*( ) as follows:

```
/* ... the fileset has been created */
error = vol_Attach(volIdp, statusp, aggrp, &vol_ops_vector);
/* ... and when all done */
return(error);
```

Note: This is NOT the VFS+ volume *vol_attach*( ) operation, but a different one. DFS has chosen to implement this up-call function in *ag_volCreate*( ) rather than having the code that called *ag_volCreate*( ) handle it.

The first three arguments are the same as those passed to *ag_volCreate*( ) although in a different order. The fourth is the address of the volume operations entrypoint vector. This call will itself make calls to the volume operations *vol_setdystat*( ) and *vol_attach*( ) fP functions. Note that there is no need to examine the error code it returns (specifically: if it fails, the fileset is not deleted or marked inconsistent).

The vol_Attach call, which is not listed in any of the **DCE DFS** header files, has the following signature:

```
vol_Attach(
    afsHyper             *volIdp,
    struct vol_status    *statusp,
    struct aggr          *aggrp,
    struct volumeops     *volopsp
    );
```

No checking is performed to see if a volume with the specified volume ID or name already exists. The LFS is thus relying upon DFS to avoid a problem in this instance. Presently, the *vol_Attach*( ) up-call will fail if an aggregate with the given ID already exists.

Note: For the sake of per-aggregate consistency, it seems reasonable to suggest that an LFS implementation check within an aggregate for duplicate fileset names and IDs at creation (returning, perhaps, EEXIST).

Appendix E

Information Pertinent to ag_volInfo()

E.1 Volume Structure Fields

Fields within the volume structure (**struct volume**) should be filled in as follows. (See Section 13.3.14 on page 285 for information on the contents of this structure):

.v_paggrp — Set to the **aggrp** argument passed to this call. (That is, this field points to the **aggr** structure for the aggregate that the fileset resides on.)

.v_volOps — Set to point at the **struct volumeops** vector of volume operations provided by the LFS.

E.2 Static Status

Fields within the volume**.v_stat_st** structure should be filled in from the fileset status stored on disk as follows. Any field not mentioned should be zeroed.

.volName

.volId — The *volIdp* argument is used instead of this.

.parentId

.cloneTime

.states — Only the following fileset state bits are set and stored on-disk. (Others should be zeroed.)

```
VOL_READONLY      \ exactly 1 of these
VOL_RW            /   should be set
VOL_IS_COMPLETE
VOL_DELONSALVAGE
VOL_ZAPME
VOL_CLONEINPROG
VOL_REPFIELD
VOL_TYPEFIELD
VOL_REPSERVER_MGD
VOL_NOEXPORT
VOL_IS_REPLICATED
VOL_MOVE_SOURCE
VOL_MOVE_TARGET
```

.type — The states of the VOL_READONLY and VOL_RW flags from the **.v_states** field are replicated here.

Appendix F

Information Pertinent to ag_[de,at]tach( )

F.1 Making an Aggregate Available

The first operation that must be performed in order for an aggregate to be available for use by DFS is to attach it to DFS, putting all its filesets on-line. No other aggregate (or volume) operations will occur on an aggregate until after this has been done. On a successful return from the *ag_attach*( ) call, an aggregate struct (**struct aggr**) will be allocated by higher level software.

F.2 Private Storage

The *af_attach*( ) call can optionally allocate and initialize a block of private storage which will then be made available to all subsequent aggregate operations. This storage must be allocated via a call to *osi_Alloc*( ). A pointer to it along with its size are returned in the **fsdatap** (private data) and **fsdatalen** (private data length) arguments. (The length is required so that higher level software can release the storage if an error occurs later in the attach algorithm.) This pointer is deposited in the **aggr** structure *.a_fsDatap* field, from where it can be retrieved on subsequent aggregate operations. If no such storage is allocated, return values of NULL and 0 should be supplied for **fsdatap** and **fsdatalen**, respectively.

Section 11.8.3 on page 241 defines the signature for *osi_Alloc*( ).

F.3 Aggregate Structure

Aggregates currently attached (see Note below, or containing filesets mounted locally) are known to DFS by means of an **struct aggr** structure that is remembered. This *ag_attach*( ) call is made by DFS whenever it encounters an aggregate that it doesn't already know about. Once this call has been made for a fileset, it will not be made again until an intervening *ag_detach*( ) occurs.

Note: The AGGR_ATTACH_NOEXPORT flag is not normally issued by DFS. It's only user is the local mount command for the Episode LFS, which invokes the DFS attach code to allocate an **aggr** structure. There doesn't appear to be any use to which an LFS can put this flag: it will be present if the first time DFS encounters an aggregate it is for a local mount (as opposed to a DFS export). This is either:

a. Merely an artifact of the Episode implementation.

b. DFS requires that it "have" appropriate **aggr** structures for any filesets mounted locally regardless of whether or not they are currently attached by DFS. However, this isn't being done for UFS file systems.

At the current time, the first alternative is assumed. If that turns out not to be the case, an interface into DFS for use by the LFS **mount** and **unmount** commands will be needed by any vendor-supplied LFS.

Chapter 15

Fileset (Volume) Operations Interface

This chapter describes the facilities provided for the VFS+ Fileset Operations. It provides an overview, discusses fileset types, clones, fileset and clone requirements, fileset indices, LFS modification of fileset status, zero link count files, quotas, anode generation numbers, file identifiers, vnode to LFS association, and lastly, fileset and fileset registry operations.

15.1 Overview

This chapter describes the portion of the VFS+ interface which allows operations on filesets, as introduced in Section 11.6 on page 239 that must be supported by a **DCE LFS**. The *Fileset Registry* is a global kernel table which contains descriptions of each fileset local to the machine. A fileset descriptor includes a pointer to a **struct volumeops** function array, namely those operations that can be performed on the fileset. The associated declaration may be found in Section 13.3.1 on page 274.

An agent that wishes to perform a fileset operation will typically use the fileset ID to hash into the *Fileset Registry* and thus acquire the appropriate fileset descriptor. It will then perform a sequence of operations, calling functions obtained from the descriptor's function array.

In **inode**-based file systems, a fileset represents an **inode** table, and individual files are accessed via **inode indexes**. In **DCE LFS**, a fileset represents an **anode** volume list (avl), and individual files are accessed via **anode** indexes. In either case, agents such as the fileset server will typically iterate over all files and directories in a fileset by iterating over all possible indexes (non-negative integers), stopping when the highest index in the fileset has been processed. It should be borne in mind that the order of iteration, from low index to high, is completely unrelated to any order that might have been derived from the directory hierarchy. There is no guarantee that directories will be processed before the files that are in them, or vice versa, although the root directory will generally be processed first.

A description of the operations available within the fileset function array appears below. The array is an exported data type. (The associated declaration is found in Section 13.3.1 on page 274). It is reproduced here for convenience.

```
struct volumeops {
    /* per-fileset operations */
    int (*vol_hold)();
    int (*vol_rele)();
    int (*vol_lock)();
    int (*vol_unlock)();
    int (*vol_open)();
    int (*vol_seek)();
    int (*vol_tell)();
    int (*vol_scan)();
    int (*vol_close)();
    int (*vol_destroy)();        /* actually whole-fileset */
    int (*vol_attach)();
    int (*vol_detach)();
    int (*vol_getstatus)();
    int (*vol_setstatus)();

    /* per-file operations */
    int (*vol_create)();
```

```
        int (*vol_read)();
        int (*vol_write)();
        int (*vol_truncate)();
        int (*vol_delete)();
        int (*vol_getattr)();
        int (*vol_setattr)();
        int (*vol_getacl)();
        int (*vol_setacl)();

        /* more whole-fileset operations */
        int (*vol_clone)();
        int (*vol_reclone)();
        int (*vol_unclone)();

        /* vnode lookup operations */
        int (*vol_vget)();
        int (*vol_root)();
        int (*vol_isroot)();

        /* more per-fileset operations */
        int (*vol_getvv)();
        int (*vol_setdystat)();
        int (*vol_freedystat)();
        int (*vol_setnewvid)();

        /* another per-file operation */
        int (*vol_copyacl)();

        /* per-fileset operations */
        int (*vol_concurr)();
        int (*vol_swapids)();
        int (*vol_sync)();
        int (*vol_pushstatus)();

        /* per-file operations */
        int (*vol_readdir)();
        int (*vol_appenddir)();

        /* per-fileset operations */
        int (*vol_bulksetstatus)();
        int (*vol_getzlc)();

        /* another per-file operation */
        int (*vol_getnextholes)();

        /* yet another per-fileset operation */
        int (*vol_deplete)();
    };
```

15.1.1 Classes of Fileset Operations

There are four classes of fileset operations, as delineated by the comments in the above structure declaration:

1. **Per-fileset operations**. These operations are used to get and set fileset status and iterate over all files in the fileset.
2. **Whole-fileset operations**. These operations are similar to the per-fileset operations in that they operate on an entire fileset, but they differ in that they embed iterations over every low-level object in a fileset, even those corresponding to **ACL**s that are not ‘‘files’’ in the usual sense. These operations are used either to destroy a fileset or to manipluate the clone status of filesets.
3. **Per-file operations**. These functions are used to create and delete files, inspect or change the contents of a file, and get and set the file attributes.
4. **Vnode lookup from fileset and file information**. Return a vnode pointer associated with a given fileset and file ID.

The signatures for the *fileset* operation functions appear in the section of this chapter labeled *Fileset Array Functions*. It is Section 15.14 on page 340.

15.1.2 Fileset Registry Functions

The following functions manipulate the *fileset registry*:

volreg_Enter( ) Given fileset and aggregate IDs, along with a pointer to the associated fileset information, create an entry in the Fileset Registry and insert the given information into it.

volreg_Delete( ) Delete the Fileset Registry entry (if any) that corresponds to the given fileset and aggregate ID pair.

volreg_Lookup( ) Given a file ID, this function returns a pointer for the fileset descriptor and, optionally, the vnode corresponding to the file.

The signatures for the *fileset registry* functions appear in the section of this chapter labeled *Fileset Registry Array Functions*. It is Section 15.15 on page 396.

15.2 Fileset Types Overview

By means of cloning and read-only replication, filesets will in general be grouped into related sets. Within these groupings, filesets can have different types (or roles) as indicated by the VOL_TYPEFIELD field of the fileset status *.states* field. In the following types, a grouping is referred to as a *group*. The types (or roles) are:

VOL_RW The read-write fileset, of which there can be only one within a *group*.

VOL_READONLY A read-only replica, of which there can be several within a *group* (at different sites). It’s name is the same as the VOL_RW one, with a ".readonly" suffix. These filesets are managed by the replication servers. It is possible for a read-only replica to reside within the same aggregate as its read-write master.

VOL_BACKUP A read-only clone of the read(hywrite fileset, created via the **fts clone** command. It’s name is the same as the VOL_RW one, with a ".backup" suffix. Because it is a clone, it MUST reside within the same aggregate as its read-write master. There can only be one of these within a *group*.

VOL_TEMP A temporary clone of the read-write master used in the process of moving or replicating a fileset.

Volume IDs corresponding to the first three of these types (RW, READONLY and BACKUP) are all allocated when a fileset is first created and are all stored in the FLDB. The fourth (TEMP) type uses a temporary ID since it is only visible to the algorithm performing the move.

15.3 Fileset Clone Algorithms

In the following discussion, the types from the previous section, Section 15.2 on page 327 , are referred to. Also, throughout this document, the symbol " ===> " will be used to denote an "is backed by" relationship.

The fileset cloning mechanism uses copy-on-write techniques to create a "new" fileset that is a snapshot of the current state of an existing one. Usually, the following steps are followed by DFS in the creation of a clone.

a. An existing fileset, either a VOL_RW (usually) or VOL_READONLY one (when updating RO replicas), is to be cloned. Let this fileset be F1.

b. A new fileset, F2, is created via *ag_volCreate*( ). (See Section 14.4 on page 311.) Since the AGGR_CREATE_ROOT flag is not specified to this call, this fileset starts out totally empty.

c. A series of *vol_clone(F2, F1, ..)* (description is found in Section 15.14 on page 369) calls are made to actually create clones on F2 for the files on F1.

d. At this point, the DFS terminology speaks in terms of fileset F2 being backed by F1. That is: "F2 is backed by F1" (represented by F2 ===> F1). F1, the original fileset, is known as a backing fileset. Files on F1 are known as backing files.

   The LFS should NEVER allow modifications to backing files. In this scenario, *only* F2 can be modified.

   Throughout this document, the symbol " ===> " is used to denote an "is backed by" relationship.

e. In many (most, perhaps) scenarios, the desire was to create a read-only clone and have the original (F1 as in this discussion) fileset continue to be writeable.

   To accomplish this, the fileset identities are swapped via a call to *vol_bulksetstatus*( ) (described in Section 15.14 on page 392) that swaps just about every status field in the two filesets. Following this, the relationship has been changed to: "F1 ===> F2" where F1 is writeable and F2 isn't. Note that the newly created physical fileset (step (b)) is now F1 (it has F1's ID, and so forth.).

f. This sequence can be repeated again, giving rise to a chain of related (cloned) filesets.

g. Assume an initial state of "F1 ===> F2". Following steps (b) through (d), the following relationship holds: "F3 ===> F1 ===> F2".

h. Following another identity swap between F3 and F1, a final result is arrived at, namely: "F1 ===> F3 ===> F2".

Note: Unless explicitly indicated in the detailed descriptions, the fileset status fields related to fileset clones (and replicas, and so on) should not be examined or used by the LFS since their use by DFS could change in the future.

15.3.1 Requirements on Cloning

The following requirements on cloning hold.

1. Under ordinary circumstances, a 3-deep chain of cloned filesets is the "worst" that DFS will ever create. The LFS, however, should be capable of supporting chains of *arbitrary*, or at least, somewhat larger than 3 depth.

2. The LFS must prevent any modifications to backing filesets, which are read-only. For example, given the following state: "F1 ===> F2 ===> F3" *only* F1 is writeable; F2 and F3 are read-only.

   Although Transarc's Episode appears to return EIO in such cases, EROFS would seem to be a more appropriate error return.

   Note: The above restriction only applies to modifications to the file system visible contents of a fileset (files, directories, ACLS, and so forth.) via either the **vnode** or **fileset** interface.

   The fileset header of these backing filesets can be modified via the following, for example:

   - *vol_setstatus*( )
   - *vol_swapids*( )
   - *vol_bulksetstatus*( ), and so on.

   The following operations are obviously permitted as well:

   - *vol_delete*( )
   - *vol_unclone*( )
   - *vol_reclone*( ).

   Note: In actuality, there appears to be a number of cases in which Transarc's Episode will allow modifications to certain extended file status on these backing filesets. If so, then perhaps it is the case that DFS is not relying on this behavior.

3. A new clone (as opposed to a reclone operation) will never be made of a backing fileset. For example,given the following state:

   ```
   "F1  ===>  F2  ===>  F3"
   ```

   only F1 can be the starting point in step (a) above.

4. The act of creating a clone is assumed to be a relatively short operation ("small" number of seconds for a "reasonable" sized fileset) and consume a "small" amount of additional disk space. See the discussion *Looping Operations* in Section 15.10 on page 335.

5. Consult the *Copy-on-Write Impacts* section (Section 16.3 on page 421) in the **vnode** operations chapter for additional assumptions regarding cloning.

6. All files on a fileset, even those with a Zero-Link-Count, are cloned.

15.3.2 Uses for Clones

Among the various uses to which DFS puts clones, the following are typical.

1. The **fts backup** command creates a backup fileset (with a suffix of ".backup" and a type of VOL_BACKUP) which can be backed up in parallel to on-going access to the original fileset or left in place as an on-line backup. The steps followed are precisely those detailed in the cloning steps (a) through (e) in Section 15.3 on page 328.
2. The **fts move** command moves a fileset to a new location. It accomplishes this via the following steps:
   i. Create a temporary clone Ft of the original fileset F1. After the fileset identity swap, the result is "F1 ===> Ft".

      If F1 had already had a backing fileset F2 (perhaps a *.backup* fileset), the result would be: "F1 ===> Ft ===> F2".

   ii. The clone Ft is copied to the new location, while on-going access to F1 is allowed.

   iii. The temporary Ft is deleted.

   iv. Changes to F1 that occurred during step (ii) are incrementally copied to the new location. During this hopefully short step, fileset F1 is un-acessible for normal access.

   v. The FLDB is updated to reflect the new location for the fileset. Recall that this new location only applies to VOL_RW and VOL_BACKUP types. (Those types were discussed in Section 15.2 on page 327.)

   vi. The original fileset is deleted. If there is a *\.backup* fileset to go with it, it is deleted as well.

   Note: It was never copied to the new location! This might be changed in the future.
3. A replicated read-only fileset (call it Fro) is updated from its master fileset (call it Frw) as follows. Initially, Frw and Fro exist at different (typically) sites.
   i. The contents of Frw are pushed to a staging fileset co-resident with it. This may be accomplished via a clone created against Frw. Following the usual identity swap, the result is:

      ```
      "Frw  ===>  F1"          Master site
      ```

      F1 will either have a temporary ID or the read-only ID from the FLDB.

   ii. A clone is then created at the replica site against Fro. This time, however, the identities are not swapped and the new fileset (F2) remains writeable. The result is:

      ```
      "F2  ===>  Fro"          Replica site
      ```

      F2 will have a temporary fileset ID.

   iii. The contents of F1 (backing Frw) are copied to the new fileset F2. Following this copy, the identities of Fro and F2 are swapped so that references to the read-only replica are now directed to the newly updated fileset. This swap is accomplished via a *vol_swapids*( ) fileset call.

   Note: Unless explicitly indicated in the detailed descriptions, the fileset status fields related to fileset clones (and replicas, and so on) should not be examined or used by the LFS since their use by DFS could change in the future.

15.3.3 Some Fileset and Clone Requirements

In addition to the requirements detailed elsewhere in these specifications, DFS makes a number of general assumptions regarding filesets and cloning. They are:

1. Several modes of fileset usage are expected. In some scenarios, small numbers of very large filesets will prevail while in others large numbers of small filesets will prevail.
2. An aggregate is expected to be capable of supporting a "large" number of filesets. For suitably large aggregates, a lower limit in the thousands or tens-of-thousands seems reasonable. (Therefore: if there is a limit, it should scale in some regard with the aggregate size.)
3. Although it has been mentioned elsewhere, this bears repeating. An LFS must dis-allow any modifications (except for those via *vol_xxx*( ) fileset operations) to files on a read-only fileset: either a *.backup* clone or a *.readonly* replica.

15.4 Fileset Indices

Filesets on an aggregate are identified by a zero-based index about which DFS makes the following assumptions.

- A fileset index is visible outside the LFS in the following ways.
  - — The index of the fileset to be queried is passed as an argument to the *ag_volInfo*( ) call.
  - — The *ag_volCreate*( ) call returns fileset index for a newly created fileset in the volume *.vol_dy.index* field.
  - — A fileset index is returned in the **Txvattr** *.backingIndex* field by the *vn_getattr*( ) and *vol_getattr*( ) calls.

  Since it is stored internally within DFS, these indices must remain consistent over time. (Strictly speaking, it ought to be "safe" to allow a fileset's index to change each time its aggregate is attached ... although there could well be problems with such an approach.)
- The general expectation is that when a fileset is created (by *ag_volCreate*( )), the lowest *unused* index is assigned to it. Although using monotonically increasing indexes might seem desirable (not re-using the indices of deleted filesets), this is not really in keeping with the manner in which *ag_volInfo*( ) incrementally steps through the filesets on an aggregate.
- A fileset's index stays with it *forever*. It is not changed by the *vol_clone*( ), *vol_reclone*( ), *vol_swapids*( ), *vol_bulksetstatus*( ) or *vol_setstatus*( ) operations. Although filesets may change or swap their DFS identity (fileset ID), their fileset index remains the same.

15.5 LFS Modification of Fileset Status

For the most part, the LFS neither interprets nor modifies volume status fields. The exceptions, which are also indicated under the appropriate aggregate and fileset operations, are listed here.. Refer to Section 13.3 on page 274, *Fileset Data Types*, for details of the fields and flags discussed in this section.

- The VOL_DELONSALVAGE flag in *vol_stat_st.states* is modified by a number of operations.
- The VD_RDONLY flag is set in a newly created vnode if either the VOL_READONLY flag is set in its fileset *vol_stat_st.states*, the fileset was mounted read-only or the file is backing (clone) another file in a copy-on-write relationship.

  Note: While the above point is correct, it does not really apply to discussing fileset status modifications.
- The **vol_stat_dy** *.updateDate* field is updated whenever a fileset's status is changed. This includes file creation and deletion, fileset status operations, the clone operations and whenever the volume version is incremented.
- The **vol_stat_dy** *.accessDate* field is updated whenever the *.updateDate* field is.
- The **vol_stat_dy** *.volversion* is advanced whenever any modification is made to the data or status of any file contained within that fileset.

  Note: Transarc's Episode advances the volume version by simply incrementing it. Other approaches, such as setting it to the current time (given enough precision), would seem to be acceptable as well ... as long as the version changes and never decreases (64-bits) in value.
- The **vol_stat_dy** *.allocUsage* and *.visQuotaUsage* fields are both adjusted as the size of a fileset changes.

15.6 Zero Link Count Files

Ordinarily, files whose link count is zero are actually deleted by the LFS when their vnode reference (usage) count goes to zero. (See the discussion under the *vn_inactive*( ) vnode operation, Section 16.7 on page 453.)

In certain circumstances, DFS will *protect* a deleted file by artificially holding its vnode until it is certain that any remote clients are through with it. Following a crash or shutdown, any remaining zero-link-count files (their vnodes were still held at crash or shutdown time) must be preserved by the LFS in order to give DFS a chance to re-adopt them once the fileset is attached again. Any FSCK-like programs that are run must also preserve these files.

Note: These zero-link-count files are **copied** by the *vol_clone*( ) operation into the cloned fileset.

Note: It is possible for a zero-link-file to have its link count incremented before it is actually deleted, returning it to a "normal" status. (Although Transarc's Episode allows for this, it is not known how this might actually occur ... if it ever does.)

When a read-write fileset is attached, DFS makes a series of *vol_getzlc*( ) volume calls to identify these files with zero link counts. This call returns held vnodes. When and if DFS determines that the file can safely be deleted, it releases the vnode which, as discussed above, actually triggers the deletion.

Note: For a readonly replica or clone, DFS never calls *vol_getzlc*( ); these zero-link-count files remain intact until they are explicitly deleted (via *vol_delete*( )) or until their fileset is destroyed.

Additional discussion can be found under the *vn_inactive*( ), *vol_delete*( ), *vol_setattr*( ) and *vol_getzlc*( ) operations.

An obvious difficulty here is that there is no guarantee that, following a re-boot, DFS will actually trigger the eventual reclamation of these files and the space they occupy. (Specifically, DFS may not be started again.) Although this problem is outside the scope of DFS, one can argue that DFS should provide the framework of a solution. One possible way of dealing with this issue is as follows.

1. Only give this zero-link-count treatment (including retention during FSCK) to files that have actually been accessed by DFS. (Perhaps their fileset has been exported; perhaps the file has been operated on by *vol_xxx*( ) operations such as *vol_vget*( ).)
2. Employ some sort of (perhaps LFS specific) timeout mechanism to ensure the recovery of such files if DFS has not come along within a "reasonable" amount of time.

Note: Although the intent here seems desirable (preserving deleted files until they are no longer in use), practical concerns seem to get in the way. The biggest problem is that the DFS layer, in order to avoid tying up all the vnodes in a system, will only artificially hold up to 220 files at most. Although this should be sufficient in some situations, it will not be enough for large servers.

15.7 Fileset Quotas

There are two measures of the amount of disk space being used by a fileset.

allocated The amount of disk space ACTUALLY in use by a fileset.

visible The amount of disk space LOGICALLY used by a fileset. This is what is returned by the *statfs*( ) vfs operation.

Normally, these these measures will be the same. For cloned filesets that are sharing disk space via a copy-on-write mechanism, these measures can differ as follows.

- A given disk block can only show up in the *allocated* measure for a single fileset, regardless of how many filesets are sharing it via copy-on-write.

  Given a disk block that is shared (copy-on-write) by multiple filesets, DFS does not specify which fileset's *allocated* usage measure that block show up under.

  Note: This is one possible mechanism by which a particular LFS (and its cloning algorithms) could be distinguished from the Transarc LFS. Under the Transarc LFS, shared (copy-on-write) blocks appear *allocated* under the backing fileset.

  This *allocated* measure is the one that is important from a *how full is this physical aggregate* aspect.

- A given disk block will show up in the *visible* measure of EVERY fileset sharing it via copy-on-write. In one sense, this *visible* measure indicates how much physical disk space a fileset could require if all its copy-on-write linkages were broken.

  This *visible* measure is the one that is normally of most use to users controlling the usage of their filesets.

- A fileset's *visible* usage measure will, therefore, always be equal to or larger than its *visible* measure (larger than for a cloned fileset; NEVER less than). Considering all the filesets on an aggregate, the sum of all the *allocated* usage measures will be no greater than the aggregate size while the sum of all the *visible* usage measures may be larger than the aggregate size.

There are several fileset status fields that are related to a fileset's disk usage. Above the LFS, they are represented as 64-bit byte counts. The first two are input to the LFS ("quotas"). The latter two are computed and maintained by the LFS and returned to the higher level's ("usage").

.allocLimit The maximum allowable size (quota) that a fileset's *allocated* usage can grow to.

.visQuotaLimit The maximum allowable size (quota) that a fileset's *visible* usage can grow to.

.allocUsage The *allocated* measure of the disk space in use by a fileset. The *visible* measure of the disk space in use by a fileset.

Operations that would cause the **.allocLimit** or **.visQuotaLimit** values for a fileset to be exceeded fail with an EDQUOT error. The Super User (ROOT) enjoys no special privilege in this regard.

15.8 Anode Generation Numbers

Note: In the following, "inode" means "anode".

Each inode has an associated generation number which, in combination with an inode index, uniquely identifies a particular instantiation or use of an inode.

An LFS has freedom in how it generates these generation numbers. For example, it can use either a per-inode counter or a per-fileset counter (Transarc's Episode uses the fileset **vol_stat_dy** .*unique* field for this). The key behavior is that each time an inode is re-used (taking into account any inode table shrinkage and re- growth), a different generation number must be used for it.

The generation number for the root directory on a fileset should always be set to 1.

Within DFS, these inode generation numbers are referred to as "uniquifiers" and reside within the .*Unique* field of an **afsFid** structure.

15.9 File Identifiers (afsFids)

DFS identifies files via an afsFid structure. The fields within an **afsFid** have the following significance to an LFS.

.*Vnode* Represents a file.

.*Unique* An anode (inode) index and generation number. An LFS is free to use whatever encoding is appropriate for it. There appears to be an assumption made that the root directory on a fileset has the lowest (from the set of legal or possible values) inode index. Consult the description under *vol_open*( ) (Section 15.14 on page 345) on the setting of the *vol_handle.index* field.

A .*Unique* field of -1 is interpreted as a don't care value; it is not required to match the inode generation number currently found on the file.

Note: DFS does not appear to rely on this above behavior (.*Unique* of -1), although this is what Transarc's Episode does.

.Cell An identifier for the DFS cell.

Although an LFS should not have any reason to to examine this field, higher levels will generally set it to 0..1 (.high = 0, .low = 1). It seems safest to follow the same convention for any **afsFid**s that are returned by the LFS.

.Volume An identifier for the DFS fileset.

In general, an LFS should not have any reason to examine this field.

Several of the fileset operations take an **afsFid** argument which identifies a file. In most (perhaps all) cases, these afsFids were originally returned to user space via an earlier *efs_scan*( ) or *efs_create*( ) call based on a canonical inode position argument (VOL_ROOTINO specifies the root) passed to them. Normally, therefore, the LFS should always be able to locate the file corresponding to the specified **afsFid**. If for some reason, however, the object in question cannot be located, the operations should fail with either an ESTALE (if the *.Unique* field was the problem) or EIO (any other problem; namely, the given *.Vnode* index not in use) error.

15.10 Looping Operation Considerations

The *vol_deplete*( ), *vol_clone*( ), *vol_reclone*( ) and *vol_unclone*( ) operations all operate over an entire fileset. To avoid executing within the kernel for an arbitrarily long time and causing user-space RPC timeouts, they should periodically return with a code of ELOOP. This return value tells the user space code that this operation hasn't completed and that it should repeat the call. Since these calls do not accept an iteration argument, it may be necessary to record a "where to continue" marker in the LFS private fileset storage that is established in *vol_setdystat*( ). If this mechanism is used, it should be reset to "start at the beginning" at *vol_open*( ) time and at the completion (that is, when something other than ELOOP is returned) of any of these operations.

Note: As currently exists, these timeouts will occur within a small number of minutes. This document recommends that these looping operations plan on spending no more than 30 seconds or so within the kernel. A certain amount of care may be required since some operations can clearly take longer than this (namely, deleting or cloning a VERY large file.) Some possibilities: processing portions of a file each iteration (as much as will "fit" within 30 seconds) or operating asynchronously with helper daemons to ensure that the process that issued the *vol_xxx*( ) operation is able to return to user space every 30 seconds.

15.11 Vnode to LFS Association

During normal operation, there is an association between a vnode and the LFS-specific data structures which represent that object.

The fileset operations described in this specification enable an administrative utility to come in and make fundamental changes to the underling LFS state. Depending on the precise implementation, some of these operations might require that the above mentioned association between a vnode and its LFS state (namely, location information) be broken. Examples of such operations are those involved in cloning, uncloning, recloning, deletion, restoration and movement. In the simplest case, the vnode usage count can be *driven* to zero so that it can be simply released or discarded. If this isn't possible (say a file is open locally), the vnode must be de-coupled from its LFS state -- which can change out from under the vnode. Such dis-association between a vnode and the LFS requires at least the following:

- Flushing any modified cache state back to the LFS.
- Leaving enough location information (inode index, generation number, fileset ID) in the vnode to enable a "hook up" with the LFS at a later time.

For similar reasons, any cached state (in the VM or naming cache, for example) must be considered in the face of volume operations that might modify or examine that state.

Once such a vnode has been dis-associated with the underlying LFS state, a corresponding mechanism is required to re-bind the vnode to the LFS (location information, status) before it is used again.

If the fileset no longer resides at the machine in question, an [ENODEV] error should be returned from the operation. If the fileset is still local to the machine but the file itself no longer exists, an [EIO] error should be returned from the operation.

Note: The precise error values returned in these cases appear not to be that crucial. DFS has suggested that returning [ESTALE] in both cases might be better.

15.11.1 Determining whether a Fileset is Local

Although there are numerous possible LFS-specific mechanisms for determining if a fileset is still local, Transarc's Episode does the following.

- Initializes an afsFid from the volume ID recorded in the vnode at dis-association time.
- Calls *volreg_Lookup(&fid, &volume_ptr)* in the DFS to let it make the determination. This call returns [ENODEV] if the fileset is not in the *fileset registry*.
- If this call succeeds, the hold count on the returned volume structure is incremented. The LFS must decrement it by calling through the fileset ops vector as follows:

  ```
  (*volume_ptr->v_volOps->vol_rele)(volume_ptr);
  ```

  Note: The fileset might reside within a different LFS entirely!):

  If the volume structure is not needed by the LFS, this step can be avoided by calling *volreg_Lookup*() with NULL as the second argument -- indicating that the caller does not desire a returned volume structure.
- The anode index and generation number recorded in the vnode are used to look up the object on the fileset.

15.11.2 Handling Dis-sociations and Re-associations

There are several alternatives for scheduling these dis-associations and re-associations.

The dis-associations can be handled in several ways. Two possibilities:

[d1] At *vol_open*( ) time, all the vnodes on a fileset can be dis-associated.

[d2] At individual *vol_xxx*( ) operations, the vnodes affected by that call can be dis-associated.

Likewise, the re-associations can be handled in several ways. Two possibilities:

[a1] At *vol_close*( ) time, re-associate any vnodes for the fileset in question. Vnodes that could not be re-associated are marked such that any use of them (checked for at individual *vn_xxx*( ) operations) fails.

[a2] Before using any vnode (say, as input to individual *vn_xxx*( ) operations), check to see if that vnode requires re-association. Again, if the re-association does not succeed, the operation fails.

The difference between [d1] and [d2] appear to be minor. One difference between [a1] and [a2] is whether it is possible for [a1] to encounter a vnode which cannot be re-associated until some later time (a subsequent *vn_xxx*( )). This should never occur (either it can be re-associated at *vol_close*( ) time or it will never be able to be).

Usually, vnodes will be unused at the time of such destructive operations. Hence, they can be cleanly discarded at *vol_open*( ) time or at re-association time if a problem is encountered. However, there are two exceptions.

1. Vnodes for files being accessed locally can be in use at *vol_open*( ) time. These should not cause a problem since DFS will not delete or move such a locally mounted fileset. (However, it would be wise for an LFS to take the proper precautions and not rely on DFS to protect it.)
2. Normally, the DFS file exporter locks out conflicting fileset operations while it is holding a vnode (the vnod--to-volume synchronization mechanism). In one situation, however, (in the read-write path waiting for an RPC data pipe) it retains a held vnode and temporarily allows fileset operations (of any type) to occur. If a destructive fileset operation does manage to interrupt it, there will be a vnode which
   - cannot be discarded at *vol_open*( ) time (since it is held)
   - cannot be re-associated at *vol_close*( ) time (due to its deletion)
   - needs to be detected when the exporter comes back and tries to use it again

Therefore, even if the re-associations are performed as in [a1], the checks in [a2] are still required.

Either way, attempts to use a vnode which could not be re-associated should fail with [ESTALE].

Additional care might be required if the implementation is such that *vol_xxx*( ) operations themselves instantiate or re-associate vnodes. Although DFS normally scans through the vnodes on a fileset from beginning to end, this need not be so. Another round of dis-association might be required under approach [d1] above if a *vol_scan*( ), *vol_seek*( ) or *vol_create*( ) are observed to back up to a previously processed object or vnode.

15.11.3 Complications in Dis-sociations and Re-associations

A related complication needs to be kept in mind as well. A number of *vol_xxx*( ) operations delete files. For example, *vol_delete*( ), *vol_deplete*( ) and some of the cloning ones. Any dis-associated vnodes which result (in other words, vnodes which, for whatever reasons, could not be outright released or freed) must be treated carefully if a new file with the same inode index and a different generation number (uniquifier) is then created. These dis-associated, *stale* vnodes must not be inadvertantly used by a lookup operation working from an anode index (only) in a directory entry. These lookups are only interested in the one, *true* object that might exist on the fileset. The permissable operations on these *stale* vnodes are:

vn_inactive( ) when their usage count finally goes to 0.

vol_create( ) if it attempts to re-create the object with the same fileset ID, anode index and generation number. (Depending upon the implementation, *vol_reclone*( ) might encounter this situation as well.)

Because dis-associated vnodes hold both an anode index and generation number (see above), determining their validity against the actual contents of the fileset is not difficult. The key point is that there can be several vnodes in existance for a given fileet ID and anode index (but with different generation numbers). At most one of these is valid; the remainder are stale ... waiting for *vn_inactive*( ) to finally retire them.

The cause of this is situation that the fileset operations aren't bound by many rules. They can delete objects that are still being used (presumably, locally mounted). They might (based on the implementation) give rise to multiple vnodes for the same anode index and fileset. While a fileset is open (blocking ordinary accesses), they can violate the invariant that a given **fid** resolves to one-and-only-one object (the same one every time). What is important is that when the fileset is finally closed, no inconsistencies are observed to remain (other than the fact that a file locally open has been deleted and perhaps replaced).

Another complexity to be aware of is that calls to *vn_inactive*( ) are not synchronized with fileset operations (this is discussed in the vnode operations chapter, *VFS (Vnode) Interface and Operations*, Chapter 16 on page 415). Therefore, this operation can come in while a fileset is open and being operated upon (by *vol_xxx*( ) operations).

Note: This will change in the future.

15.11.4 Fileset Moves

A brief summary of how the DFS client handles moved filesets follows:

- Fileset movement between machines is only cleanly handled for requests that come in via the protocol exporter (from a remote cache manager). As alluded to previously in Section 15.11 on page 336 , the DFS **fts** command refuses to delete, zap or move (non-locally) a fileset that is mounted locally.

  When a remote file request arrives, the exporter first determines if the desired fileset (identified by a volume ID) is in fact local and available. This determination is made by a routine (*volreg_Lookup*( ), see Section 15.15 on page 396) that consults a table, maintained by DFS, of locally attached and exported filesets. If the desired fileset is available and local, it is *held* while the operation proceeds. While the fileset is held this way, any conflicting fileset operations are delayed.

  If it turns out that the fileset is currently in use by a conflicting fileset operation, this request is stalled until it is able to proceed.

If, alternatively, the fileset is not found locally, a [VOLERR_PERS_DELETED] (derived from a preliminary [ENODEV]) error is returned to the remote client. That client, on receipt of this error, attempts to re-bind to a correct location (from an updated FLDB) before retrying its request. (This occurs in the cache manager routine *cm_Analyze*( ).)

- Fileset motion within a given machine must be handled entirely within the LFS (The above DFS mechanism is not triggered since the fileset is still local.). This mechanism also allows local, non-DFS access to be unaffected by local moves.

The key point is that any LFS implementation needs to carefully consider the VOL_OP_XXX bits (defined in Section 13.3.18 on page 286) in the type argument passed to *vol_open*( ) and decide, on a case-by-case basis, which warrant vnode dis-association and which might require cache purging, and so forth. Each VOL_OP_Xxx bit warns of an impending *vol_xxx*( ) operation whose potential effect on file system state requires analysis.

DFS guarantees, with a few exceptions, that while a fileset is open for one of these operations, conflicting vnode operations are not allowed to come in. See *Synchronization Between Vnode and Fileset Operations* Section 16.5 on page 422, in the **vnode** operations chapter titled *VFS (Vnode) Interface and Operation*, Chapter 16 on page 415 of this document. As discussed there, DFS presently does allow some vnode operations to proceed. The LFS has the responsibility for these of dealing with in-progress volume operations and dis-associated vnodes.

15.12 Private LFS Fileset Data

As each fileset is exported to DFS, file system independent code makes calls to the *vol_setdystat*( ) and *vol_attach*( ) fileset operations. If an LFS requires that some private data or a handle be made available to it on subsequent volume operations, it can arrange for this as follows.

- Allocate space during the *vol_setdystat*( ) operation and place a pointer to this in the *.v_fsDatap* fileset **struct volume** structure field.
- In subsequent calls, this pointer can be retrieved from the supplied fileset (volume) structure.

Filesets are *detached* via calls to *vol_freedystat*( ) and *vol_detach*( ). Any storage allocated by *vol_setdystat*( ) must be released in *vol_freedystat*( ).

15.13 Fileset Operations

File system independent code above the LFS enforces the requirement that these operations can only be issued by ROOT (the local super user). For this reason, the credentials argument to many of these calls can be ignored except as indicated in the individual descriptions in Section 15.14 on page 340, *Fileset Array Functions*.

The descriptions do not list all possible error cases and return values. The primary intent has been to identify the DFS specific behavior and situations that would not arise in an ordinary **UNIX** (UFS) file system implementation.

With the exception of *vol_open*( ), *vol_setdystat*( ), *vol_hold*( ) and *vol_rele*( ), all of these fileset operations have been preceeded by a corresponding call to *vol_open*( ).

Many of these operations accept a position or anode (inode) index argument. These are intended to ONLY select ordinary files and directories. **ACL**s and property lists are not visible as distinct objects via any of the standard LFS interfaces (fileset, vnode, aggregate operations). If they are in fact implemented as objects on the disk, they must not appear in the "inode space" of visible filesets.

Except where explicitly noted in the following descriptions, these fileset operations do *not* have the side affect of updating the accessed, modified or changed times (atime, mtime, ctime) of a file.

15.14 Fileset Array Functions

Descriptions of the members of the **struct volumeops** function array follow. There are four classes of fileset operations in this array; per-fileset, whole-fileset, per-file, and vnode lookup. The function array definition in Section 15.1 on page 325 delineate them as comments in the structure itself. Refer to it for information about which class the operations belong to.

NAME

vol_hold — Increment the reference count of a fileset descriptor

SYNOPSIS

```
int vol_hold(
    /* IN */ struct volume *vold
    );
```

ARGUMENTS

vold fileset descriptor pointer.

DESCRIPTION

Hold a volume structure by incrementing the reference count of the **vold** fileset descriptor.

DISCUSSION

This operation should be performed under the volume lock as follows:

```
lock_ObtainWrite(&volp->v_lock);
volp->v_refCount++;
lock_ReleaseWrite(&volp->v_lock);
```

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vol_rele — Decrement the reference count of a fileset descriptor

SYNOPSIS

```
int vol_rele(
    /* IN */ struct volume *vold
    );
```

ARGUMENTS

vold fileset descriptor pointer.

DESCRIPTION

Release a volume structure by decrementing the reference count of the **vold** fileset descriptor. Wake up anyone that might be waiting on the structure.

DISCUSSION

This operation should be performed under the volume lock as follows.

```
{
lock_ObtainWrite(&volp->v_lock);
if (volp->v_count == 1)
    vol_VolInactive(volp);
else
    volp->v_refCount--;
    lock_ReleaseWrite(&volp->v_lock);
}
```

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vol_lock — Lock access to a fileset

SYNOPSIS

```
int vol_lock(
    /* IN */ struct volume *vold,
    /* IN */ int           type
    );
```

ARGUMENTS

vold Fileset descriptor pointer.

type type of lock to be obtained. Allowable values are **READ_LOCK**, **WRITE_LOCK**, and **SHARED_LOCK**.

DESCRIPTION

Lock access to a fileset.

DISCUSSION

Although this function is exported to user-space **DCE DFS** , there are no users of it.

This operation should utilize the standard DFS lock primitives as follows:

```
if (type == READ_LOCK)
    lock_ObtainRead(&volp->v_lock);
else if (type == WRITE_LOCK)
    lock_ObtainWrite(&volp->v_lock);
else if (type == SHARED_LOCK)
    lock_ObtainShared(&volp->v_lock);
```

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL] The value of *type* is not READ_LOCK, WRITE_LOCK or SHARED_LOCK.

NAME

vol_unlock — Unlock access to a fileset

SYNOPSIS

```
int vol_unlock(
    /* IN */ struct volume *vold,
    /* IN */ int           type
    );
```

ARGUMENTS

vold Fileset descriptor pointer.

type type of lock to be released. Allowable values are **READ_LOCK**, **WRITE_LOCK**, and **SHARED_LOCK**.

DESCRIPTION

Unlock access to a fileset.

DISCUSSION

Although this function is exported to user-space **DCE DFS** , there are no users of it.

This operation should utilize the standard DFS lock primitives as follows:

```
if (type == READ_LOCK)
    lock_ReleaseRead(&volp->v_lock);
else if (type == WRITE_LOCK)
    lock_ReleaseWrite(&volp->v_lock);
else if (type == SHARED_LOCK)
    lock_ReleaseShared(&volp->v_lock);
```

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL] The value of *type* is not READ_LOCK, WRITE_LOCK or SHARED_LOCK.

NAME

vol_open — Begin a sequence of user space administrative operations in the given fileset

SYNOPSIS

```
int vol_open(
    /* IN */  struct volume     *vold,
    /* IN */  long              opentype,
    /* IN */  long              errtoreturn,
    /* OUT */ struct vol_handle *handle
    );
```

ARGUMENTS

vold	Fileset descriptor pointer.
opentype	mask of operations to be carried out in this sequence containing a set of VOL_OP_xxx bits that indicate the types of operations that will be performed on the open fileset.
errtoreturn	This argument is unused. The error code to return for operations that attempt to access this fileset between this *vol_open*( ) and the associated *vol_close*( ).
handle	pointer to the fileset iterator structure created for the given fileset. It should be filled in as per the DISCUSSION section.

DESCRIPTION

This call is used to begin a sequence of operations on the fileset described by *vold*. The *opentype* parameter declares the kinds of fileset operations to be carried out. The *errtoreturn* parameter is used to set the error code to return should other operations attempt to access this fileset between this *vol_open*( ) call and the bracketing *vol_close*( ).

The value of the input *opentype* parameter is constructed as the bitwise inclusive-OR of the VOL_OP_xxxx definitions from the include file **volume.h**, such as **VOL_OP_SCAN**. This value is useful as a hint that allows the file system to prepare a fileset to perform the operations. If such preparation is expensive for some operations, that preparation need not be carried out unless the sequence in question will include those operations.

This is the same operation bit-mask that is passed to the *vol_concurr*( ) operation, which is responsible for selecting the kinds of vnode operations that may be carried out concurrently with the given sequence of operations.

At most one sequence of fileset operations may be in progress at any time. Calls to *vol_open*( ) that are made while another sequence is in progress will fail, being given as their error code the value passed as *errtoreturn* when beginning the sequence in progress.

A sequence of fileset operations is terminated by a call to *vol_close*( ).

DISCUSSION

More details pertaining to this operation can be found in Appendix I on page 405.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

Appendix I on page 405.

NAME

vol_seek — Seek to a particular index within a fileset

SYNOPSIS

```
int vol_seek(
    /* IN */    struct volume     *vold,
    /* IN */    long              position,
    /* INOUT */ struct vol_handle *handle
    );
```

ARGUMENTS

vold	Fileset descriptor pointer.
position	canonical anode index within a fileset.
handle	pointer to the fileset iterator structure created for the given fileset to be filled in by this call.

DESCRIPTION

This call is used to seek to a desired *position* within the file table of the fileset associated with *vold*. The *handle* iterator structure is updated to reflect the new position.

DISCUSSION

The canonical inode number supplied in the position argument is converted to an LFS relative one and stored in the **vol_handle** .*index* field. As a general rule, this indicates the next file that will be operated upon.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL] The supplied anode index is an illegal value (It is less than VOL_ROOTINO.).

[VOL_ERR_EOF]
There are no valid objects on the fileset with an inode index greater than or equal to the supplied position argument.

[VOL_ERR_EOW]
Returned by the UFS volume operations when the position argument is greater than the size of the statically configured inode table.

SEE ALSO

vol_open( ).

NAME

vol_tell — Return the current index of the selected file within a fileset

SYNOPSIS

```
int vol_tell(
    /* IN */   struct volume     *vold,
    /* IN */   struct vol_handle *handle,
    /* OUT */  long              *position
    );
```

ARGUMENTS

vold	Fileset descriptor pointer.
handle	pointer to the fileset iterator structure created for the given fileset holding current position.
position	on return, set to the current anode index contained within *handle*.

DESCRIPTION

Given an iteration structure *handle* for the fileset described by *vold*, set *position* to the current index pointed to by *handle*.

DISCUSSION

This LFS relative anode index stored in the **struct vol_handle** *field (by vol_seek( ), vol_scan( ) or vol_create( )) is returned in the position argument. As discussed in vol_seek( ), this is an LFS-specific (non canonical) inode index.*

Note: *Note the discrepency with the behavior of vol_seek( ).*

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

vol_seek( ) for discrepancy in behavior with this (*vol_tell*( )) operation.

NAME

vol_scan — Read the file table for the given fileset at the current position, verifying that there is a file at the position

SYNOPSIS

```
int vol_scan(
    /* IN */    struct volume     *vold,
    /* IN */    long              position,
    /* IN */    struct vol_handle *handle
    );
```

ARGUMENTS

vold	Fileset descriptor pointer.
position	canonical anode index within a fileset.
handle	pointer to the fileset iterator structure associated with the given fileset described by *vold* to be filled in by this call.

DESCRIPTION

Set file descriptor fields in *handle* to describe the current file for the fileset described by *vold*, as specified by the associated iterator *handle*. If the position within *handle* is a legal one (it corresponds to an actual file within the fileset), then information about the file is written in *handle*. If the position within *handle* is *not* legal, then *vol_scan*( ) returns [VOL_ERR_EOF] (if the position is past the end of the fileset's file table) or [VOL_ERR_DELETED] (if the position happens to be empty).

DISCUSSION

This operation performs the same functions carried out by *vol_seek*( ) and, additionally, returns in the supplied **vol_handle** structure some information on the file selected by position.

The **vol_handle** *.type* field is set to the file's mode bits (including the type -- S_IFMT). The *.fid.Vnode* and *.fid.Unique* fields in the **vol_handle** are filled in (for *.Vnode*, from the canonical anode index in the position argument).

Note: The significance of the *vol_handle.fid.Vnode* field isn't actually mandated by DFS. This *vol_handle.fid* field will subsequently be passed back to *vol_xxx*( ) operations requiring a *fid* argument. The *vol_scan*( ) and *vol_create*( ) calls can use any encoding they wish for the *.Vnode* field (either a canonical/DFS or private/LFS index) as long as the other *vol_xxx*( ) operations accept the same encoding in *fid*s passed to them.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[VOL_ERR_EOF]
The supplied position argument is past the end of the fileset's file table.

[VOL_ERR_DELETED]
The supplied position argument is empty. The *vol_SEEK*( ) error values take precedence.

SEE ALSO

vol_seek( ) for additional error values returned.

NAME

vol_close — Finish an iteration for the given fileset

SYNOPSIS

```
int vol_close(
    /* IN */    struct volume     *vold,
    /* IN */    struct vol_handle *handle,
    /* IN */    long              isabort
    );
```

ARGUMENTS

vold	Fileset descriptor pointer.
handle	pointer to the fileset iterator structure associated with the given fileset described by *vold*.
isabort	If non-zero, this call is being made in order to abort an open fileset request (The *vol_open*( ) call has already occurred.).

DESCRIPTION

Indicate to the system that file-level operations on the fileset described by *vold* have completed. *handle* is updated so that it no longer refers to the fileset to be closed. Upon a successful *vol_close*( ), the given fileset is put back on line.

DISCUSSION

If this is an ordinary close (isabort equals 0) and the **vol_handle** .*voltype* field indicates that the VOL_DELONSALVAGE flag was set at *vol_open*( ) time, this flag is cleared in both *volp->v_stat_st.states* and in the on-disk fileset storage.

If *isabort* is non-zero, the fileset may be left marked in an inconsistent state (VOL_DELONSALVAGE) after it is closed.

See the discussion under *vol_open*( ).

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

See the DISCUSSION in *vol_open*( ).

NAME

vol_destroy — Delete an empty fileset

SYNOPSIS

```
int vol_destroy(
    /* IN */    struct volume     *vold
    );
```

ARGUMENTS

vold	Fileset descriptor pointer.

DESCRIPTION

This operation deletes a fileset that is empty. It contains no files or objects.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EIO]	An error was encountered (the fileset wasn't empty) and the operation did not complete successfully.

NAME

vol_deplete — Destroy a fileset

SYNOPSIS

```
int vol_deplete(
    /* IN */    struct volume     *vold
    );
```

ARGUMENTS

vold Fileset descriptor pointer.

DESCRIPTION

Destroy the fileset described by *vold*, including all of its files. All in-memory vnodes associated with this fileset are also deleted, and the *Fileset Registry*

In the interest of preventing system calls from taking an uncomfortable length of time, this operation may return before the fileset is completely destroyed. If this has occurred, the operation should return code [ELOOP], and the caller should simply call it again, until it stops returning [ELOOP].

DISCUSSION

This operation deletes all files on the specified fileset. Note that files are unconditionally deleted and do not undergo the normal Zero-Link-Count processing discussed in Section 15.6 on page 332. The fileset itself is not deleted: that task is performed by *vol_delete*( ). See the discussion under *Looping Operations* in Section 15.10 on page 335.

When this operation is first called to begin deleting a fileset, the VOL_DELONSALVAGE flag should be set in both *volp->v_stat_st.states* and in the on-disk fileset storage.

While DFS guarantees that this call will not be made on a fileset which is backing for another fileset, this fileset might currently be backed itself. Therefore, using the notation described in Section 15.3 on page 328 ,if F1 is the fileset being deleted:

```
    F1 ===> Fxxx    can happen
while
    Fxxx ===> F1    will never occur
```

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ELOOP] The operation has not finished. The *vol_deplete*( ) call should be made again.

[EIO] An error was encountered and the operation did not complete successfully.

SEE ALSO

Section 15.3 on page 328, *Fileset Clone Algorithms*, discusses backing filesets. Also, see the discussion under *Looping Operations* in Section 15.10 on page 335.

NAME

vol_attach — Initialize a fileset for use by DFS as the aggregate is attached

SYNOPSIS

```
int vol_attach(
    /* IN */    struct volume     *vold
    );
```

ARGUMENTS

vold Fileset descriptor pointer.

DESCRIPTION

Perform filesystem-specific initialization associated with bringing a fileset on-line while attaching an aggregate.

DISCUSSION

As each fileset is exported to DFS, a call is made to *vol_setdystat*( ) and *vol_attach*( ) - in that order. This occurs at aggregate attach time (following calls to *ag_attach*( ) and *ag_volInfo*( )) and at fileset creation time (during the call to *ag_volCreate*( ) -- consult its description).

Any LFS actions necessary before a fileset can be exported should be performed here. Such operations should be performed while a write lock is held on **volp->v_lock**.

This call succeeds if the VOL_DELONSALVAGE flag is set, even though the fileset may be in a corrupt state.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

ag_volCreate( ) and *ag_volInfo*( ).

NAME

vol_detach — Make a fileset unavailable for use by DFS as the aggregate is detached

SYNOPSIS

```
int vol_detach(
    /* IN */    struct volume     *vold,
    /* IN */    int               localrefs
    );
```

ARGUMENTS

vold	Fileset descriptor pointer.
localrefs	Count of references held by the caller.

DESCRIPTION

Perform filesystem-specific takedown associated with taking a fileset off-line while detaching an aggregate or destroying a fileset. Fails if the fileset descriptor is not idle, where ‘‘idle’’ is defined as having a reference count equal to the count of long-uterm references for the structure plus *localrefs*, the count of fileset references held directly in the caller.

DISCUSSION

As a fileset is made unavailable for use by DFS, calls are made to *vol_detach*( ) and *vol_freedystat*( ) - in that order.

Any EBUSY checks and other LFS-specific activity should be performed while a write lock is held on *volp->v_lock*.

A fileset can be exported (attached) to DFS and (or) locally mounted. Presumably, an LFS will defer any low-level "disconnect" until a fileset is neither locally mounted nor exported. If there are any vnodes that are in-use by DFS, this call should fail with EBUSY. (The *Vnode to LFS Association* section, Section 15.11 on page 336, mentions one case in which the file exporter allows volume operations such as this to come in while it has a held vnode.) Of course, it may not be possible to differentiate vnodes being used by DFS from those being used locally.

If a fileset is not mounted locally, any in-use vnodes are be due to DFS. If the fileset is mounted locally, it isn’t crucial that DFS-held vnodes be detected here.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EBUSY]	An error was encountered (the fileset descriptor wasn’t idle) and the operation did not complete successfully.

SEE ALSO

Section 15.11 on page 336.

NAME

vol_getstatus — Read specified fileset status block and return status

SYNOPSIS

```
int vol_getstatus(
    /* IN */    struct volume      *volp,
    /* OUT */   struct vol_status  *statusp
    );
```

ARGUMENTS

volp Fileset descriptor pointer.

statusp pointer to the fileset status structure to be filled in by this call.

DESCRIPTION

Place the current status of the fileset described by *vold* into the *statusp* fileset status block.

DISCUSSION

See Appendix J on page 409 for details concerning the processing this operation does.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EIO] An error was encountered and the operation did not complete successfully.

SEE ALSO

Appendix J on page 409.

NAME

vol_setstatus — Write (set) specified fileset status block

SYNOPSIS

```
int vol_setstatus(
    /* IN */    struct volume      *volp,
    /* IN */    long               mask,
    /* IN */    struct vol_status  *statusp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
mask	A mask consisting of a set of VOL_STAT_XXX bits.
statusp	pointer to the supplied fileset status structure.

DESCRIPTION

Set the fileset status block for the fileset described by *volp* to the contents of *statusp*, modifying only the status information indicated by bits set in *mask*.

DISCUSSION

See Appendix K on page 411 for details concerning the processing this operation does.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	A supplied status parameter is illegal or cannot be handled. For instance, if an attempt is being made to set both the VOL_RW and VOL_READONLY status flags.
[EIO]	An error was encountered and the operation did not complete successfully.

SEE ALSO

Appendix K on page 411.

NAME

vol_create — Create a file (or directory, symlink, and so forth) at a given position within a fileset

SYNOPSIS

```
int vol_create(
    /* IN */    struct volume      *volp,
    /* IN */    long               position,
    /* IN */    struct xvfs_attr   *xvattrp,
    /* INOUT*/  struct vol_handle  *handlep,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
position	A canonical anode index representing the position at which to create a file.
xvattrp	The vnode attribute structure giving the attributes to be applied to the new file.
handlep	A pointer to the fileset iterator structure associated with the fileset described by *volp*.
credp	A pointer to the credentials structure.

DESCRIPTION

Create and initialize a file at the given *position* within the fileset described by *volp*. The initial type, mode, and other attributes of the new file are given by *vattrp*. Anode descriptor information in *handle* is set to describe the new file. If the file at the given *position* exists, *vol_create*( ) returns [EEXIST] and does not reinitialize it.

DISCUSSION

See Appendix L on page 413 for a more detailed description of the processing done by this operation.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	A supplied position parameter is illegal or cannot be handled.
[EEXIST]	An error was encountered and the operation did not complete successfully. Specifically, the file already exists at the specified position.

SEE ALSO

Appendix L on page 413.

NAME

vol_read — Read data from the given file

SYNOPSIS

```
int vol_read(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* IN */    struct afsHyper    offset,
    /* IN */    long               length,
    /* IN */    char               *readbuff,
    /* IN */    struct ucred       *credp,
    /* OUT */   long               *amt_read
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
Fidp	An **afsFid** identifying the file ID of the file to be read.
offset	Byte position in the file at which the read is to commence.
length	The number of bytes to be read.
readbuf	A pointer to the buffer in which the file's data are to be deposited.
credp	A pointer to the credentials structure.
amt_read	Upon return, set to the number of bytes actually read from the specified file.

DESCRIPTION

Given a file described by *Fidp* within the fileset associated with *volp*, attempt to read *length* bytes, starting at byte position *offset*. The file data are placed into the buffer pointed to by *readbuff*, which must be at least *length* bytes long. Upon completion, *amt_read* is set to the number of bytes actually placed into *readbuff*. Other than the standard return code indicating success, and [EINVAL], *vol_read*( ) may return [EIO] if a lower-level read error occurred.

DISCUSSION

This call is used to read the contents of a symbolic link. It is the ONLY *vol_xxx*( ) operation that can do this.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	An attempt was made to read a FIFO, Block special device, CHAR special device or a directory.
[EIO]	An actual read error occurred.

NAME

vol_write — Write data to the given file

SYNOPSIS

```
int vol_write(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* IN */    struct afsHyper    offset,
    /* IN */    long               length,
    /* IN */    char               *writebuf,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
Fidp	An **afsFid** identifying the file ID of the file to be written.
offset	Byte position in the file at which the write is to commence.
length	The number of bytes to be written.
writebuf	A pointer to the buffer holding the data to be written to the file.
credp	A pointer to the credentials structure.

DESCRIPTION

Given a file described by *Fidp* within the fileset associated with *volp*, attempt to write *length* bytes, starting at byte position *offset*. The file data are taken from the buffer pointed to by *writebuf*, which must be at least *length* bytes long. Other than the standard return code indicating success, and [EINVAL], *vol_srite*( ) may return [EIO] if a lower-level write error occurred.

DISCUSSION

This call is used to write the contents of a symbolic link. It is the ONLY *vol_xxx*( ) operation that can do this.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	An attempt was made to write a FIFO, Block special device, CHAR special device or a directory.
[EINVAL]	An attempt was made to write to a file offset that is larger than the maximum supported file size. This can occur if the *length* argument is larger than 4 Gigabytes and the filesystem is limited to 32 bits.
[EIO]	An actual read error occurred.

NAME

vol_truncate — Truncate a file to a specified size

SYNOPSIS

```
int vol_truncate(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* IN */    struct afsHyper    newsize,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
Fidp	An **afsFid** identifying the file ID of the file to be truncated.
newsize	New size of *Fidp*'s file, in bytes.
credp	A pointer to the credentials structure.

DESCRIPTION

In the fileset described by *vold*, truncate the file associated with *Fidp* to *newsize* bytes.

DISCUSSION

The specified file is truncated to the given size. A file can have its size increased by this call. If so, the newly acquired file space is "sparse" and does not show up as allocated space (see the quota discussion in Section 15.7 on page 333) in the fileset.

If a directory is being truncated, the newsize argument MUST be zero. In this case, the directory is left in an "empty" state without entries for "." and ".

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	A newsize of something other than 0 is specified for a directory.
[EINVAL]	A newsize argument is supplied that is larger than the maximum supported file size. This can occur if the *length* argument is larger than 4 Gigabytes and the filesystem is limited to 32 bits.

SEE ALSO

Section 15.7 on page 333.

NAME

vol_delete — Delete a file at a given position within a fileset

SYNOPSIS

```
int vol_delete(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
Fidp	An **afsFid** identifying the file ID of the file to be truncated.
credp	A pointer to the credentials structure.

DESCRIPTION

Delete the file described by *Fidp* in the fileset associated with *volp*.

DISCUSSION

The specified file is unconditionally deleted. It does not undergo the normal Zero-Link-Count processing. The deletion is immediate. Any vnodes for the file will be in a dis-associated state (consult *Vnode to LFS Association*, Section 15.11 on page 336, and the discussion in *Zero Link Count Files*, Section 15.6 on page 332.

If a directory is being truncated, the newsize argument MUST be zero. In this case, the directory is left in an "empty" state without entries for "." and ".

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

Section 15.11 on page 336, *Vnode to LFS Association*, and also *Zero Link Count Files*, Section 15.6 on page 332.

NAME

vol_getattr — Get a file's attributes

SYNOPSIS

```
int vol_getattr(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* OUT */   struct xvfs_attr   *xvattrp,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp Fileset descriptor pointer.

Fidp An **afsFid** identifying the file ID of the file.

xvattrp Pointer to a returned vnode structure in which to record the file's attributes.

credp A pointer to the credentials structure.

DESCRIPTION

Read the attributes of the file indicated by *Fidp* in fileset *volp*, into the attribute structure *xvattrp*.

DISCUSSION

Except as noted below, the standard vnode attributes at *xvattrp->vattr* are returned in an obvious manner.

.va_fsid The value returned here should be chosen with care since it needs to be unique (per fileset). Note that a given aggregate (device) can hold several filesets. Although a individual LFS is free to use any appropriate algorithm to construct this field, the following is done by Episode:

```
.va_fsid = (volumeID.low   << 16) |
   (device major # << 8)  |
   (device minor #)
```

.va_blocks The blocks used value is based on the logical amount of disk space used by the file, as if there were no copy-on-write (clone) sharing taking place. This corresponds to the visible (vs actual allocated) quota on a fileset.

The extended attributes at *xvattrp->xvattr* are returned as described in the *VFS (Vnode) Interface and Operations*, Chapter 16 on page 415. The *credp* argument is used in the computation of the *.callerAccess* field.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

VFS (Vnode) Interface and Operations, Chapter 16 on page 415.

NAME

vol_setattr — Set a file's attributes

SYNOPSIS

```
int vol_setattr(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* IN */    struct xvfs_attr   *xvattrp,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
Fidp	An **afsFid** identifying the file ID of the file.
xvattrp	Pointer to a vnode structure from which to set the file's attributes.
credp	A pointer to the credentials structure.

DESCRIPTION

Set the attributes of the file indicated by *Fidp* in the fileset *volp*, according to the vnode attribute structure *xvattrp*. In this structure, a -1 in a field indicates that no new value is to be set for the corresponding attribute; this is the same convention as is used in *VOP_SETATTR*.

DISCUSSION

The following file attributes can be set by this call.

- From the standard vnode attributes at *xvattrp->vattr*:

.va_ctime	If the *.tv_sec* sub-field is not equal to –1.
.va_nlink	If not equal to –1. If the link count is being set to zero, the file will be deleted by *vn_inactive*( ) when its vnode reference count goes to zero. As described in the topic, *Zero Link Count Files*, Section 15.6 on page 332, such files are preserved across reboots and FSCKs until *vn_inactive*( ) explicitly deletes them.
.va_mode	If not equal to –1.
.va_uid	If not equal to –1.
.va_gid	If not equal to –1.
.va_size	If not equal to –1. If a directory is being operated on, an EISDIR is returned.
.va_mtime	If the *.tv_sec* sub-field equals 0 and the *.tv_usec* sub-field equals –1, the file's atime and mtime are both set to the current time. (System V style *utime*( ) call.) Alternatively, if the *.tv_sec* field is not equal to –1, the file's mtime is set.
.va_atime	If the .tv_sec sub-field is not equal to –1 and the above *.va_mtime* case does not supercede this one, the file's atime is set.

- From the extended vnode attributes at xvattrp->xvattr:

.volVersion	If the *.high* or *.low* sub-field is not equal to –1.
.dataVersion	If the *.high* or *.low* sub-field is not equal to –1.
.fileI	The anode generation value, from the *.low* sub-field, can be changed if it is not equal to –1. The *.high* field, the anode index, is ignored.

.clientOnlyAttrs If not equal to –1.

The file's ctime is NOT advanced by this call (except for an explicit setting).

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EISDIR]	An attempt is made to change the length of a directory via this call.
[EROFS]	The fileset is read-only.

SEE ALSO

Section 15.6 on page 332.

NAME

vol_getacl — Get a file's **ACL**

SYNOPSIS

```
int vol_getacl(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* OUT */   struct dfs_acl     *aclp,
    /* IN */    long               which,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
Fidp	An **afsFid** identifying the file ID of the file.
aclp	Pointer to a structure into which the ACL and its length are to be stored.
which	A value indicating the type of ACL that is being read.
credp	A pointer to the credentials structure.

DESCRIPTION

Read the **ACL** of the file indicated by *Fidp* in fileset *volp*, into the **ACL** structure *aclp*. For regular files this is unambiguous, but directories may have up to three different **ACL**s, and *which* indicates which one is to be read: the directory's own **ACL**, the default **ACL** for subdirectories, or the default **ACL** for subfiles.

DISCUSSION

The LFS ACL specification chapter should be consulted for information about the values discussed here. It is Chapter 12 on page 251. There is further information on DFS ACLs in Chapter 8 on page 155.

The **ACL** designated by the *which* argument is encoded as described in the LFS ACL specification chapter, Chapter 12 on page 251, and returned. The *which* argument can take the following values.

VNX_ACL_REGULAR_ACL
: The **ACL** for the object itself is returned.

VNX_ACL_DEFAULT_ACL
: The initial directory **ACL** for a directory object is returned. If the object in question is not a directory, a zero-length **ACL** is returned instead.

VNX_ACL_INITIAL_ACL
: The initial file **ACL** for a the object in question is not a directory. A zero-length **ACL** is returned instead.

If the requested **ACL** does not exist, a zero length **dfs_acl** is returned (*aclp->dfs_acl_len*).

Note: If *which* is VNX_ACL_REGULAR_ACL, the internal Episode implementation of **ACL**s makes itself apparent here.

Normally, **ACL** operations under Episode behave as follows. When one sets an ACL, the file's mode bits are changed to agree with the required ACL entries. When a file's mode bits are changed (**chmod**( )), its **ACL** is not physically changed on disk. Instead, the mode bits are "merged" into the **ACL** dynamically during **ACL** checks and when **ACL**s are returned to user space.

The *vol_getacl*( ) operation fetches the **ACL** as it resides on disk, without coercing the **ACL**'s required entries to agree with the file's mode bits (which might be different due to a **chmod**( ) operation that was performed after the **ACL** was initially applied).

Consult the DFS **ACL** information in Chapter 8 on page 155 under the topic, *Interaction of Filesystem* **ACL***s with UNIX Permission Bits*, and also Appendix A on page 221, *Mapping DFS* **ACL***s to UNIX mode bits*. See Chapter 9 on page 165 for information about the structures used by DFS for **ACL**s in memory.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL] The *which* argument does not equal one of the values mentioned in DISCUSSION.

SEE ALSO

Chapter 12 on page 251, Chapter 8 on page 155 and Appendix A on page 221.

NAME

vol_setacl — Set a file's **ACL**

SYNOPSIS

```
int vol_setacl(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *Fidp,
    /* IN */    struct dfs_acl     *aclp,
    /* IN */    long               index,
    /* IN */    long               which,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
Fidp	An **afsFid** identifying the file ID of the file.
aclp	Pointer to a structure from which the ACL is to be applied (or used).
index	Index of another file from which to copy the file's new ACL. This argument is presently UNUSED by this operation. The argument *aclp* is always used.
which	A value indicating the type of ACL that is to be set.
credp	A pointer to the credentials structure.

DESCRIPTION

Set the **ACL** of the file indicated by *Fidp* in the fileset *volp*. If *aclp* is non-null, the new **ACL** is read from it. If *aclp* is null, the new **ACL** is copied from the **ACL** associated with the file at *index*. Parameter *which* indicates which of the file's **ACL**s is to be set; for details see the description of *vol_getacl*( ).

DISCUSSION

As described under the *vol_getacl*( ) operation, the *which* argument indicates the type of **ACL** on the specified object that is to be set. The mode bits on the target object are NOT changed to agree with the **ACL** being applied: see the note below.

Note: If *which* equals VNX_ACL_REGULAR_ACL, the internal Episode implementation of **ACL**s makes itself apparent here.

Normally, **ACL** operations under Episode behave as follows. When one sets an ACL, the file's mode bits are changed to agree with the required ACL entries. When a file's mode bits are changed (**chmod**( )), its **ACL** is not physically changed on disk. Instead, the mode bits are "merged" into the **ACL** dynamically during **ACL** checks and when **ACL**s are returned to user space.

The *vol_setacl*( ) operation applies the **ACL** to the file without changing the file's mode bits (which might be different due to a **chmod**( ) operation that was performed). Later, when the **ACL** is used, the "current" mode bits *will* actually be used.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EIO]	The *which* argument equals VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL and the specified object is not a directory. This is

returned by Episode only.

[EINVAL] The *which* argument equals VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL and the specified object is not a directory.

[EINVAL] The *which* argument does not equal one of the values mentioned in DISCUSSION.

[EINVAL] The supplied **ACL** is missing a required field or is in an inconsistent or illegal state. Specifically, if:

- The **ACL**'s length is inconsistent with its contents.
- The *.mgr_type_field* is incorrect.
- The required entries are not all present.
- The *entry_type_foreign_user* and *entry_type_foreign_group* do not contain a realm UUID that differs from the default one.
- The *user_obj* entry does not grant *perm_control* rights.

SEE ALSO

vol_getacl( ). There is also a set of references there that are pertinent to this operation as well.

NAME

vol_clone — Create a cloned image of the files on a fileset in a different fileset

SYNOPSIS

```
int vol_clone(
    /* IN */    struct volume      *destvolp,
    /* IN */    struct volume      *srcvolp,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

destvolp	Fileset descriptor pointer for the destination fileset.
srcvolp	Fileset descriptor pointer for the source fileset.
credp	A pointer to the credentials structure.

DESCRIPTION

Make progress in making the entire fileset described by *destvolp* be a set of copy-on-write clones of all objects in the fileset described by *srcvolp*. The destination fileset is presumed to be empty before this call is first made.

In the interest of preventing system calls from taking an uncomfortable length of time, this operation may return before the fileset is completely processed. If this has occurred, the operation should return code [ELOOP], and the caller should simply call it again, until it stops returning [ELOOP].

DISCUSSION

Refer to the earlier topics on *Looping Operation Considerations*, Section 15.10 on page 335, and *Fileset Clone Algorithms*, Section 15.3 on page 328.

When the clone operation is first started, the target fileset specified by *destvolp* will be completely empty. When the clone operation has successfully completed, this target fileset will contain a "snapshot" of the files that reside on the source fileset identified by *srcvolp*.

The *.volversion* (VV) on the target (*destvolp*) fileset is not updated by this operation. The quota usage fields *.visQuotaUsage* and *.allocUsage* on the target (*destvolp*) fileset are updated as appropriate.

Both fileset header and individual file status on the source (*srcvolp*) fileset are unaffected by this call. When a clone operation is first started, the VOL_DELONSALVAGE flag should be set in both the target's on-disk fileset storage and in the *vol1p->v_stat_st.states*. When the clone has been successfully completed, the VOL_DELONSALVAGE flag should be cleared again. If the clone fails part way through, the fileset is left in a potentially inconsistent state with the VOL_DELONSALVAGE flag set.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ELOOP]	The clone operation has not yet completed. This call should be made again.
[EDQUOT]	The quota on the target *destvolp* has been exceeded. (This error should not occur if the quotas on the destination fileset have been properly established.)
[ENOSPC]	The space on the aggregate has been exhausted.

SEE ALSO

Refer to the topics on *Looping Operation Considerations*, Section 15.10 on page 335, and *Fileset Clone Algorithms*, Section 15.3 on page 328 for more information.

NAME

vol_reclone — Update a cloned image of the files on a fileset

SYNOPSIS

```
int vol_reclone(
    /* IN */    struct volume      *srcvolp,
    /* IN */    struct volume      *destvolp,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

destvolp	Fileset descriptor pointer for the destination fileset.
srcvolp	Fileset descriptor pointer for the source fileset.
credp	A pointer to the credentials structure.

DESCRIPTION

This call performs actions on each of a pair of filesets that are presumed to be in the copy-on-write relationship established by *vol_clone*( ). The *destvolp* parameter should refer to the fileset in which copy-on-write pointers were established by *vol_clone*( ) (as its *destvolp* parameter), and the *srcvolp* parameter should refer to the fileset to which the copy-on-write pointers were made by *vol_clone*( ) (as its *srcvolp* parameter).

All changes made to objects in the copy-on-write fileset (*destvolp*) are propagated to corresponding objects in the backing fileset (*srcvolp*). Any conflicting data that had been in the backing fileset is truncated and (or) deleted. The copy-on-write fileset is made to contain no data itself, but to be filled with copy-on-write pointers to the objects in the backing fileset.

In the interest of preventing system calls from taking an uncomfortable length of time, this operation may return before the fileset is completely processed. If this has occurred, the operation should return code [ELOOP], and the caller should simply call it again, until it stops returning [ELOOP].

DISCUSSION

Note: Note that the first two arguments have their order swapped from that in the *vol_clone*( ) operation.

The fileset identified by the *srcvolp* argument must be directly backed by the fileset identified by argument *destvolp* (F1 ===> F2).

This operation renews the clone relationship so that the backing fileset (destvolp) is a snapshot of the source (srcvolp) as it exists currently. Note that since the clone was originally created, the source fileset can have been modified with files being either modified or deleted.

If the filesets in question are named F1 and F2 (from srcvolp and destvolp, respectively), there can in general be other filesets in the backing hierarchy which are logically unaffected by this operation. For example, filesets Fxxx and Fyyy in the following relationship are not affected:

```
Fxxx  ===>  F1  ===>  F2  ===>  Fyyy
```

Note: Although DFS will never normally generate the above scenario, it could arise in error situations.

The *.volversion* (VV) on the backing (*destvolp*) fileset is not updated by this operation. The quota usage fields *.visQuotaUsage* and *.allocUsage* on the backing (*destvolp*) fileset are updated as appropriate.

Both fileset and individual file status on the source (*srcvolp*) fileset are unaffected by this call.

When a reclone is first started, the VOL_DELONSALVAGE flag is set in both the on-disk fileset storage and in the *destvolp->v_stat_st.states*. When the reclone has been successfully completed, the VOL_DELONSALVAGE flag is cleared again. If the reclone fails part way through, the fileset is left in a potentially inconsistent state with the VOL_DELONSALVAGE flag set.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ELOOP]	The clone operation has not yet completed. This call should be made again.
[EDQUOT]	The quota on the target *destvolp* has been exceeded. (This error should not occur if the quotas on the destination fileset have been properly established.)
[ENOSPC]	The space on the aggregate has been exhausted.
[EINVAL]	A file on a fileset indicated by *srcvolp* (F1) is backed by a different fileset than the one indicated by the *destvolp* (F2) argument.
[EFBIG]	A file on the backing fileset required deleting and cannot be deleted.

SEE ALSO

Refer to the topics on *Looping Operation Considerations*, Section 15.10 on page 335, and *Fileset Clone Algorithms*, Section 15.3 on page 328 for more information.

NAME

vol_unclone — Prepare to destroy a cloned image of the files on a fileset

SYNOPSIS

```
int vol_unclone(
    /* IN */    struct volume      *srcvolp,
    /* IN */    struct volume      *destvolp,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

srcvolp	Fileset descriptor pointer for the source fileset.
destvolp	Fileset descriptor pointer for the destination fileset.
credp	A pointer to the credentials structure.

DESCRIPTION

This call performs actions on each of a pair of filesets that are presumed to be in the copy-on-write relationship established by *vol_clone*( ). The *destvolp* parameter should refer to the fileset in which copy-on-write pointers were established by *vol_clone*( ) (as its *destvolp* parameter). The *srcvolp* parameter should refer to the fileset to which the copy-on-write pointers were made by *vol_clone*( ) (as its *srcvolp* parameter).

This call breaks the copy-on-write relationship between the two filesets. All copy-on-write data pointers in the copy-on-write fileset (*destvolp*) are replaced by ordinary data pointers, which are moved there from the backing fileset (*srcvolp*). The pointers are deleted from the backing fileset, thus truncating the objects it contains. After this operation completes successfully, the contents of the former backing fileset (*srcvolp*) are not necessarily meaningful; the former backing fileset is usually destroyed.

In the interest of preventing system calls from taking an uncomfortable length of time, this operation may return before the fileset is completely processed. If this has occurred, the operation should return code [ELOOP], and the caller should simply call it again, until it stops returning [ELOOP].

DISCUSSION

Note: Note that the first two arguments have their order swapped from that in the *vol_clone*( ) operation.

The fileset identified by *srcvolp* must be directly backed by the fileset identified by *destvolp* (F1 ===> F2).

This operation is made prior to deleting the backing fileset (*destvolp*) in order to "break" the copy-on-write relationship between files on it, files on the source fileset (*srcvolp*) and on any fileset that it is backing. Since the fileset is left in an inconsistent state (VOL_DELONSALVAGE) and is about to be deleted, the internal state of it and its files is undefined at completion. The important point is that the copy-on-write relationship with upstream and downstream filesets be broken and their quotas be adjusted. It is unspecified as to whether the no longer needed raw disk space is given back to the aggregate at *vol_unclone*( ) or at a*vol_delete*( ) time.

This operation should never fail due to a disk full condition. Specifically, it should NOT require additional disk space beyond what is already in use.

In theory, the quota on an upstream or downstream fileset could be exceeded as the copy-on-write relationship is broken. In this case, the unclone should proceed, exceeding (although updating) whatever quotas it encounters.

Note: Another possibility would be for the unclone to fail with [EDQUOT].

With the exception of copy-on-write related fields and quotas, this call does not affect the fileset status of upstream and downstream filesets (specifically: *.llFwdID* ,and *.llBackId*).

After the unclone operation completes successfully, the backing fileset will actually be deleted via *vol_destroy*( ).

Consider a fileset backing relationship as follows:

```
F1 ===> F2 ===> F3
```

Prior to deleting F2, DFS will call *vol_unclone(F1, F2, ..)* to break the copy-on-write relationships involving F2, resulting in the following:

```
F1 ===> F3          F2
```

If F3 were being deleted instead, DFS would call *vol_unclone(F2, F3, ..)* to break the copy-on-write relationship between F2 and F3, resulting in:

```
F1 ===> F2      F3 .
```

When an unclone is first started, the VOL_DELONSALVAGE flag is set in both the on-disk fileset storage and in the *destvolp->v_stat_st.states*. It remains in this state until the fileset is deleted.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ELOOP]	The clone operation has not yet completed. This call should be made again.
[EINVAL]	A file on a fileset indicated by *srcvolp* (F1) is backed by a different fileset than the one indicated by the *destvolp* (F2) argument.
[EFBIG]	A file on the backing fileset required deleting and cannot be deleted.

SEE ALSO

Refer to the topics on *Looping Operation Considerations*, Section 15.10 on page 335, and *Fileset Clone Algorithms*, Section 15.3 on page 328 for more information.

NAME

vol_vget — Return the vnode for a designated afsFid

SYNOPSIS

```
int vol_vget(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *fidp,
    /* OUT */   struct vnode       *vnodep
    );
```

ARGUMENTS

volp	Fileset descriptor pointer for a fileset.
fidp	Fileset descriptor (**afsFid**) for the desired file.
vnodep	A pointer to the vnode corresponding to the given **asfFid**, upon completion.

DESCRIPTION

Given a fileset descriptor pointer and a file ID, generate the associated vnode structure.

DISCUSSION

The returned vnode should be held (its *.v_count* incremented).

This call should return a vnode for a file even if its link count is zero. This is not an error. Consult the topic on *Zero Link Count Files*, Section 15.6 on page 332.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ESTALE]	The *.Unique* generation number in the **afsFid** does not agree with what is found in the selected vnode. Also, this code is returned on any errors that might be attributable to a bad *.vnode* or *.unique* **afsFid** field.

SEE ALSO

Refer to the topic on *Zero Link Count Files*, Section 15.6 on page 332.

NAME

vol_root — Return the vnode for the root directory on a particular fileset

SYNOPSIS

```
int vol_root(
    /* IN */    struct volume      *volp,
    /* OUT */   struct vnode       *vnodep
    );
```

ARGUMENTS

volp Fileset descriptor pointer for a fileset.

vnodep A pointer to the vnode corresponding to the given *volp*, upon completion.

DESCRIPTION

Given a fileset descriptor pointer, generate the vnode structure for the root of the fileset.

Note: This fileset operation is not used and may be obsolete.

DISCUSSION

The returned vnode should be held (its *.v_count* incremented).

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

None.

NAME

vol_isroot — Determine if a supplied **afsFid** represents the root directory on a fileset

SYNOPSIS

```
int vol_isroot(
    /* IN */    struct volume      *volp,
    /* IN */    struct afsFid      *fidp,
    /* OUT */   long               *flagp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer for a fileset.
fidp	Fileset descriptor (**afsFid**) for some file.
flagp	A pointer to the returned (boolean) flag, upon completion.

DESCRIPTION

Set the flag pointed to by *flagp* to 1 if the *Fidp*resents the root of the fileset, or to 0 if it represents some non-root file.

DISCUSSION

Only the .*Vnode* and .*Unique* fields of the **afsFid** are examined.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

None.

NAME

vol_getvv — Get Fileset Version number

SYNOPSIS

```
int vol_getvv(
    /* IN */    struct volume      *volp,
    /* OUT */   struct afsHyper    *vnp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer for a fileset.
vnp	The returned fileset version number.

DESCRIPTION

Put the fileset version number of the volume represented by *volp* in the location pointed to by *vnp*. (This is a special case of *vol_getstatus*( ).)

DISCUSSION

None.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vol_setdystat — Allocate private per-fileset filesystem-specific data area (if needed)

SYNOPSIS

```
int vol_setdystat(
    /* INOUT */ struct volume       *volp,
    /* IN */    struct vol_stat_dy  *dystatp
    );
```

ARGUMENTS

volp Fileset descriptor pointer for a fileset.

dystatp Pointer to dynamic area of fileset status block that has already been established by a call to *ag_volInfo*( ) or within the *ag_volCreate*( ) call.

DESCRIPTION

Allocate storage for file-system-specific data about a fileset that is to be attached, and put a pointer to that storage in the *Fileset Registry* entry that is being prepared. For **DCE LFS** filesets, the file-specific data includes the index of the fileset within the aggregate, the highest valid file index within the fileset, and an **DCE LFS** *handle* token representing the open fileset). The fiileset index field is initialized at this time from the corresponding field in *dystatp*, but the other fields are initialized during *vol_attach*( ).

DISCUSSION

If private LFS per-fileset data is required, it should be allocated and a pointer to it stored in the fileset structure (**struct volume**) .*v_fsDatap* field.

Note: Higher level DFS functions do not examine this field, so any desired storage allocation mechanism can be used.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

Refer to the topic, *Private LFS Fileset Data* in Section 15.12 on page 339.

NAME

vol_freedystat — Free private per-fileset filesystem-specific data area

SYNOPSIS

```
int vol_freedystat(
    /* INOUT */ struct volume     *volp
    );
```

ARGUMENTS

volp Fileset descriptor pointer for a fileset.

DESCRIPTION

Free the file-specific data area that was allocated by *vol_setdystat*.

DISCUSSION

Any storage allocated by *vol_setdystat*( ) should be freed. The **struct volume** .*v_fsDatap* field should be cleared as well.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

vol_setdystat( ).

NAME

vol_setnewvid — Set new Fileset's fileset ID

SYNOPSIS

```
int vol_setnewvid(
    /* INOUT */ struct volume      *volp,
    /* IN */    struct afsHyper    *idp
    );
```

ARGUMENTS

volp	Fileset descriptor pointer for a fileset.
idp	The returned Fileset ID.

DESCRIPTION

Set the ID of the fileset represented by *volp* to the ID in *idp*. This is called twice to carry out an exchange of fileset identities, generally after a clone fileset is created. For **DCE LFS**, it modifies entries in the mount table as well as modifying the fileset data structures.

DISCUSSION

The fileset ID of the specified fileset is changed to the specified value. Only LFS internal and on-disk structures should be updated by this call; specifically, the fileset ID in the **struct volume** structure should not be changed.

Note: This call is obsolete; *vol_swapids*( ) appears to be in use these days. It should be implemented, however, since DFS could choose to use it again in the future (to swap the IDs of filesets on different aggregates).

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vol_copyacl — Copy an ACL from one file to another

SYNOPSIS

```
int vol_copyacl(
    /* IN */    struct volume    *volp,
    /* IN */    struct afsFid    *Fidp,
    /* IN */    long             destw,
    /* IN */    long             index,
    /* IN */    long             srcw,
    /* IN */    struct ucred     *credp
    );
```

ARGUMENTS

volp Fileset descriptor pointer for a fileset.

fidp Fileset descriptor (**afsFid**) identifying the destination file to which the **ACL** should be applied.

destw A value indicating the type of **ACL** on the destination that is to be set.

index A canonical anode index, as used by *vol_seek*( ) or *vol_scan*( ), that identifies the source file from which the **ACL** is to be copied.

srcw A value indicating the type of **ACL** on the source file that is to be copied (from).

credp A pointer to a credentials structure.

DESCRIPTION

Set an **ACL** of the file indicated by *Fidp* in the fileset *volp* by copying it from an ACL of the file indicated by *index*. The **ACL** copying may be performed by sharing a reference to an **ACL**, if the underlying file system supports such shared references, but there is no semantic requirement for such sharing. Parameter *destw* indicates which of the file's **ACL**s is to be set. Parameter *srcw* indicates which of the source file's **ACL**s is to be copied. For details, see the description of *vol_getacl*( ).

DISCUSSION

The *srcw* argument selects an **ACL** on the source object, identified by the *index* argument:

```
VNX_ACL_REGULAR_ACL     ACL on the object itself.
VNX_ACL_DEFAULT_ACL     Initial directory ACL on a directory.
VNX_ACL_INITIAL_ACL     Initial file ACL on a directory.
```

This **ACL** is copied to the destination object, identified by the *Fidp* argument. The type of **ACL** to be set is indicated by the *destw* argument which has the same VNX_ACL_XXX values listed above.

It is legal for the source **ACL** to not exist; in this case, the destination **ACL**, if any, is deleted and the destination is left without an **ACL** of the specified type.

Except as noted further below, the *srcw* and *destw* arguments can each take any of the legal values. For example, it is legal to copy an *initial directory* **ACL** of a directory to the *regular* **ACL** of a file.

Note: If *srcw* or *destw* equal VNX_ACL_REGULAR_ACL, the internal DFS implementation of ACLs makes itself apparent here. The normal, vnode **ACL** operations under Episode behave as follows:

When one sets an **ACL**, the file's mode bits are changed to agree with the required entries. When a file's mode bits are changed (*chmod*( )), its **ACL** is not phyically changed on disk. Instead, the mode bits are merged with the **ACL** dynamically during **ACL** checks and when **ACL**s are returned to user space.

As is the case for *vol_getacl*( ) and *vol_setacl*( ), this operation entirely ignores the mode bits on the file (either merging in if *srcw* is VNX_ACL_REGULAR_ACL or setting if *destw* is VNX_ACL_REGULAR_ACL).

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[VOL_ERR_EOF] There are no valid objects on the fileset with an anode index greater than or equal to the supplied source index.

[EINVAL] The supplied source *index* is an illegal value. It is a too-small or a negative value other than VOL_ROOTINO.

[VOL_ERR_DELETED] No valid file has been selected by the position argument (*index*). The preceeding errors, [VOL_ERR_DELETED] and [EINVAL] take precedence.

[EINVAL] If either the *srcw* or *destw* arguments are not one of the leagl VNX_ACL_XXX values; or the *srcw* or *destw* argument is either VNX_ACL_INITIAL_ACL or VNX_ACL_DEFAULT_ACL and the destination object is not a directory; or a supplied **ACL** is inconsistent (see *vol_setacl*( )).

SEE ALSO

vol_getacl( ), *vol_setacl*( ). Consult the DFS **ACL** information in Chapter 8 on page 155 under the topic, *Interaction of Filesystem* **ACL***s with UNIX Permission Bits*, and also Appendix A on page 221, *Mapping DFS* **ACL***s to UNIX Mode Bits*. **See Chapter 9 on page 165 for information about the structures used by DFS for ACLs in memory.**

NAME

vol_concurr — Determine allowable concurrency on a fileset (being opened)

SYNOPSIS

```
int vol_concurr(
    /* IN */    struct volume    *volp,
    /* IN */    long             type,
    /* IN */    long             errorType,
    /* OUT */   char             *concurr
    );
```

ARGUMENTS

volp	Fileset descriptor pointer.
type	A mask consisting of VOL_OP_XXX bits that indicate the activities that will be performed on the open fileset.
errorType	This argument is unused.
concurr	Location to store descriptor for the allowable kinds of calls that can take place conucrrently with the set of planned operations. They are VOL_CONCUR_ALLOPS, VOL_CONCUR_READONLY, and VOL_CONCUR_NOOPS.

DESCRIPTION

This function is given a bit mask giving the set of fileset operations to be carried out on the given fileset. It returns an abbreviated description of the class of vnode operations that may be executed concurrently with that set of fileset operations.

The value of the input *type* parameter is constructed as the bitwise inclusive-OR of the VOL_OP_XXX definitions from the include file **<volume.h>**, such as VOL_OP_SCAN.

DISCUSSION

This operation examines the *type* argument and returns a VOL_CONCURR_XXX value according to the same rules that are listed under *vol_open*( ).

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

vol_open( ).

NAME

vol_swapids — Swap fileset identifiers of two filesets

SYNOPSIS

```
int vol_swapids(
    /* IN */    struct volume      *vol1p,
    /* IN */    struct volume      *vol2p,
    /* IN */    struct ucred       *credp
    );
```

ARGUMENTS

vol1p	Fileset descriptor pointer for a fileset.
vol2p	Fileset descriptor pointer for aanother fileset.
credp	A pointer to the credentials structure.

DESCRIPTION

This operation exchanges the fileset IDs of the two filesets pointed to. Both filesets are of the same type.

DISCUSSION

This operation is usually called following a clone operation in order to swap the identities of two filesets. In addition to updating any relevant LFS-specific data structures, the volume IDs (as found in the fileset (**struct volume**) structure *.vol_stat_st.volId* field) should be swapped in on-disk storage for the two filesets. The fileset structure itself should not be changed; that will be taken care of by the caller of this operation.

The two filesets in question will usually reside on the same aggregate (by virtue of being clones).

Note: Currently, Episode DOES allow the filesets to reside on different aggregates. It is the discretion of the LFS to either restrict the two filesets to the same aggregate or not.

Assume that vol1p and vol2p, prior to this call, referred to filesets as follows:

vol1p fileset index F1 on some aggregate, where say, the fileset ID = X

vol2p fileset index F2 on some aggregate, where say, the fileset ID = Y

After this call completes, fileset F1 will have an ID of Y and fileset F2 will have an ID of X. An active vnode which, prior to this call, referred to an object on fileset F1 will, after this call, refer to an object on fileset F2.

The contents of the *vol1p* and *vol2p* volume structures, with the exception of the fileset ID fields, will be swapped by the DFS layer once this call returns.

Note: If, in the future, filesets are allowed to reside on different aggregates, atomicity concerns arise unless the aggregates can be updated atomically. In this case, the aggregate holding *vol1p* should be updated before the one holding *vol2p*. This rule permits higher level recovery code to operate and take advantage of this ordering.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vol_sync — Sync the given fileset

SYNOPSIS

```
int vol_sync(
   /* IN */ struct volume *vold,
   /* IN */ int guarantee
   );
```

ARGUMENTS

vold Fileset descriptor pointer.

guarantee Describes the type of sync to be done, choose from one of: **VOL_SYNC_COMMITSTATUS**, **VOL_SYNC_COMMITMETA** or **VOL_SYNC_COMMITALL**.

DESCRIPTION

Sync the state of the fileset to the permanent storage, according to the *guarantee* parameter:

VOL_SYNC_COMMITSTATUS
The fileset's status information is written to permanent storage. The call does not return until the I/O has completed.

VOL_SYNC_COMMITMETA
All dirty meta-data is written to permanent storage. The call does not return until the I/O has completed.

VOL_SYNC_COMMITALL
All dirty data (both meta-data and user data) is written to permanent storage. The call does not return until the I/O has completed.

In all cases, this call does not return until the I/O operations have completed. The ability to write *just* the requested types of data benefits certain of the DFS fileset operations.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EINVAL] The **guarantee** argument is not one of VOL_SYNC_COMMITSTATUS, VOL_SYNC_COMMITMETA or VOL_SYNC_COMMITALL.

SEE ALSO

The definitions of the VOL_SYNC_XXX types can be found in Section 13.3.20 on page 288.

NAME

vol_pushstatus — Serialize fileset status updates

SYNOPSIS

```
int vol_pushstatus(
   /* IN */ struct volume *vold,
   );
```

ARGUMENTS

vold Fileset descriptor pointer.

DESCRIPTION

Serialize subsequent changes to a fileset's status with respect to prior changes. This operation guarantees that any preceding modifications of the fileset's status will be serialized with respect to later meta-data updates.

DISCUSSION

When this call returns, the LFS guarantees that any operations started subsequently on the fileset will not commit to disk unless any that ended prior to this call commit as well.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vol_readdir — Read entries from a directory

SYNOPSIS

```
int vol_readdir(
   /* IN */    struct volume *vold,
   /* IN */    struct afsFid *Fidp,
   /* IN */    u_long bufSize,
   /* IN */    char *bufferp,
   /* IN */    struct ucred *credp,
   /* INOUT */ struct afsHyper *positionp,
   /* OUT */   u_long *numEntriesp
   );
```

ARGUMENTS

vold	Fileset descriptor pointer.
Fidp	File ID of the directory.
bufSize	Size, in bytes, of *bufferp*.
bufferp	Pointer to buffer of *bufSize* bytes into which the directory entries are to be deposited.
credp	Credential structure.
positionp	Position at which to start reading. This parameter should be set to zero (0) on the first call for a given directory. The value of *positionp* will be modified by each call, so that, if passed in to a subsequent call, the directory entries will be fetched starting where the last call ended.
numEntriesp	Set to indicate the number of directory entries that are being returned.

DESCRIPTION

This operation returns directory entries in a system-independent representation. The buffer is filled with **vol_dirent** structures, as defined in Section 13.3 on page 286. Each directory entry is returned on a 4 byte boundary.

This operation is meant to be called in a loop until **numEntriesp* is zero (0).

DISCUSSION

Appendix H on page 403 contains the fields as they should be filled in by this function.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EINVAL]	(This error is optional. The **DCE LFS** does not return this value.) If the supplied buffer is not aligned on a 0-modulo-4 byte boundary.
[EINVAL]	The specified object is not a directory.
[EINVAL]	A position argument is supplied that is larger than the maximum supported file size.

Note: The intent is to catch position arguments that are larger than 4-Gigabyes on file systems limited to 32-bits.

NAME

vol_appenddir — Append entries to a directory

SYNOPSIS

```
int vol_appenddir(
   /* IN */    struct volume *vold,
   /* IN */    struct afsFid *Fidp,
   /* IN */    u_long *numEntriesp
   /* IN */    u_long bufSize,
   /* IN */    char *bufferp,
   /* IN */    int  preservedOffsets,
   /* IN */    struct ucred *credp,
   );
```

ARGUMENTS

vold	Fileset descriptor pointer.
Fidp	File ID of the directory.
numEntriesp	Set to indicate the number of directory entries that are to be appended.
bufSize	Size, in bytes, of *bufferp*.
bufferp	Pointer to buffer of *bufSize* bytes into which the directory entries are to be appended.
preserveOffsets	If true, try to preserve the entries' offsets; otherwise, strictly append.
credp	Credential structure.

DESCRIPTION

This operation appends entries to a directory. The buffer is expected to contain directory entries in the system-independent representation defined by **struct vol_dirent** (see Section 13.3 on page 286).

DISCUSSION

Consult the description under the *vol_readdir*( ) operation. This function is the inverse of that operation. Directory entries in the format of the entries in **struct vol_dirent** in the supplied buffer, *bufferp*, are added to the directory.

The **inode** link counts for files being appended are not updated by the operation of this function. If the *preserveOffsets* argument is zero, the entries can be placed anywhere within the directory. If the *preserveOffsets* argument is non-zero, the entries should be placed at the directory location indicated by the *.offset* field in the **struct vol_dirent**. This will only be the case if the source (the *vol_readdir*( )) and the destination (the *vol_appenddir*( )) aggregates are of the same type.

Entries for "." and ".." can be added with this call since directories created by the *vol_create*( ) function operation initially do not contain them.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EINVAL] The specified object is not a directory.

Note: Episode currently does not perform this check.

[EEXIST]	A name being added already exists in the directory.
	Note: Episode currently does not perform this check.
[E2BIG]	The end of the supplied buffer is reached before the specified number of directory entries is processed.
[EINVAL]	An inconsistency is encountered in a directory entry. For example, if the *.offset* field in the **struct vol_dirent** is not a legal encoding.
[EINVAL]	*preserveOffsets* is non-zero and the entry cannot be placed at the desired location (for instance, something else is there).

SEE ALSO

vol_readdir( ).

NAME

vol_bulksetstatus — Set statuses of multiple filesets atomically

SYNOPSIS

```
int vol_bulksetstatus(
   /* IN */    unsigned int           arrayLen,
   /* IN */    struct vol_statusDesc  *statusArray,
   );
```

ARGUMENTS

arrayLen The number of elements in *statusArray*. This number can range from 1 through VOL_MAX_BULKSETSTATUS

statusArray An array containing one status descriptor for each volume being updated. It describes which filesets' statuses should be updated and what their new values should be.

DESCRIPTION

This operation atomically updates the status of multiple filesets and possibly performs a fileset ID swap. If two of the **vol_statusDesc** structures indicate that their respective filesets should have a new ID, the IDs of the two filesets are swapped. See Section 13.3.7 on page 282 for the definition of the **vol_statusDesc** structure and also the definition of VOL_MAX_BULKSETSTATUS.

DISCUSSION

This operation atomically updates the status for a number of filesets. Either all the changes are made or none of them are. The specified filesets must all reside on the same aggregate.

Each entry in the supplied *statusArray* argument contains the information necessary to update a single fileset. The fields have the following meanings:

volp pointer to a fileset structure

vsd_mask a mask (VOL_STAT_xxx flags) indicating the particular status fields to be set

vsd_status The volume status being set

If this call is used to *change* the fileset IDs of two filesets (VOL_STAT_VOLID mask bits), an identity swap is being performed and the comments found in the *vol_swapids*( ) function apply as well.

Note: The emphasis here is on *change*. The LFS can safely ignore attempts to set a fileset ID to its current value.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

[EXDEV] The indicated filesets do not all reside on the same aggregate.

[EINVAL] The two filesets being updated have not had their IDs swapped. Otherwise, the aggregate would be left with two filesets having possibly the same ID. This case can be returned if any fileset ID (VOL_STAT_VOLID) is being *changed*, meaning that an identity swap is being performed and the fileset IDs have not been successfully swapped.

SEE ALSO

See the description under the *vol_setstatus*( ) operation for the details of how each fileset is updated. Also, see the *vol_swapids*( ) function for details on identity swap.

NAME

vol_getzlc — Get a vnode from fileset zero-link-count (ZLC) list

SYNOPSIS

```
int vol_getzlc(
   /* IN  */     struct volume   *vold,
   /* INOUT */   unsigned long   *iterator,
   /* OUT */     struct vnode    **new_vnode,
   );
```

ARGUMENTS

volp	Fileset descriptor pointer.
iterator	Iterator used to enumerate the files on the fileset ZLC list.
new_vnode	Held vnode for the specified ZLC file, or NULL if there are no more files to be enumerated.

DESCRIPTION

Return a held vnode representing a file on the fileset's zero-linkcount (ZLC) file list. It is used at **attach** time following the *vol_attach*( ) volume operation so that fileset ZLC files can be added to the DFS ZLC list for use by remote clients. The iterator, **iterator*, should be initialized to zero prior to the first call to this function. It will be updated by every call to this operation so this function can locate the "*next*" ZLC file for the next call. There are no more files to be enumerated when *new_vnode* comes back NULL. At this point, the return value will also be zero.

DISCUSSION

Consult the Zero-Link-Count file discussion earlier in this document. (See Section 15.6 on page 332.)

Since this operation is performed as each fileset is attached, performance is a consideration and the time required to identify these zero-link-count files must be kept in mind. As mentioned earlier in Section 15.6 on page 332, DFS will not make this call on a readonly (*.readonly* replica or *.backup* clone) fileset.

RETURN VALUE

This function is always successful. If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function was successful.

ERRORS

None.

SEE ALSO

See Section 15.6 on page 332 for information on zero-link-count files.

NAME

vol_getnextholes — Identify allocation holes in files.

SYNOPSIS

```
int vol_getzlc(
   /* IN */    struct volume        *vold,
   /* IN */    struct afsFid        *Fidp,
   /* INOUT */ struct vol_NextHole  *iterp,
   /* IN */    struct ucred         *credp
   );
```

ARGUMENTS

vold	Fileset descriptor pointer.
Fidp	An afsFID (file ID) of the directory.
iterp	Descriptor for a set of allocation holes. See DISCUSSION below.
credp	Credential structure pointer.

DESCRIPTION

Refer to Section 13.3.3 on page 277 for a description of the **struct vol_NextHole** structure used as the INOUT parameter *iterp*.

On each call, the filesystem determines whether there are any allocation holes with byte addresses beginning at or after the value of **startPoint**. If there are any, the filesystem describes between 1 and **VOLHOLE_MAX_HOLES** of them in the **holes** array by giving their starting byte addresses and their length in bytes, and by placing the count of filled-in allocation hole descriptions in the **outCount** field. Further, if there are no further allocation holes beyond those that are being described in the **holes** array, the filesystem should set the **VOLHOLE_FLAG_LAST** bit in the **flags** field; otherwise, if hole descriptors are being returned and **VOLHOLE_FLAG_LAST** is *not* set, the filesystem updates the **startPoint** to point past the last allocation hole being returned. If there are no allocation holes meeting the criteria (starting at or past the given value of **startPoint**), a zero is placed in the **outCount** field and the **VOLHOLE_FLAG_LAST** bit is set in the **flags** field.

All allocation holes for a file may be discovered by making a sequence of calls, where for the first call, the **startPoint** field of the **struct vol_NextHole** structure is zeroed.

DISCUSSION

For each hole returned, **.holes[]**.*holeStart* gives the file offset of the hole returned, and **.holes[]**.*holeLen* gives the length in bytes of the hole.

This function is useful for avoiding backing up, copying (during a move) and restoring regions within a file that are sparse.

This function is currently *unimplemented* and returns [ENOSYS].

RETURN VALUE

This function presently never succeeds as it is unimplemented. If this function were to succeed (in the future), it would return a value of zero.

ERRORS

[ENOSYS]	This function always returns this error as it is currently unimplemented.

15.15 Fileset Registry Array Functions

Descriptions of the *Fileset Registry* Functions follow. These are exported functions that permit a client to address the registry. They consist of functions to enter, delete and look up entries (filesets) in the *Fileset Registry*.

NAME

volreg_Enter — Enter a fileset and its information into the Fileset Registry

SYNOPSIS

```
int volreg_Enter(
    /* IN */ afsHyper      *avolid,
    /* IN */ struct volume *avolP,
    /* IN */ char          *avolname
    );
```

ARGUMENTS

avolid	The Fileset ID to enter.
avolP	Pointer to the above fileset's descriptor.
avolname	Fileset's name.

DESCRIPTION

Given fileset and aggregate IDs, along with a pointer to the associated fileset information, create an entry in the *Fileset Registry* and insert the given information into it.

DISCUSSION

None.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EEXIST]	The fileset was already in the *Fileset Registry* table.

NAME

volreg_Delete — Delete a fileset entry from the Fileset Registry

SYNOPSIS

```
int volreg_Delete(
    /* IN */ afsHyper      *avolid,
    /* IN */ char          *avolname
    );
```

ARGUMENTS

avolid The Fileset ID to delete.

avolname Above Fileset's name.

DESCRIPTION

Delete the *Fileset Registry* entry (if any) that corresponds to the given fileset and aggregate ID pair.

DISCUSSION

None.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

volreg_Lookup — Find the entry in the Volume Registry

SYNOPSIS

```
int volreg_Lookup(
    /* IN */  struct afsFid  *afidP,
    /* OUT */ struct volume  **avolPP,
    );
```

ARGUMENTS

afidP	The File ID to look up.
avolPP	Set to the address of the fileset descriptor associated with the given file.

DESCRIPTION

Given a file ID, find the entry in the Volume Registry corresponding to the fileset containing the file and return pointers to the associated **struct volume** fileset structure and the address of the file's vnode pointer.

DISCUSSION

None.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	The file ID has a bad fileset ID.
[ENODEV]	The file ID could not be found.

Appendix G

Exporting the Filesets in an Aggregate

G.1 Fileset Export Steps

The steps involved in exporting an aggregate's filesets are the following.

- The *dfsexport* command reads the **dfstab** file and collects the appropriate aggregate paramaters (device pathname, aggregate name, aggregate type, aggregate Id) into an argument structure. See Section 13.2.16 on page 273 for a definition of the **dfstab** entry structure.
- For an exported UFS volume, additional arguments which will be interpreted by the UFS-specific file system code, are collected into an auxiliary structure. See Section 13.2.17 on page 274 for a definition of the **ufs_astab** entry structure used for UFS volumes.

  Note: Other LFS implementations could, if desired, pass in additional information this way: via an argument structure that is interpreted by only the dfsexport command and the file system dependent code (with appropriate modifications to suit this arrangement).
- This aggregate argument structure and optionally, the file system dependent auxiliary one, is passed into the kernel to execute an aggregate attach operation.
- An *ag_attach*( ) call to the file system-dependent aggregate code is made.
- This is followed by a series of *ag_volInfo*( ) aggregate operations which are used to obtain the identities of the filesets that are present. Volume calls to *vol_setdystat*( ) and *vol_attach*( ) are made for each fileset discovered.
- At this point, the filesets are considered exported to DFS and can be accessed at any time.
- In order to have file system syncs work properly for filesets that are DFS exported but not locally mounted, the aggregate (note: aggregate, not fileset!) is specially mounted at **/opt/dcelocal/var/dfs/aggrs/<aggregate-name>**. This is discussed in greater detail in *Aggregate Mounts*, Section 14.3 on page 305.

An LFS may permit individual filesets to be mounted for local access outside of DFS. Episode allows this either before or after they have been exported to DFS.

Note: This local mount mechanism is primarily an issue between the local operating system (OS) and the LFS with, hopefully, marginal impact on DFS.

Similar steps occur when an aggregate is detached from (made unavailable to) DFS. What the LFS observes is the following (omitting miscellaneous calls to *ag_hold*( ) (and) *ag_rele*( ) as well as to *vol_hold*( ) (and) *ag_rele*( )):

- The *dfsexport* **-detach** command first verifies that filesets on the aggregate are idle by revoking all tokens for files on them. An optional force argument to the command specifies that tokens are to be forcibly revoked. This optional option should be used with caution, as it forces an aggregate (or partition) to be detached even if all tokens cannot be revoked. This can result in users accessing the data from the aggregate that are unable to save that data.
- A series of *ag_volinfo*( ) calls are made in order to ascertain the identities of the filesets on the aggregate.

- For each of these filesets, calls to *vol_detach*( ) and *vol_freedystat*( ) are made -- in that order.
- Finally, an *ag_detach*( ) call occurs.

Note: DFS is currently re-working the mechanics of fileset export and attachment (more cleanly separating the two notions). It is not known when these will appear in the general DFS product.

Appendix H

Filled Values for **vol_dirent** *Fields*

H.1 Returned Values

Refer to Section 13.3 on page 286 for the definition of the **struct vol_dirent** structure.

Fields within the **vol_dirent** structure should be filled in as follows.

.offset — An indication of where within the directory this entry was found. The intention is that if this offset is passed back via the *vol_appenddir*( ) operation during a fileset move, the directory entry will go back to the "same" location. This allows in-progress (interrupted by the move) directory scans to be oblivious to the fact that the directory has moved. This functionality is only required when the *vol_readdir*( ) and *vol_appenddir*( ) operations were performed on aggregates of the same type. (That is, the encoding of this *.offset* is LFS-private.)

Additional discussion of directory offsets and their properties can be found under the topic, *Directory Offsets*, Section 16.5.1 on page 423, and also under *vn_readdir*( ), Section 16.7 on page 447.

.vnodeNum — A canonical inode (anode) index in which *-1* refers to the root directory, *0* refers to the first inode after that, and so forth.

.codesetTag — For the current implementation, this should be set to *0*.

.reclen — The total length of this entry, rounded up to a 4 byte boundary. (Therefore it is the distance from the start of this entry to the next one in a supplied buffer.)

.namelen — The length of the file name field, not including the terminating null character.

.name — A variable sized file name, including a terminating null character.

As many entries as will fit in the supplied buffer, or as remain in the directory, are actually read.

On return, the count pointed at by the *numentriesp* argument is set to the number of actual directory entries returned. If there are no remaining directory entries, this count should be set to 0.

The *positionp* argument, an **afsHyper**, will be set to zero to indicate that the directory should be read from its beginning. On return, it should be updated by the LFS so that a subsequent call with it will continue with the "next" entry in the directory. DFS makes no assumptions regarding the ordering of entries within a directory. There is no reason to require the first two entries returned for a directory to be "." and "..".

Note: Episode rounds the supplied output buffer size down to a multiple of 512 bytes. The *last* directory entry returned has its *.recordlen* field incremented to cover the remaining space within the final 512 byte "block". (It is set to the next 512 byte boundary)

It is believed that this behavior exists only to allow the file exporter to produce meaningful directory offsets on platforms whose native entries do not contain such a field. This scheme is not required. There does not appear to be any reason for an LFS to mimic this 512-byte block behavior.

Appendix I

Values for **vol_open**

I.1 The Type Argument

The individual VOL_OP_XXX bits in the type argument indicate the types of *vol_xxx*( ) operations that might be performed on the fileset while it is open. (A user space utility wishing to open a fileset actually specifies a set of VOL_SYS_XXX values, each of which has been **#define**'d to a corresponding VOL_OP_XXX value.) The VOL_SYS definitions can be found in Section 13.3.19 on page 287. Before a subsequent *vol_xxx*( ) operation is executed, DFS will ensure that the fileset was opened with the appropriate VOL_OP_XXX flag. The following table indicates the VOL_OP_XXX flags that are required for each *vol_xxx*( ) operation. Operations that aren't mentioned perform no check (either because they can be performed on any fileset that is open or because they are invoked as part of a larger DFS operation such as attaching or opening a fileset).

```
vol_seek            VOL_OP_SEEK
vol_tell            VOL_OP_TELL
vol_scan            VOL_OP_SCAN
vol_destroy         VOL_OP_DESTROY
vol_deplete         VOL_OP_DEPLETE
vol_getstatus       VOL_OP_GETSTATUS
vol_setstatus       VOL_OP_SETSTATUS
vol_create          VOL_OP_CREATE
vol_read            VOL_OP_READ
vol_write           VOL_OP_WRITE
vol_truncate        VOL_OP_TRUNCATE
vol_delete          VOL_OP_DELETE
vol_getattr         VOL_OP_GETATTR
vol_setattr         VOL_OP_SETATTR
vol_getacl          VOL_OP_GETACL
vol_setacl          VOL_OP_SETACL
vol_clone           VOL_OP_CLONE
vol_reclone         VOL_OP_RECLONE
vol_unclone         VOL_OP_UNCLONE
vol_getvv           VOL_OP_GETSTATUS && VOL_OP_SETSTATUS
vol_copyacl         VOL_OP_COPYACL
vol_swapids         VOL_OP_SWAPIDS
vol_sync            VOL_OP_SYNC
vol_pushstatus      VOL_OP_PUSHSTATUS
vol_readdir         VOL_OP_READIR
vol_appenddir       VOL_OP_APPENDDIR
vol_bulksetstatus   VOL_OP_SETSTATUS
vol_getnextholes    VOL_OP_GETNEXTHOLES
```

User space code uses the VOL_OP_NOACCESS flag as a general mechanism to lock out all other fileset (*vol_xxx*( )) and vnode accesses to a fileset.

I.1.1 Concurrency

Based upon the *type* argument, this call establishes the *volp->v_stat_st.concurrency* field which controls the extent to which vnode operations can proceed on the fileset while it is open. Higher level code in DFS guarantees that a given fileset will only be in use (*vol_open*( )'d) by one utility (such as **fts**) at a time.

If there are no restrictions on vnode operations which might proceed in parallel, the *.concurrency* field should be set to VOL_CONCUR_ALLOPS.

Note: Episode does this if only VOL_OP_GETSTATUS is set.

If read-only type vnode operations are to be allowed to proceed in parallel, the .concurrency field should be set to VOL_CONCUR_READONLY.

Note: Episode does this if only bits from the following list are specified.

VOL_OP_GETSTATUS	VOL_OP_GETATTR
VOL_OP_GETACL	VOL_OP_SEEK
VOL_OP_READ	VOL_OP_TELL
VOL_OP_SCAN	VOL_OP_READDIR
VOL_OP_GETNEXTHOLES	VOL_OP_SETSTATUS
VOL_OP_SYNC	VOL_OP_PUSHSTATUS

If *no* vnode operations are to be allowed to proceed in parallel, the *.concurrency* field should be set to VOL_CONCUR_NOOPS.

Note: Episode does this if any other VOL_OP_XXX bits are specified.

WARNING: DFS does allow vnode operations initiated by the process that has the volume open to proceed!

I.1.2 Handling Inconsistent State

If any of the bits listed below are set in the type argument, the VOL_DELONSALVAGE flag should be set in both *volp- >v_stat_st.states* and in the on-disk fileset storage. (This indicates that the volume may be left in an inconsistent state if DFS is not completely successful in the operations it is about to perform.)

VOL_OP_CREATE	VOL_OP_WRITE
VOL_OP_TRUNCATE	VOL_OP_DELETE
VOL_OP_SETACL	VOL_OP_SETNEWVID
VOL_OP_APPENDDIR	

Note: Certaion volume operations, such as the cloning ones, explicitly set and clear the VOL_DELONSALVAGE flag.

I.2 The Fileset Handle

Refer to Section 13.3.21 on page 288 for the definition of the **struct vol_handle** structure.

The caller of this operation will record the type argument into the **vol_handle** *.opentype* field.

The *.index* field of the supplied *vol_handle* argument should be initialized as follows. Note that the caller of this operation will record the *type* argument into the **vol_handle** *.opentype* field.

The assumption is made that the root directory on a fileset will have the lowest possible (legal) inode index. This field is set to one past that root index: the index that is (or would be) used by the "first" non-root object on the fileset. Note that an LFS specific index (as in the *.vnode* component of an **afsFid**) and not a canonical one is used.

Note: There is some uncertainty as to whether DFS really relies on the setting of the *.index* field by *vol_open*( ) or whether it always does a *vol_scan*( ) or *vol_seek*( ) before using it. The behavior of Episode is described here, although in the future this document may change to leave this field undefined here.

I.2.1 Preparing for Fileset Operations

Although conflicting operations (through either the volume or vnode operations vectors) will be blocked while this fileset is held open, it might be necessary to purge certain structures or take other actions in preparation for the volume operations that are about to be performed.

Depending upon the type of open requested, either the purging of cached file and fileset state or the breaking of associations between vnodes and LFS state might be required. This topic is discussed in the topic, *Vnode to LFS association* in Section 15.11 on page 336.

Appendix J

Status Returned for **vol_getstatus**

J.1 Fileset Status Set

The following status fields are set from on-disk fileset storage.

```
statusp->vol_st.volId
statusp->vol_st.parentId
statusp->vol_st.cloneTime
statusp->vol_st.volName
statusp->vol_st.states

statusp->vol_dy.creationDate
statusp->vol_dy.updateDate
statusp->vol_dy.accessDate
statusp->vol_dy.copyDate
statusp->vol_dy.volversion
statusp->vol_dy.backupId
statusp->vol_dy.cloneId
statusp->vol_dy.llBackId
statusp->vol_dy.llFwdId
statusp->vol_dy.allocLimit
statusp->vol_dy.allocUsage
statusp->vol_dy.visQuotaLimit
statusp->vol_dy.visQuotaUsage
statusp->vol_dy.unique
statusp->vol_dy.index
statusp->vol_dy.parentIndex
statusp->vol_dy.nodeMax
statusp->vol_dy.aggrId
statusp->vol_dy.statusMsg
```

The *statusp->vol_dy.aggrId* field is set to the aggregate-ID of the aggregate holding this fileset; namely, from the aggregate structure:

```
volp->v_paggrp->a_aggrid
```

The *statusp->vol_st.states* field is treated specially. Any flags not stored on disk for the fileset (see Section 13.3.4 on page 277 and Section 13.3.5 on page 279) are instead taken from whatever value they currently hold in the *volp->v_stat.states field*. The VOL_LCLMOUNT flag in this states field is set (otherwise, cleared) if the fileset is mounted locally. Once the *statusp->vol_st.states* field has been computed, it is copied into the fileset **struct volume** structure as well (at *volp->v_stat_st.states*).

The *statusp->vol_dy.nodeMax* field is computed in an LFS-specific manner.

Note: Episode always sets *statusp->vol_dy.parentIndex* to 0.

Static status fields in *statusp->vol_st* not mentioned above should be copied from the passed in *volp->v_stat_st.vol_st* structure.

Dynamic status fields in *statusp->vol_dy* not mentioned above should be zeroed.

If the fileset in question has not been attached, only the static status from *volp->v_stat_st* is copied into the returned *vol_status*.

Appendix K

Status Set for **vol_setstatus**

K.1 Status Set

The mask argument consists of a set of VOL_STAT_xxx flags which indicate which fields from the supplied status are to be applied to the fileset (and stored on-disk).

The following VOL_STAT_xxx flags are processed by this call.

VOL_STAT_VOLID
statusp->vol_st.vol_st.volId is copied to on-disk fileset storage.

VOL_STAT_PARENTID
statusp->vol_st.vol_st.parentId is copied to on-disk fileset storage and also to the volume structure (volp->v_stat_st.vol_st.parentId).

Note: Why doesn't the DFS caller do this?

VOL_STAT_CLONETIME
statusp->vol_st.vol_st.cloneTime is copied to on-disk fileset storage and also to the volume structure (*volp->v_stat_st.vol_st.cloneTime*).

If fewer than 64 bits of time are stored on-disk, the value deposited in the volume structure MUST agree exactly with what is placed on disk.

VOL_STAT_VVCURRTIME
statusp->vol_st.vvCurrentTime is copied to on-disk fileset storage.

VOL_STAT_VVPINGCURRTIME
statusp->vol_st.vvPingCurrentTime is copied to on-disk fileset storage.

VOL_STAT_TYPE
statusp->vol_st.type is copied to on-disk fileset storage.

VOL_STAT_STATES
flags stored by LFS on-disk (see Section 13.3.4 on page 277 and Section 13.3.5 on page 279) are copied from *statusp->vol_st.states* to on-disk fileset storage.

With the exception of the following, the supplied state flags are copied into the volume structure as well (*volp->v_stat_st.vol_st.states*):

```
VOL_BUSY
VOL_DEADMEAT
VOL_GRABWAITING
VOL_LOOKUPWAITING
VOL_DELONSALVAGE
VOL_OPENDONE
```

VOL_STAT_RECLAIMDALLY
statusp->vol_stat.reclaimDally is copied to on-disk fileset storage.

VOL_STAT_VOLMOVETIMEOUT
statusp->vol_stat.volMoveTimeout is copied to on-disk fileset storage.

VOL_STAT_VOLNAME
statusp->vol_stat.volName is copied to on-disk fileset storage and also the volume structure

(*volp->v_stat_st.vol_st.volName*).

VOL_STAT_COPYDATE
statusp->vol_dy.copyDate is copied to on-disk fileset storage.

VOL_STAT_VERSION
statusp->vol_dy.volversion is copied to on-disk fileset storage.

VOL_STAT_BACKUPID
statusp->vol_dy.backupId is copied to on-disk fileset storage.

VOL_STAT_CLONEID
statusp->vol_dy.cloneId is copied to on-disk fileset storage.

VOL_STAT_LLFWDID
statusp->vol_dy.llFwdId is copied to on-disk fileset storage.

VOL_STAT_LLBACKID
statusp->vol_dy.llBackId is copied to on-disk fileset storage.

VOL_STAT_ALLOCLIMIT
statusp->vol_dy.allocLimit is copied to on-disk fileset storage.

VOL_STAT_VISLIMIT
statusp->vol_dy.visQuotaLimit is copied to on-disk fileset storage.

VOL_STAT_UNIQUE
statusp->vol_dy.unique is copied to on-disk fileset storage.

VOL_STAT_STATUSMSG
statusp->vol_dy.statusMsg is copied to on-disk fileset storage.

Appendix L

Processing for vol_create

L.1 Processing Accomplished

The canonical anode number supplied in the position argument specifies the location (anode index) at which the file is to be created. The anode generation number to be used is taken from the *xvattrp->xvattr.fileID.low* field.

Note: If the root directory of a fileset is being created, its anode generation number should be set to 1.

The initial file state is taken from the following fields in the initial **struct vattr** at *xvattr->vattr*.

```
.va_mode , .va_type        file type
.va_oid  , .va_gid         file owner
.va_rdev                   for device files
```

The link count is set to 1. The atime, mtime and ctime times are all set to the current time.

The file's **uuid** should be set to zeroes -- ignoring the value supplied via the *xvattr* argument.

File status fields not mentioned above should be set to 0 (or whatever other default value is appropriate for the LFS).

The newly created file has no associated **ACL**s or property lists. In particular, no initial or inheritance **ACL** is applied.

Ordinary files, directories, symlinks, FIFOs and device files can all be created via this call.

If a directory is created (the supplied *.va_type* specifies VDIR), it should be initialized to an empty state WITHOUT "." and ".." entries (these will be supplied in a subsequent *vol_appenddir*( ) operation).

This operation fills in several fields in the *handlep* **vol_handle** as follows.

handlep->fid The *.Vnode* and *.Unique* fields are set to the appropriate value for the newly created file. See the discussion under *vol_scan*( ).

handlep->type Set to a combination of the initial mode (from *.va_mode*) and type (S_IFMT encoding based on the specified *.va_type*) bits for the newly created file.

handlep->index Set to the same value as the *fid.Vnode* field (LFS-specific anode index).

Chapter 16

VFS (Vnode) Interface and Operations

16.1 Overview

As DFS was being designed, there were several goals that related to the VFS interface.

1. That DFS itself be as portable as possible.
2. That DFS "plug" into an OS kernel with minimal (none, if possible) changes to that kernel.
3. If files are being accessed both through DFS as well as a local path (including,say, an NFS exporter), that the local accesses perform the proper DFS token synchronization.

To achieve the first two goals, DFS is itself standardized on a particular definition of the VFS interface which is very similar to SUN's initial design. The **vnode** operations vector for this interface is the **struct xvfs_xops** defined in Section 13.6.4 on page 296 and is often referred to as the extended vnode interface. All components of DFS (the Cache Manager (CM), Protocol Exporter (PX) and LFS) are written to this set of vnode operations.

The mapping between the format of the native (expected by the OS) vnode operations and this extended set of operations is handled by a set of glue code. Appendix C on page 245 describes typical components of a VFS+ package. In particular, some glue code examples are presented in Section C.3 on page 246. This glue code comes in several types which are invoked in different situations.

- One type of glue code is invoked whenever the native OS performs a vnode operation on the filesystem (that is, NOT a DFS access). Its job is to obtain DFS tokens for the operation in question, call into the file system to perform the operation and then release the DFS tokens before returning. This code is referred to as "token glue".
- A second type of glue converts from the native OS vnode interface (what its caller expects) to the extended vnode interface (what the callee expects).
- A third type of glue converts from the extended vnode interface (what its caller expects) to the native OS vnode interface (what the callee expects). This code is referred to as "vfs glue".

How this glue is invoked will become clear in the following sections.

16.1.1 Enhanced Vnode Operations Vector

For files exported to DFS, vnodes are assumed to have a **vnode** ops vector that points at a **struct xvfs_vnodeops** vector which contains three sets of **vnode** operation pointers (see Section 13.6 on page 293 for definition).

In general terms (the specifics should become clearer below), the three sets of vnode operation vectors contained within this **struct xvfs_xops** (see Section 13.6.4 on page 296) are the following.

O-ops Native OS format
Invoked by the VOPO_XXX or VOP_XXX macros

The glue routines pointed at by these pointers are invoked by the native OS as part of ordinary file system activities. For the most part, they behave as shown in the following pseudo-code:

```
xglue_xxx(  )
if ((the fileset is exported to DFS) &&
(the fileset is read/write))
obtain the needed DFS tokens

/* Perform the operation via indirecting
* through the .nops vector */
VOPN_xxx(  )

if (we obtained tokens above)
return the tokens
```

For a typical system, the source for these routines can be found in a location such as **src/file/xvnode/<system>/xvfs_osglue.c** .

X-ops Extended format

Invoked by the VOPX_XXX macros.

On a file server the DFS Protocol Exporter, which doesn't need to obtain DFS tokens, calls through this vector to perform operations on exported files.

If the physical file system is an LFS that directly supports the extended vnode operations (as Episode does), the entries in this vector point directly at LFS operations.

If the physical file system does not support the extended vnode operations (a native UFS), this vector points at a set of glue routines which attempt to implement the extended operations in terms of the native (or "original") ones pointed at by the **N-ops** vector. For a typical system, the source for this glue (the *xufs_xxx*( ) routines) can be found in **src/file/xvnode/<system>/xvfs_vfs<xxx>.c** .

N-ops Native OS format

Invoked by the VOPN_XXX macros.

Operations pointed at by this vector are only ever directly invoked by "glue" code in either the **O-ops** or **X-ops** routines mentioned above.

If the physical file system is an LFS that directly supports the extended **vnode** operations (as Episode does), this vector points at a set of glue routines that convert from native OS vnode operations to extended ones. For a typical system, the source for this glue (call them, say, *nux_xxx*( ) routines) can be found in src/file/xvnode/HPUX/xvfs_os2vfs.c .

If the physical file system does not support the extended vnode operations (as say, in a native UFS), this is the original (vendor supplied) vector that points into the filesystem code.

The manner in which these three vectors, along with the different types of glue code, are used is most easily demonstrated by a few examples.

Example 16-1 Protocol Exporter Access to a Fully Functional LFS

The Protocol Exporter already has any tokens that are required. Therefore, it simply calls the through the **X-ops** vector:

```
/* PX routine responding to a remote DFS request */
VOPX_xxx()
/* which lands in the LFS directly */
lfs_xxx()

=>Perform the operation<=
```

Example 16-2 Protocol Exporter Access to a non-LFS Filesystem

This would be a file system such as UFS. The **X-ops** vector, which the PS calls through as in the case above, must point at glue which "reformats" the operation into one that the native filesystem can perform:

```
/* PX routine responding to a remote DFS request */
VOPX_xxx()
/* which lands in the xufs_xxx glue */
xufs_xxx()
/* Convert to a native vnode operation and invoke */
VOPN_xxx()
/* which lands in the native f.s. */
ufs_xxx()
=>Perform the operation<=
```

Example 16-3 Local OS to a Fully Functional LFS

(Not through the DFS protocol exporter.) The native OS, knowing nothing about DFS, calls through the **O-ops** vector which lands in the token-obtaining glue:

```
/* native OS */
VOP_xxx()      /* equivalently: VOPO_xxx() */
    /* which lands in the xglue_xxx() glue */
    xglue_xxx()
        Obtain tokens, if necessary
        /* Invoke operation */
        VOPN_xxx()
            /* which lands in the nux_ glue */
            nux_xxx()
                /* Invoke extended vnode operation */
                VOPX_xxx()
                    /* which lands in the LFS */
                    lfs_xxx()
                      =>Perform the operation<=
```

Example 16-4 Local OS Access to a Non-LFS Filesystem Such as UFS.

(Not through the DFS protocol exporter.) The native OS, knowing nothing about DFS, calls through the **O-ops** vector which lands in the token-obtaining glue:

```
/* native OS */
VOP_xxx()        /* equivalently: VOPO_xxx() */

    /* which lands in the xglue_xxx() glue */
    xglue_xxx()
        /* Obtain tokens, if necessary */
        /* Invoke operation */
        VOPN_xxx()
            /* which lands in the native f.s. */
            ufs_xxx()
              =>Perform the operation<=
```

16.1.2 Converted Vnodes

Vnodes which contain an enhanced (3-part, see above) operations vector are said to be converted. They are identified by the fact that the V_CONVERTED bit is set in their flags word. The easiest way to obtain such an enhanced vnode vector is to make a call to the DFS procedure *xvfs_InitFromXOps*( ) at initialization time. Once a vector has been constructed, individual vnodes can be pointed at it and have their V_CONVERTED flag set via one of the mechanisms described below:

```
xvfs_InitFromXOps(&lfs_xops, &enh_ops_vector);
```

The first argument is a **struct xvfs_xops** vector of extended vnode operations provided by the LFS.

The second argument is a **struct xvnodeops** that is filled in by the call as listed below:

O-ops pointed at *xglue_xxx*( ) glue routines

X-ops pointed at supplied LFS operations (from the first argument)

N-ops pointed at *nux_xxx*( ) glue routines

As mentioned above, files being accessed via DFS must have their **vnode**s converted to guarantee that the proper token synchronization occurs for all LFS accesses.

Vnodes for the LFS can always be converted as described above. Although the *xglue_xxx*( ) and *nux_xxx*( ) glue are ALWAYS invoked for local accesses, even if DFS is not in the picture, they "pass through" relatively quickly and don't have a significant impact on performance.

Note: This is the approach taken by Episode.

16.1.3 Enhanced Vfs Vector

In a similar manner, DFS assumes that the vfs operations vector has been converted into an enhanced **struct xvfs_vfsops** as follows.

```
struct xvfs_vfsops {
    struct vfsops       xvfsops;
    struct vfsops       vfsops;
    int (*vfsgetvolume)();
};
```

The components of this structure have the following significance:

.xvfsops With the few exceptions listed below, the pointers in this vector are the same as those found in the *.vfsops* vector for the LFS.

The *_vget*, *_root* and *_unmount* pointers are replaced by pointers to glue code: *xglue_vget*( ), *xglue_root*( ) and *xglue_unmount*( ).

The *xglue_root*( ) and *xglue_vget*( ) interludes both invoke the LFS operation via the VFSX_xxx macro and, if necessary (the fileset is exported and writeable), converts the returned vnode as discussed above. Note that this behavior is only relevant for exported native UFS filesets; for a fully functional LFS, the assumption is that the *vnodes* it returns are already converted.

The *xglue_unmount*( ) interlude flushes any held tokens for the fileset in question if necessary (the fileset is exported and writeable). Unlike the *xglue_root*( ) and *xglue_vget*( ) functions, this does serve a purpose for fully functional LFS filesystems.

This vector is used by the native OS and is accessed via the standard VFS_xxx macros.

.vfsops The "original" vfs operations vector supplied by the LFS. This vector is only used by DFS and is accessed via the VFSX_xxx macros.

.vfsgetvolume

This pointer is set to point at an LFS routine which is capable of converting between a vfs structure and a held DFS volume pointer. See the detailed description of *vfs_getvolume*( ) in Section 16.6.1 on page 424.

This procedure is used by the *xglue_unmount*( ), *xglue_root*( ) and *xglue_vget*( ) glue routines mentioned above. (They need a fileset structure in order to determine if the fileset is exported and writeable.)

For native UFS filesystems, this pointer is set to a DFS routine which is able to "simulate" the needed functionality.

The easiest way to construct a "converted" vfs ops vector is by means of the DFS procedure *xvfs_InitFromVFSOps*( ) as follows:

```
xvfs_InitFromVFSOps(&lfs_vfsops, &enh_vfsops, lfs_getvolfunct)
```

The first argument is the address of the standard vfs operations vector provided by the LFS.

The second argument is the address of an enhanced **xvfs_vfsops** vector which is, as described above, built by the call.

The third argument is the *vfs_getvolume*( ) function provided by the LFS and described in Section 16.6.1 on page 424.

There are several options for how the vfs operations vector for a fileset is "converted" as described above. A few are listed below.

1. At initialization time, if it is known that DFS is present, the vfs operations vector can be constructed as shown above and inserted into the **vfssw[]** table so that it will work its way into all vfs structures.

   If it is unknown at initialization whether DFS will be installed (dynamically) later, this **vfssw[]** can be performed at a later time when DFS "announces" itself to the LFS for the first time. One possibility would be to do this when an aggregate or fileset was first attached.

   To avoid having to write into the **vfssw[]** table, it can initially point at a full sized xvfs_vfsops structure within which only a standard vfsops vector at the front is present. If DFS is present, the full **xvfs_vfsops** structure can be converted "in place" by passing that structure to the first two arguments of *xvfs_InitFromVFSOps*( ).

   Note: *xvfs_InitFromVFSOps*( ) is carefully coded to allow this in-place conversion to occur.

   Once again, this conversion can be done either at initialization or a later time. (Assuming, as appears to be the case, that the in-place conversion of the vfs ops vector is safe with regard to other processes and processors.

   Note: Episode behaves this way, performing the in-place conversion at initialization time.

2. Schemes that involve changing, at fileset attach and detach time, the *.vfs_ops* pointer in a vfs structure are also possible. The question of what to do if the fileset isn't mounted (and hence, there is no vfs structure) at attach time must be addressed, however.

   Note: Although DFS currently "mounts" LFS aggregates that are attached in order to make *vol_sync*( ) work properly, this doesn't occur until after they are attached.

16.2 Administrative Rights

Several of the operations listed here require that the caller have administrative rights to the file being operated upon. Under standard UNIX, these rights would be granted for ROOT as well as the owner of the file.

Within the LFS, administrative rights to a file are granted under the following situations (see *Access Control List Overview*, Chapter 8 on page 155).

1. If the caller is the local ROOT.
2. If the caller is in the special DFS administrative group.
3. If the caller has **perm_control** rights to the file in question. If the file is not protected by an **ACL**, this right will be granted to the file's owner.

Vnode operations, both standard and extended, which have traditionally required the caller to be the owner of a file (or ROOT) should be changed to work as shown above instead.

16.3 Copy-on-Write Impacts

As discussed in the volume operations specification, it is expected that some sort of copy-on-write technology is used in order to make clone operations space and time efficient. Depending on the particular implementation, a number of the **vnode** operations listed below may unexpectedly fail with [ENOSPC] or errors if the copy-on-write sharing cannot be broken. Specifically:

a. the fileset is currently cloned and the aggregate in question is full (ENOSPC) or

b. the fileset quota limits prevent the fileset in question (which one depends on the implementation) from physically growing in size (EDQUOT)

As one might expect, any operation that creates or grows something on disk (file, directory, ACL, and so on) can fail. Additional operations that MIGHT fail as well are listed below.

- Over-writing an existing piece of a file.
- Any operations that modify the contents of a directory. For example: *unlink*( ), *rename*( ), *rmdir*( ), and so forth.
- Truncating a file to a smaller size. The copy-on-write algorithm should be designed so that the following operations never fail (with [ENOSPC] or [EDQUOT] errors, at least).
- The *vn_setattr*( ) operation. This implies that either anodes are copied or space for them is reserved during a clone operation. Although it is not a DFS requirement, it is suggested that the following operations likewise never fail (with (with [ENOSPC] or [EDQUOT] errors).
- Truncating a file to 0 size. Some implementations may only be able to do this if the file has not been modified since the time of the clone.

16.4 Swap Files

Some operating systems allow swapping to ordinary files in the file system. Since there's no benefit in cloning such files, it is suggested that an LFS avoid performance and unexpected error (see ENOSPC and EDQUOT discussion earlier) problems by "ignoring" such files at clone time. Although there are any number of possible approaches here (so long as the OS and LFS agree), the following is one possibility:

- At clone (or reclone) time, any swap files that are encountered are treated as zero-length files. Although the file IS cloned in order to maintain consistent link counts, the contents of the swap file are ignored and not subject to any copy-on-write behavior.
- Any attempt to enable swapping on a currently cloned file (perhaps: with greater than zero length) fails.
- Obviously, any attempt to enable swapping to a file on a read-only clone should fail (presumably, with [EROFS]).

16.5 Synchronization Between Vnode and Fileset Operations

For the most part, DFS protects against undesired interactions between volume (vol_xxx) administrative operations and ordinary (local or exporter) **vnode** operations.

a. Both the vnode glue layer and the DFS protocol exporter ensure that vnode operations are properly synchronized with respect to any fileset operations in progress. If the fileset in question has been opened with a "cuncurrency mode" that conflicts with a vnode operation, that **vnode** operation blocks until fileset is available again. See (c) below.

b. While an administrative application has a fileset open (*vol_open*( )), any other open requests that arrive for that fileset are refused.

c. When a fileset is opened, the caller indicates the "type" of operations that will be performed on it. The open blocks until any "conflicting" **vnode** operations currently in progress finish.

   For the duration of the open, step (a) protects against additional **vnode** operations and step (b) protects against additional fileset (volume) operations. Consult the *vol_open*( ) and *vol_concurr*( ) operations.

d. When a fileset is closed, blocked vnode operations are allowed to proceed. There are a number of vnode operations for which the glue does not perform any synchronization, however. The two reasons for these are: (1) the calls are not allowed to block just because a fileset operation is in progress, or (2) there's no harm in allowing the call to proceed concurrently with a volume operation.

   In these cases, the LFS appears to be left "on its own" to protect against undesirable (if any are possible) interactions. A typical set of these glue functions is listed below:

   - *vn_inactive*( )

     This call cannot block waiting for a series of volume operations to complete. The LFS is on its own to ensure that the correct thing eventually happens (like, ZLC file deletion) if the fileset in question is open (via *vol_open*( )).

   - *vn_bmap*( )
   - *vn_strategy*( )
   - *vn_bread*( )
   - *vn_brelse*( )
   - *vn_select*( )
   - *vn_ioctl*( )
   - *vn_pathconf*( )
   - *vn_fpathconf*( )
   - *vn_fid*( )

     It appears this operation left unprotected for performance reasons since it doesn't do anything that could get it into trouble with respect to concurrent volume operations.

   - *vn_lockctl*( )

     Due to the blocking nature of this call, the proper synchronization (which itself can block out other users) is not performed. Depending upon the implementation, no synchronization may be necessary.

16.5.1 Directory Offsets

The *vn_readdir*( ) operation returns (when called by the DFS file exporter, at least) directory entries that contain an *.offset* field. DFS assumes that these offsets have the following properties.

- The offset of the first entry in a directory is 0.
- Each directory entry is assumed to have an offset that never changes for that entry. Specifically: the offset for an entry never changes as the result of the addition or removal of other entries.
- These offsets increase for directory entries as they are returned by *vn_readdir*( ).
- Within a returned directory entry, the *.offset* field is set to the offset of the NEXT entry to be returned or, if empty space follows that entry, the offset that would be associated with any entry to be placed within that space in the future.
- As discussed under *vn_readdir*( ), these offsets are consistent with the *.uio_offset* value input to and returned from the call.

Some file systems provide a mechanism for performing online reorganization or compaction of directories. If such an operation would cause directory entry offsets to change, the following must hold.

- There must be a way of dynamically disabling this functionality.
- Such reorganization must NEVER be attempted on a readonly fileset (either a *.readonly* replica or a *.backup* clone). Since DFS can switch between replicas at ANY time, these replicas must appear identical, even down to these directory offsets.
- Such reorganization must advance the directory's *1.dataVersion* attribute to allow DFS clients to recognize that any directory offsets being cached might be invalid.

Many file systems, including the standard UNIX UFS, will automatically de-fragment (compact) a directory block during an insertion if there is insufficient space to add a desired entry. Although there may be no way to inhibit this behavior, the file system should at least advance the directory's *.dataVersion* attribute as mentioned above.

16.6 Vfs Operations

Since DFS doesn't generally use the vfs operations, they have not been extended or standardized the way that the vnode ones have. The DFS requirements that do exist on these operations are listed below:

vfs_vget( ) Obtain a vnode given an NFS-style fid that was originally handed out by *vn_fid*( ).

As discussed under *vn_fid*( ) (see Section 16.7 on page 459), the anode index and generations within the **fid** are represented in network (as in *hton32*( )) order.

vfs_statfs( ) Obtain statistics for a file system. See Section 15.14 on page 340. More specifically, see Section 13.3.5 on page 279 to see the dynamic status portion of the fileset's status.

The returned *.f_blocks* and *.f_free* fields should be computed from the fileset visible quota fields (*.visQuotaLimit* and

Since filesets do not possess fixed size anode tables, something "appropriate" should be set for the returned .f_files and .f_ffree fields.

Note: *Episode returns –1 for these.*

16.6.1 *vfs_getvolume*( )

The *vfs_getvolume*( ) function below is not part of the standard vfs operations vector. Instead, it is pointed at by the *vfsgetvolume* pointer in the extended **xvfs_vfsops** structure established when a fileset is exported. See *Define the Enhanced Operations Vector*, in Section 13.6.8 on page 299 for information about its fields. There, expect to see the pointer just mentioned.

The *vfs_getvolume*( ) function has the following signature:

```
vfs_getvolume(
    /* IN */  struct osi_vfs   *vfsp,
    /* OUT */ struct volume    **volpp
);
```

It's description is as follows:

DESCRIPTION Obtain a volume structure from a vfs pointer.

ARGUMENTS

vfsp A pointer to a vfs structure.

volpp A pointer to a held volume structure is returned here.

DISCUSSION This call should obtain a fileset ID from the supplied vfs structure and then call *volreg_Lookup*( ) with that in order to obtain a held fileset structure to return.

See Section 15.15 on page 399 for a description of *volreg_Lookup*( )

RETURN VALUES This call should return whatever value *volreg_Lookup*( ) returned to it. If all is successful, a 0 will be returned. If the fileset does not reside locally, [ENODEV] will be returned instead.

16.7 Base Vnode Interface

This chapter describes the **X**-*ops*. As discussed in Section 11.4 on page 238, this set of functions is one of two extensions to the array of vnode operations (or equivalent array of operations in some kernels). The **X**-*ops* do not perform their own synchronization with clients, but assume that it has been done by the caller. Also, the **X**-*ops* have an interface that is nearly identical across platforms, making them suitable for use by the Protocol Exporter and other components of the DFS File Server. Most of the **X**-*op* function specifications are derived from the original Sun Microsystems **vnode** architecture, as described in a paper by Kleiman appearing in the proceedings of the 1986 Summer Usenix Conference[11]. Those specifications are given in this chapter. The next section, *Extended Vnode Interface*, Section 16.8 on page 460, gives specifications for some additional **X**-*ops*, added to meet the needs of DFS.

A description of the operations available within the vnode function array appears below. The array is an exported data type. (The associated declaration is found in Section 13.6.4 on page

11. S. R. Kleinman. *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, Proceedings Summer Usenix Technical Conference & Exhipition, pp. 238-247, Atlanta, Georgia, June 1986.

296). It is reproduced here for convenience.

```
    /* vnode operations */
struct xvfs_xops {
    int      (*vn_open)();
    int      (*vn_close)();
    int      (*vn_rdwr)();
    int      (*vn_ioctl)();
    int      (*vn_select)();
    int      (*vn_getattr)();
    int      (*vn_setattr)();
    int      (*vn_access)();
    int      (*vn_lookup)();
    int      (*vn_create)();
    int      (*vn_remove)();
    int      (*vn_link)();
    int      (*vn_rename)();
    int      (*vn_mkdir)();
    int      (*vn_rmdir)();
    int      (*vn_readdir)();
    int      (*vn_symlink)();
    int      (*vn_readlink)();
    int      (*vn_fsync)();
    int      (*vn_inactive)();
    int      (*vn_bmap)();
    int      (*vn_strategy)();
    int      (*vn_ustrategy)();  /* assuming stuff already mapped in */
    int      (*vn_bread)();
    int      (*vn_brelse)();
    int      (*vn_lockctl)();
    int (*vn_fid)();             /* op for old style fid op */
    int (*vn_hold)();            /* maybe don't need these; revisit */
    int (*vn_rele)();
    /*
     * new ones for us to provide, rather than just existing to
     * make writing the O functions easier (i.e. porting).
     */
    int (*vn_setacl)();
    int (*vn_getacl)();
    int      (*vn_afsfid)();
    int (*vn_getvolume)();
    int (*vn_getlength)();
    /*
     * Some new ops for AIX 3
     */
    int (*vn_map)();             /* also used for SunOS 5 */
    int (*vn_unmap)();
    /*
     * A new op for OSF/1
     */
    int (*vn_reclaim)();
    /*
     * Some new ops for SunOS 5
```

```
     */
    int (*vn_read)();
    int (*vn_write)();
    int (*vn_realvp)();
    void (*vn_rwlock)();
    void (*vn_rwunlock)();
    int (*vn_seek)();
    int (*vn_space)();
    int (*vn_getpage)();
    int (*vn_putpage)();
    int (*vn_addmap)();
    int (*vn_delmap)();
    int (*vn_pageio)();
#define   vn_frlock  vn_lockctl         /* overlay equivalent ops */
    /*
     * Ops for HP/UX
     */
    int (*vn_pagein)();
    int (*vn_pageout)();
};
```

The following descriptions do not list all possible error cases and return values. The primary intention has been to identify DFS specific behavior that differs from what would be expected in an *ordinary* **vnode**-based file system implementation. Some operations have a more comprehensive set of error returns defined than others.

For *vn_xxx*( ) operations that are not expected to be called (and, hence, not implemented), it is recommended that the operations vector point to a function which returns [ENOSYS].

NAME

vn_open — Open a file associated with a vnode

SYNOPSIS

```
int vn_open(
    /* INOUT */ struct vnode **avpp,
    /* IN */    long         aflags,
    /* IN */    struct ucred *acred
    );
```

ARGUMENTS

avpp	Pointer to the vnode to open.
aflags	Open flags.
acred	Pointer to caller's credential structure.

DESCRIPTION

Perform an open protocol on the given *avpp* vnode pointer. The open flags are provide by *aflags*, and the caller's credentials are passed in *acred*. A new vnode may be created, and is passed back in *avpp*.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

None.

NAME

vn_close — Close a file associated with a vnode.

SYNOPSIS

```
int vn_close(
    /* IN */   struct vnode *avp,
    /* IN */   long         aflags,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode to close.
aflags	Flags used with the open.
acred	Pointer to caller's credential structure.

DESCRIPTION

Perform a close protocol on the given *avp* vnode pointer. The flags passed to the original *vn_open*( ) are passed in *aflags*, and the caller's credentials are passed in *acred*.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

None.

NAME

vn_rdwr — Read or Write data from or to a vnode

SYNOPSIS

```
int vn_rdwr(
    /* IN */   struct vnode *avp,
    /* IN */   struct uio   *auio,
    /* IN */   enum uio_rw  arw,
    /* IN */   int          aio,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode.
auio	Pointer to the user structure which supplies the I/O arguments, including the address in user space where the data to be read (written) are.
arw	Specifies the direction of the I/O. For a read, specify (UIO_READ); for write, (UIO_WRITE).
aio	The associated I/O flags such as IO_APPEND or (and) IO_SYNC.
acred	Pointer to caller's credential structure.

DESCRIPTION

Read or write data from/to the vnode pointed to by *avp*. The user structure which supplies the I/O arguments is pointed to by *auio*, and includes the address in user space where the data is to be read from or written to. The I/O flags contained in *aio* may specify that, if the direction of I/O is output, the output is to be done synchronously. Other flags may be present but will generally be ignored for regular disk files (they may be used by device special files, FIFO's, and the like). If input is done, the file's atime will be modified. If output is done, the file's mtime and ctime will be modified. Under some obscure circumstances, the file's mode may be modified, depending on the caller's credentials.

DISCUSSION

The specified file is either read or written, based on the *arw* argument. The actual transfer is controlled by paramaters supplied in the **uio** structure and *aio* word.

Note: A symlink (its link text) can be accessed via this call.

For a write (UIO_WRITE):

The usual tests for maximum file size (ulimit) are performed.

IF IO_SYNC is present, a synchronous write is performed. Unless this call is being issued by ROOT, the file's set-uid and set-group (VSUID, VSGID) flags are cleared.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	Returned for a FIFO, BLK or CHR special device.
[EROFS]	Returned if the fileset is read-only.

NAME

vn_ioctl — Handle I/O control on an open file descriptor

SYNOPSIS

```
int vn_ioctl(
    /* IN */   struct vnode *avp,
    /* IN */   long         acmd,
    /* IN */   char         *adata,
    /* IN */   long         aflags,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to vnode to affect.
acmd	Command to perform.
adata	Pointer to the data involved.
aflags	The associated open flags.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Perform an *ioctl*( ) on the vnode pointed to by *avp*. The command to perform is found in *acmd*, the data involved in the command is found in *adata*, the open flags associated with the file are in *aflags*, and *acred* points to the caller's credential structure.

This function is more or less irrelevant for files. Both the Cache Manager and Episode functions return [EINVAL] without doing anything.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

None.

NAME

vn_select — Perform a select on a vnode

SYNOPSIS

```
int vn_select(
    /* IN */   struct vnode *avp,
    /* IN */   long         awhich,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode on which to do the select.
awhich	Specifies the I/O direction.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Perform a select on the vnode pointed to by *avp*. The direction of the I/O is specified in *awhich*, and the caller's credential structure is pointed to by *acred*.

This function is more or less irrelevant for files. Both the Cache Manager and Episode functions do nothing. On most platforms they return [EINVAL], but on some platforms they return 0.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

None.

NAME

vn_getattr — Get the attributes for a vnode

SYNOPSIS

```
int vn_getattr(
    /* IN */   struct vnode *avp,
    /* IN */   struct vattr *aattrs,
    /* IN */   int          aflag,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp Pointer to the vnode whose attributes are desired.

aattrs Pointer to the buffer in which to place the given vnode's attributes. If *aflag* is zero, this is a **vattr**, structure, else a **xfvs_vattr** structure.

aflag Flag indicating whether *aattrs* is an extended attribute structure.

acred Pointer to the caller's credential structure.

DESCRIPTION

Get the attributes associated with the vnode pointed to by *avp*. The caller provides *aattrs*, a pointer to a **struct vattr** in which to place the attributes. Also passed is a pointer to the caller's credential structure, *acred*. Vnode attributes include much of the information reported by the *stat* system call, such as the owner's **uid** and **gid**, the file size, and access and modify times.

DISCUSSION

The standard **vnode** attributes at *aattrs->vattr* are returned in an obvious manner. Fields that might require special attention are listed below.

.va_fsid The value returned here should be chosen with care since it needs to be unique (per fileset) and a given aggregate (device) can hold several filesets. Although an individual LFS is free (seemingly) to use any appropriate algorithm to construct this field, the following is done by Episode:

```
.va_fsid = (volumeID.low   << 16) |
           (device major # << 8)  |
           (device minor #)
```

.va_rdev The device number of the aggregate on which this fileset resides is returned.

.va_blocks The blocks used value is based on the logical amount of disk space used by the file (as if there were no copy-on-write (clone) sharing taking place). This corresponds to the visible (versus actual allocated) quota on a fileset.

If the flag argument is non-zero, the extended attributes at *aattr->xvattr* are returned as well. Definitions of the fields in this structure and the values to be returned for them can be found earlier in this document. Fields requiring special attention are listed below.

.anonAccess If the object in question does not have an **ACL**, 0 is returned in this field. Likewise, if the *any_other* entry in the **ACL** is invalid, a 0 is returned. Otherwise, it is set to the logical intersection of EVERY valid entry in its **ACL**.

The *acred* argument is used in the computation of the *.callerAccess field*.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[non-zero] This function returns an error.

NAME

vn_setattr — Set the attributes for a vnode

SYNOPSIS

```
int vn_setattr(
    /* IN */   struct vnode *avp,
    /* IN */   struct vattr *aattrs,
    /* IN */   int          aflag,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode whose attributes are to be set.
aattrs	Pointer to the buffer from which to get the given vnode's attributes.
aflag	Flag indicating whether *aattrs* is an extended attribute structure.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Set the attributes of the vnode pointed to by *avp*. The caller provides *aattrs*, a pointer to a **struct vattr** from which the attributes are obtained. In this structure, a value of –1 in any field indicates that the corresponding attribute is not to be changed. Also passed is a pointer to the caller's credential structure, *acred*. This function is called on behalf of various system calls, such as **chmod**, **chown**, **utimes**, and **ftruncate**.

DISCUSSION

The standard vnode attributes at *aattrs–>vattr* are applied to the designated file as follows.

.*va_ctime*	If the .*tv_sec* sub–field is not equal to –1. This field is ONLY settable if the flags argument is non–zero.
.*va_nlink*	If not equal to –1. If the link count is being set to zero, the file will be deleted by *vn_inactive*( ) when its vnode reference count goes to zero. As described earlier in *Zero Link Count Files*, Section 15.6 on page 332, such files are preserved across reboots and FSCKs until *vn_inactive*( ) explicitly deletes them.
.*va_mode*	If not equal to –1.
.*va_uid*	If not equal to –1.
.*va_gid*	If not equal to –1.
.*va_size*	If not equal to –1. The file's modified time (mtime) is set to the current time UNLESS it is being explicitly set (.*va_mtime*). If a directory is being operated on, an [EISDIR] error is returned.
.*va_mtime*	If the .*tv_sec* sub–field is 0 and the .*tv_usec* sub–field is –1, the file's atime and mtime are both set to the current time. (System V style *utime*( ) call.) Alternatively, if the .*tv_sec* field is not equal to –1, the file's mtime is set.
.*va_atime*	If the .*tv_sec* sub–field is not equal to –1 and the above .*va_mtime* case does not supercede this one, the file's atime is set.

If the flag argument is non-zero, the extended attributes at *aattrs–>xvattr* are additionally set as follows.

.volVersion	If the *.high* or.*low* sub–field is not equal to –1.
.dataVersion	If the *.high* or.*low* sub–field is not equal to –1.
.fileID	The anode generation value, from the *.low* sub–field, can be changed if it is not equal to –1. The *.high* field, the anode index, is ignored.
.clientOnlyAttrs	If not equal to –1.

Unless the file's ctime is explicitly set, it is set to the current time by this call.

If setting *.va_mode*, the set-uid (VSVTX) flag is cleared if the caller is not ROOT and the set-gid (VSGID) flag is cleared if the caller is not a member of the file's group. Other actions (regarding the sticky (VTEXT) flag) might be required as well, depending on the OS.

Administrative rights to the object are required for the caller if any of the following fields are being set: *.va_mode*, *.va_uid*, *.va_gid*, *.va_atime*, *.va_mtime*. If *.va_mtime* and *.va_atime* are being set to the current time, either administrative rights or write access to the file are sufficient.

Notes:

1. If flags is non-zero, an extended set-attribute attribute is being performed and DFS (at the client side) has performed its own permissions check.
2. Episode returne EROFS if the *.va_uid* or *.va_gid* is being changed and the file resides on a readonly fileset

Any permissions checks performed by this operation must be performed against the original *mode*, *uid* and *gid* fields (as opposed to the new ones set by this call).

DFS expects to be able to change or set the size of a file even if the caller does not currently own or have write access to the file.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EISDIR]	An attempt was made to change the length of a directory with this call.
[EPERM]	The caller possesses insufficient rights to perform this operation.
[EROFS]	The fileset is read-only.

NAME

vn_access — Check access permissions for a vnode

SYNOPSIS

```
int vn_access(
    /* IN */   struct vnode *avp,
    /* IN */   long         amode,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode to examine.
amode	Mode to check for access. Values are S_IREAD, S_IWRITE, S_IEXEC.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Check the access permission for the vnode pointed to by *avp*. The access mode to be checked (read, write, execute) is specified by *amode*, and a pointer to the caller's credential structure is provided by *acred*. If the specified access is denied, *vn_access*( ) returns a non-zero value.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success. The caller has the rights specified in the mode argument for the object in question.

ERRORS

[EROFS]	The fileset is read-only and the caller is asking for writh access. This is either a *.backup* clone or a *.readonly* replica.
[EACCES]	The caller does not have the specified rights to the object.

NAME

vn_lookup — Look up a component name in a directory vnode

SYNOPSIS

```
int vn_lookup(
    /* IN */   struct vnode *adp,
    /* IN */   char         *aname,
    /* OUT */  struct vnode **avpp,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

adp	Pointer to the directory vnode to be searched.
aname	Character string name being looked for.
avpp	A place to store the address of the vnode corresponding to the name being looked up (if one exists).
acred	Pointer to the caller's credential structure.

DESCRIPTION

Look up a pathname component name *aname* in the directory represented by the *adp* vnode pointer. The vnode for the corresponding object (if one exists in the given directory) is stored in *avpp*. The caller's credentials are passed in through the *acred* pointer.

DISCUSSION

If this call succeeds, it returns a pointer to a held (as in *vn_hold*( ), for the object that was found.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok]	This function returns success. The caller has the rights specified in the mode argument for the object in question.

ERRORS

[EACCES]	The caller does not have the search rights to the specified directory.
[ENOENT]	The name is not found or the directory has been deleted.

NAME

vn_create — Create a new file in the given directory

SYNOPSIS

```
int vn_create(
    /* IN */   struct vnode *adp,
    /* IN */   char         *aname,
    /* IN */   struct vattr *aattrs,
    /* IN */   enum  vcexcl aexcl,
    /* IN */   int          amode,
    /* OUT */  struct vnode **avpp,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

adp	Pointer to the directory vnode in which the new file is to be created.
aname	Character string name for the file to create.
aattrs	Pointer to the attribute structure to give to the new file.
aexcl	Exclusive or non-exclusive creation flag. If exclusive, the specified name (*aname*) must not already reside within the directory.
amode	Access rights required if file exists in a non-exclusive creation (S_IREAD, S_IWRITE, S_IEXEC).
avpp	Set to the address of the vnode corresponding to the newly-created file, if successful.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Create a file named *aname* in the directory associated with the *adp* vnode. The new file's information is passed in via the following parameters: *aattrs* contains the initial attribute structure, and *aexcl* specifies whether the create is exclusive or non-exclusive. *amode* is not the mode of the new file (that is in the appropriate field in *aattrs*), but specifies access rights that the existing file must allow if there is an existing file and the create is non-exclusive. It is derived from the flags argument to the *open* system call. A pointer to the caller's credentials are provided via *acred*. If *vn_create*( ) is successful, *avpp* is set to the address of the vnode pointer describing the new file.

DISCUSSION

- **If the desired object already exists**:

  If the exclusive argument is non-zero, the call fails with [EEXIST].

  If the existing object is a directory and the mode argument indicates that the caller wants write access (S_ISWRITE), the call fails with [EISDIR].

  If the caller does not have the rights given by the mode argument to the file, the call fails with [EACCES].

  If the existing object is an ordinary file and the supplied *aattrs->va_size* field is 0, it is truncated to zero length.

- **If the desired object does not already exist**:

The new file's user field (*va_uid*) is taken from the DFS authentication structure which itself is obtained from the supplied credentials. Depending on whether SYS5 or BSD semantics are being adhered to, the group field (*va_gid*) is either taken from this PAC or from the parent directory. Consult *Obtaining the Identity of the Principal*, Section 11.8.10 on page 243. If the PAC indicates that this is an unauthenticated RPC operation, the *user* and *group* fields are set instead to –2 and –1, respectively.

Note: If BSD behavior is in effect (va_gid from parent directory), then it seems equally valid to form the va_gid this way even for an unauthenticated access. (Episode does not do this.)

The supplied mode bits (in the attributes structure) are used as supplied if (a) a link (VLNK) is being created or (b) the parent directory possesses an *Initial File* **ACL**. Otherwise, the mode bits are weakened by the process umask.

Note: For the creation of directories, files and symlinks, DFS sets the file exporter's *umask* to that of the remote client.

Ordinary files, FIFOs and device files can be created via this call. For the latter, the supplied va_rdev device number is applied.

The new file's link count is set to 1. Its atime, mtime and ctime are all set to the current time.

The extended object UUID (as in the *.objid* field of a **Txvattr** structure) is set to zero.

Any user or system property lists (PLIST) on the parent directory are inherited by the newly created file.

Consult *Initial* **ACL** *and File Creation - Algorithm*, Section 12.11 on page 258 for additional **ACL** processing that might be required for the new object.

The parent directory's ctime and mtime fields are set to the current time.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EISDIR]	An attempt is made to create a directory with this call. Also returned if the desired object already exists, is a directory and the *amode* argument indicates that the caller desired write access (S_IWRITE).
[ENOTDIR]	The object indicated by the*adp* **vnode** is not a directory.
[ENOENT]	The name is not found or the directory has been deleted.
[EEXIST]	An attempt is mad to create a "." or "..", or if the name already exists and the exclusive flag is non-zero.
[EACCES]	The caller does not have the search, write and insert rights to the specified directory (perm_execute, perm_write, perm_insert). Also returned if the desired file already exists and the called does not have the rights specified in the mode argument to the file.

SEE ALSO

Initial **ACL** *and File Creation - Algorithm*, Section 12.11 on page 258. Also, *Obtaining the Identity of the Principal*, Section 11.8.10 on page 243.

NAME

vn_remove — Delete a file in the given directory

SYNOPSIS

```
int vn_remove(
    /* IN */   struct vnode *adp,
    /* IN */   char         *name,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

adp	Pointer to the directory vnode from which file is to be deleted.
aname	Character string name for the file to delete.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Delete a file named *aname* from the directory corresponding to the vnode pointed to by *adp*. A pointer to the caller's credential structure is found in *acred*.

In some kernels, the corresponding vnode operation is passed vnodes for both the parent directory and the file to be deleted. That is, the kernel is expected to look up the file before calling the vnode operation. When this **X-***op* is called on behalf of such a vnode operation, the lookup done by the kernel is wasted, as the **X-***op* does the lookup itself.

DISCUSSION

The specified name is removed from the parent directory and the link count on the object is represents is decremented. The parent's mtime and ctime fields are set to the current time. The ctime of hte object being removed is set to the current time its link count is being decremented.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ENOTDIR]	The object indicated by the*adp* **vnode** is not a directory.
[EPERM]	The specified object is a directory.
	Note: The local OS might dictate that ROOT be allowed to *vop_remove*( ) a directory. In such cases, DFS access should fail.
[ENOENT]	The name (by *aname*is not found in the parent directory.
[EINVAL]	An attempt was made to remove "." or ".." by this call.
[EACCES]	The caller does not have the search, write and delete rights to the specified directory (perm_execute, perm_write, perm_delete).

SEE ALSO

Zero Link Count Files, Section 15.6 on page 332.

NAME

vn_link — Create a hard link

SYNOPSIS

```
int vn_link(
    /* IN */   struct vnode *avp,
    /* IN */   struct vnode *adp,
    /* IN */   char         *aname,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode to be hard-linked.
adp	Pointer to the directory vnode in which the hard link is to be made.
aname	Target name for the hard link.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Link the vnode *avp* to the target name *aname* in the target directory associated with vnode pointer *adp*.

DISCUSSION

An entry is made within the target directory which links to the supplied soruce object. The link count on the source object is incremented. The ctime on the source object along with the ctime and mtime on the target directory are set to the current time.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ENOTDIR]	The object indicated by the*adp* **vnode** is not a directory.
[EACCES]	The caller does not have the search, write and insert rights to the specified directory (perm_execute, perm_write, perm_insert).
[EXDEV]	The sourc eobject and target directory are on different filesets.
[EPERM]	The specified object is a directory. **Note:** The local OS might dictate that ROOT be allowed to *vop_remove*( ) a directory. In such cases, DFS access should fail.
[EEXIST]	The name (by *aname*is already exists in the target directory.

NAME

vn_rename — Rename a file

SYNOPSIS

```
int vn_rename(
    /* IN */   struct vnode *aodp,
    /* IN */   char         *aname1,
    /* IN */   struct vnode *andp,
    /* IN */   char         *aname2,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

aodp	Pointer to the directory vnode where the file currently exists.
aname1	Name of the current file within the above directory.
andp	Pointer to the directory vnode to which the file will move.
aname2	The new name of the file.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Rename a file currently named *aname1* in the directory associated with vnode pointer *aodp*. The file's new name will be *aname2*, and it will be moved to the directory associated with vnode pointer *andp*. A pointer to the caller's credential structure is provided by *acred*.

In some kernels, the corresponding vnode operation is passed vnodes for the parent directories and the subfiles (or subdirectories). That is, the kernel is expected to look up the files before calling the vnode operation. When this **X**-*op* is called on behalf of such a vnode operation, the lookups done by the kernel are wasted, as the **X**-*op* does the lookups itself.

DISCUSSION

If an object with the target's name (*aname2*) already exists:

- If it is a directory, it must be empty.
- If it is a directory, the link count on the target directory is decremented by 1 to account for the removed ".." reference.
- Its link count is decremented by either 1 if it is a file, or 2 if a directory.

If the source object (*aname1*, object being renamed) is a directory, the source directory has its link count decremented by 1 and the target directory has its link count incremented by 1 to account for the moved ".." reference.

The mtime and ctime of both the source and target directories are set to the current time. The ctime of the source object (*aname1*) is set to the current time.

If the target object (*aname2*) originally existed, tis ctime is set to the current time as well.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ENOENT]	The source name does not exist within the source directory.

[EXDEV]	The source object and target directory are on different filesets.
[ENOTDIR]	The object indicated by the*andp* (target) **vnode** is not a directory.
	The object indicated by the*aodp* (source) **vnode** is not a directory.
	The source object (*aname1*) is a directory, the target object (*aname2*) exists and that target object is not a directory.
[EISDIR]	The source object (*aname1*) is not a directory, the target object (*aname2*) exists and that target object is a directory.
[ENOTEMPTY]	The target object (*aname2*) exists and is a not-empty directory.
[EINVAL]	The source object (*aname1*) is a directory, and the target directory is a descendant of it.
	The source object (*aname1*) is either "." or "..".
[EACCES]	The caller does not have the search, write and delete rights to the source directory (perm_execute, perm_write, perm_delete).
	The caller does not have the search, write and insert rights to the target directory (perm_execute, perm_write, perm_insert).
	The directories are different, the source object (*aname1*) is a directory and the caller does not have write access (perm_write) to that source object.

NAME

vn_mkdir — Create a directory

SYNOPSIS

```
int vn_mkdir(
    /* IN */   struct vnode *adp,
    /* IN */   char         *aname,
    /* IN */   struct vattr *aattrs,
    /* OUT */  struct vnode **avpp,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

adp	Pointer to the directory vnode in which the new directory is to be created.
aname	Character string name for the directory to create.
aattrs	Pointer to the attribute structure to give to the new directory.
avpp	If successful, set to the pointer of the newly-created directory's vnode.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Create a directory named *aname* in the directory associated with the *adp* vnode. The new directory's initial attribute structure is passed in *aattrs*, and a pointer to the caller's credential structure appears in *acred*. If *vn_mkdir*( ) is successful, *avpp* is set to the address of the vnode pointer describing the new directory.

DISCUSSION

An empty directory, containing just the "." and ".." entries, is created.

The new directory's user field (*va_uid*) is taken from the DFS authentication structure which itself is obtained from the supplied credentials. Depending on whether SYS5 or BSD semantics are being adhered to, the group field (*va_gid*) is either taken from this PAC or from the parent directory. Consult *Obtaining the Identity of the Principal*, Section 11.8.10 on page 243. If the PAC indicates that this is an unauthenticated RPC operation, the *user* and *group* fields are set instead to –2 and –1, respectively.

Note: If BSD behavior is in effect (va_gid from parent directory), then it seems equally valid to form the va_gid this way even for an unauthenticated access. (Episode does not do this.)

The supplied mode bits (in the attributes structure) are used as supplied if the parent directory possesses an *Initial File* **ACL**. Otherwise, the mode bits are weakened by the process umask.

Note: For the creation of directories, files and symlinks, DFS sets the file exporter's *umask* to that of the remote client.

The new directory's link count is set to 2. Its atime, mtime and ctime are all set to the current time.

The extended object UUID (as the *.objid* field of a **Txvattr** structure) is set to zero.

Any user or system property lists (PLIST) on the parent directory are inherited by the newly created directory.

The parent directory's ctime and mtime fields are set to the current time and its link count is incremented to account for the new ".." reference.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ENOTDIR]	The object indicated by the*adp* (target) **vnode** is not a directory.
[EEXIST]	The directory name already exists.
[EACCES]	The caller does not have search, write and insert rights to the target directory (perm_execute, perm_write, perm_insert).

SEE ALSO

Also, *Obtaining the Identity of the Principal*, Section 11.8.10 on page 243.

NAME

vn_rmdir — Delete a directory

SYNOPSIS

```
int vn_rmdir(
    /* IN */   struct vnode *adp,
    /* IN */   char         *aname,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

adp	Pointer to the directory vnode in which the directory appears.
aname	Character string name for the directory to delete.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Delete a directory named *aname*, which appears within the directory whose vnode is pointed to by *adp*. The caller's credentials are passed in via *acred*.

In some kernels, the corresponding vnode operation is passed vnodes for both the parent directory and the file to be deleted. That is, the kernel is expected to look up the file before calling the vnode operation. When this **X**-*op* is called on behalf of such a vnode operation, the lookup done by the kernel is wasted, as the **X**-*op* does the lookup itself.

DISCUSSION

If the specified directory (*aname*) is empty, its name is removed, its link count is decremented by two (name plus "." reference) and its ctime is set to the current time.

The containing directory (*adp*) has its link count decremented by 1 (to account for the ".." reference) and its mtime and ctime set to the current time.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ENOENT]	The specified directory (*aname*) does not exist.
[EINVAL]	The specified directory (*aname*) is either "." or "..".
[ENOTDIR]	The object specified by the *adp* **vnode** is not a directory.
[EACCES]	The caller does not have the search, write and delete rights to the containing directory (perm_execute, perm_write, perm_delete).
[EBUSY]	The directory being deleted is mounted on top of.
[ENOTEMPTY]	The directory being deleted is not empty.

SEE ALSO

Zero Link Count Files, Section 15.6 on page 332.

NAME

vn_readdir — Read entries from a directory

SYNOPSIS

```
int vn_readdir(
    /* IN */   struct vnode *adp,
    /* IN */   struct uio   *auio,
    /* IN */   struct ucred *acred,
    /* IN */   int          *eofp,
    /* IN */   int          isPX
    );
```

ARGUMENTS

adp	Pointer to the directory vnode being read from.
auio	Pointer to user structure which supplies the I/O arguments, including the address in user space into which the directory data are to be read.
acred	Pointer to the caller's credential structure.
eofp	Optional. A pointer to a returned EOF (end of file) flag.
isPX	Optional. If non-zero, this call is being made by the file exporter on behalf of a remote DFS client.

DESCRIPTION

Read entries from directory *adp*. The user structure pointed to by *auio* supplies the I/O arguments. The **uio** offset is set to a file system-dependent number which represents the logical offset in the directory when the reading is done. This is necessary because the number of bytes returned by *vn_readdir*( ) is not necessarily the number of bytes in the equivalent part of the on-disk directory.

DISCUSSION

Directory entries are read into the supplied buffer. The format of these returned entries is controlled by the *isPX* argument. If it is zero, entries are returned in whatever native format the local OS expects. If this argument is non-zero, the DFS file exporter expects to see entries in **struct dirent** format as defined in *Directory Entry Formats*, Section 13.6.11 on page 301. Fields should be filled in as follows:

.*offset*	The directory offset of whatever (either a valid entry or empty space) FOLLOWS this entry in the directory. This field must be consistent with the **uio** .*uio_offset* argument to this call. Specifically: calling *vn_readdir*( ) with a .*uio_offset* set to the .*offset* from some directory entry results in any and all entries AFTER that one being returned.
.*inode*	An LFS-dependent anode number.
.*recordlen*	The offset (in bytes) from this entry in the output buffer to the next entry. Since directory entries should be aligned on modulo-4 boundaries, this field will be a multiple of 4.
.*namelen*	The length of the file name field, not including the terminating null character.
.*dir_name*	A variable sized file name, including a terminating null character.

Note: The *isPX* argument is present because the DFS file exporter requires an .*offset* field within directory entries while HPUX (along with OSF1, etc.) does not provide such a field in its native entry format. For AIX and SunOS-5 platforms, this call behaves the

same regardless of the setting of this flag.

As many entries as will fit in the supplied buffer, or as remain in the directory, are actually returned. Entries are returned on mod-4 byte boundaries. (The supplied return buffer has this alignment.)

Fields within the uio structure are used as follows.

.uio_iov On entry: a pointer to an array of io vectors describing the buffer to be read (into).

.uio_iovec On entry: the number of io vectors pointed at by *.uio_iov*. Episode fails with [EINVAL] if this number is other than 1.

.uio_offset On entry: the offset, initially 0, at which to begin "reading" directory entries from.

At exit: the offset (LFS-dependent) at which to continue reading directory entries from on the next call.

This field should be treated consistently with the *.offset* field in returned entries.

.uio_segflg UIO_<whatever>

.uio_resid On entry: the total size of the user supplied buffer.

At exit: the number of bytes in the user's buffer that were NOT read into by this call.

If no directory entries are actually returned, the *.uio_resid* field should be left unchanged on return.

The *.uio_offset* argument will initially be set to zero if the caller wishes to read the directory from the beginning. On return, it should be updated by the LFS so that a subsequent call with it will continue at the next entry in the directory. As mentioned above, if on input it is set to the *.offset* field from an entry read earlier, the directory read will commence immediately FOLLOWING that entry.

DFS makes no assumptions regarding the ordering of entries within a directory.

The atime of the directory is set to the current time.

Note: Episode rounds the supplied output buffer size down to a multiple of 512 bytes. The LAST directory entry returned has its *.recordlen* field incremented to cover the remaining space within the final 512 byte block. (it "points" to the next 512 byte boundary)

This behavior exists only to allow the file exporter to manufacture meaningful directory offsets on platforms whose native entries do not contain such a field. It turns out that the higher level exporter cannot manufacture correct offsets. There is no reason for an LFS to mimic this 512-byte block behavior.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL] (Optional - not done by Episode) The supplied buffer is not aligned on a 0-modulo-4 byte boundry.

[ENOTDIR] The object indicated by the *adp* **vnode** is not a directory.

[EACCES] The caller does not have read rights to the directory.

SEE ALSO

Directory Offsets, Section 16.5.1 on page 423, and also, *Directory Entry Formats*, Section 13.6.11 on page 301.

NAME

vn_symlink — Create a symbolic link

SYNOPSIS

```
int vn_symlink(
    /* IN */   struct vnode *adp,
    /* IN */   char         *aname,
    /* IN */   struct vattr *aattrs,
    /* IN */   char         *atargetName,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

adp	Pointer to the directory vnode which will contain the new symlink.
aname	Name of the new link, to be inserted into the directory associated with the *adp* directory vnode pointer.
aattrs	Pointer to the attribute structure to give the new syslink.
atargetName	String name of the path which is the target of the new symlink.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Create a symlink named *aname* within the directory associated with directory vnode pointer *adp*. This symlink's initial attributes are taken from *aattrs*, and its target pathname is taken from *atargetName*. A pointer to the caller's credential structure is present in *acred*.

DISCUSSION

The terminating null at the end of *atargetName* is *not* written into the link object. The link file is created as described under *vn_create*( ), with the following exceptions:

- The target name *aname* cannot already exist.
- The supplied .*va_type* field is ignored and VLNK is used instead.
- Since the permissions on a symlink are never checked, there is no reason for it to inherit an *Initial* **ACL** from the parent directory.
- The parent directory's ctime and mtime fields are set to the current time.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ENOTDIR]	The object indicated by the*adp* **vnode** is not a directory.
[EACCES]	The caller does not have search, write and insert rights to the target directory.
[EEXIST]	The target name (*aname*) already exists within the target directory.

NAME

vn_readlink — Read a symbolic link's value

SYNOPSIS

```
int vn_readlink(
    /* IN */   struct vnode *avp,
    /* IN */   struct uio   *auio,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the symlink vnode being read from.
auio	Pointer to user structure which supplies the I/O arguments, including the address in user space into which the symlink value is to be read.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Read the value (contents) associated with the symlink associated with the *avp* vnode pointer. The user structure pointed to by *auio* supplies the I/O arguments, including where the symlink's value is to be placed. A pointer to the caller's credential structure is passed via *acred*.

DISCUSSION

The symbolic link link is read from the supplied object and deposited into the supplied buffer. A terminating null is *not* appended to the returned link text.

At the time the call is made, the *.uio_offset* field should have been set to 0 and the *.uio_resid* field has been set to the total size of the supplied buffer. When the call returns, the *.uio_resid* field has been decremented by the actual size of the link text copied into the buffer.

The atime of the symbolic link si set to the current time.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL]	The object indicated by the*avp* **vnode** is not a symbolic link (VLNK).

NAME

vn_fsync — Write out all in-memory information for a file

SYNOPSIS

```
int vn_fsync(
    /* IN */   struct vnode *avp,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the file to be flushed to disk.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Flush the state of the file associated with vnode pointer *avp* to disk. The credential structure found in *acred* is ignored.

DISCUSSION

Any modified data or status for the specified file is written to disk along with data from the VM system.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vn_inactive — The given vnode is no longer referenced

SYNOPSIS

```
int vn_inactive(
    /* IN */   struct vnode *avp,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode no longer being actively referenced by the system.
acred	Pointer to the caller's credential structure.

DESCRIPTION

The vnode pointed to by *avp* is no longer being referenced by any agent in the vnode layer, and may be safely deallocated. This function is called when a vnode's reference count drops to zero. If the corresponding file also has a zero link count, any space allocated for it on disk may also be freed at this point. The credential structure is ignored.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

This call is made by the **vnode** (fs independent) layer to indicate that a **vnode** is no longer referenced because its *.v_count* field has been decremented to 0. The deletion does *not* occur on a readonly fileset (a *.readonly* replica or a *.backup* clone) or a fileset mounted readonly for local access. Files on a *.readonly* replica are only destroyed by an explicit *vol_depletd*( ) operation. Files on a *.backup* clone are only destroyed when their fileset is deleted or as the result of a *vol_reclone*( ) operation.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vn_bmap — Logical to physical block number mapping

SYNOPSIS

```
int vn_bmap(
    /* IN */   struct vnode *avp,
    /* IN */   long         abn,
    /* OUT */  struct vnode *anvp,
    /* OUT */  long         anbn
    );
```

ARGUMENTS

avp	Pointer to the vnode to consider.
abn	Logical block number to map.
anvp	Pointer to the vnode representing the associated physical device.
anbn	Set to the associated physical block number.

DESCRIPTION

Map logical block number *abn* in the file associated with vnode pointer *avp* to its physical device and block number. A vnode pointer describing the corresponding physical device is returned in *anvp*, and *anbn* is set to the physical block number mapped to by *abn*.

This **X** methods of mapping files to virtual memory (see Section 16.8 on page 460).

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vn_strategy — Asynchronous read/write interface between file blocks and buffers

SYNOPSIS

```
int vn_strategy(
    /* IN */   struct buf  *abp
    );
```

ARGUMENTS

abp Pointer to buffer descriptor.

DESCRIPTION

This function provides an asynchronous, block-oriented interface to read or write a logical block from a file into or out of a buffer. The only parameter, *bp*, is a pointer to a buffer header which contains a pointer to the affected vnode. This function does *not* write through the buffer cache, and is used by the kernel's virtual memory manager (if any) to perform I/O to and from virtual memory.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vn_bread — Read a logical block from a file into a buffer

SYNOPSIS

```
int vn_bread(
    /* IN */   struct vnode *avp,
    /* IN */   daddr_t      albn,
    /* OUT */  struct buf   **abpp,
    /* OUT */  long         *asizep
    );
```

ARGUMENTS

avp	Pointer to the vnode for the file being read from.
albn	Logical block number to read.
abpp	Set to the pointer to a buffer header containing the desired block.
asizep	Pointer to location at which to store the buffer size.

DESCRIPTION

Read logical block *albn* from the file associated with the vnode pointed to by *avp*. Parameter *abpp* is set to point to a buffer header pointer which describes the buffer just read in.

This **X**-*op* is not used in kernels that have more general methods of mapping files to virtual memory (see Section 16.8 on page 460).

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

Extended Vnode Interface, Section 16.8 on page 460.

NAME

vn_brelse — Release a buffer header pointer

SYNOPSIS

```
int vn_brelse(
    /* IN */   struct vnode *avp,
    /* IN */   struct buf   *abp
    );
```

ARGUMENTS

avp	Pointer to the vnode to which the buffer header belongs.
abp	Pointer to the buffer header to be released.

DESCRIPTION

Release the buffer header pointed to by *abp*, which is associated with the vnode pointed to by *avp*. The buffer header pointer was originally generated by *vn_bread*( ).

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vn_lockctl — File and record locking interface

SYNOPSIS

```
int vn_lockctl(
    /* IN */   struct vnode *avp,
    /* IN */   struct flock *af,
    /* IN */   int          acmd,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to vnode to be locked.
af	Pointer to lock structure.
acmd	Locking command.
acred	Pointer to the caller's credential structure.

DESCRIPTION

This function provides System V-style locking on vnodes. A pointer to the vnode to be operated upon is found in *avp*, a pointer to the desired lock structure in *af*, the lock command to be executed in *acmd*, and a pointer to the caller's credential structure in *acred*.

DISCUSSION

This operation is for local access ony. It is NEVER called by the file exporter.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

vn_fid — Return the file ID associated with a vnode

SYNOPSIS

```
int vn_fid(
    /* IN */   struct vnode *avp,
    /* OUT */  struct fid   **afidpp
    );
```

ARGUMENTS

avp	Pointer to the vnode whose file ID is to be discovered.
afidpp	Set to the address of a pointer containing the file ID associated with vnode *avp*.

DESCRIPTION

Given a pointer to a vnode *avp*, set *afidpp* to point to a file ID pointer describing the file ID associated with the vnode.

This **X**-*op* is called only on behalf of the NFS server. It is not called on behalf of the Protocol Exporter, which uses a different interface (see Section 16.8 on page 460).

In most kernels, this **X**-*op* allocates the *struct fid* that is requested; that structure is later freed by the kernel, rather than by another **X**-*op*. In IBM's AIX kernel, the *struct fid* is allocated by the caller. Accordingly, the parameter *afidpp* is a **struct fid *** rather than a **struct fid ****.

DISCUSSION

An NFS-style **fid** (file ID) is allocated, filled in and returned to the caller of this operation. The storage for this **fid** should be allocated via *kmem_alloc*( ) since it will be freed by the FS-independent kernel at a later time.

Any information stored within the **fid** (for example, an LFS-dependent anode index and generation number) should be stored in network byte order. Information within these **fid**s must be stored in network byte order to allow the movement of a fileset to a server with a different byte ordering while the **fid** is held at some client.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

16.8 Extended Vnode Interface

16.8.1 Generalized Differences

A number of **X**-*ops*, in addition to those described in the *Base Vnode Interface* section, have been added to meet the needs of various components of DFS. There are the following categories:

1. **Support for Protocol Exporter**. The *vn_afsfid*( ) **X**-*op* is used by the Protocol Exporter analogously to the use of the *vn_fid*( ) **X**-*op* by the NFS server.

2. **Support for O**-*op* **wrapper functions**. As described in earlier chapters, the glue functions (''**O**-*ops*'') must perform fileset synchronization to avoid conflicts between individual file references and whole-fileset operations such as cloning, dumping, or restoring. The *vnx_getvolume*( ) **X**-*op* is used to identify a file's fileset. Referring to the pseudocode of Section C.3 on page 246, this **X**-*op* would be called from *ReferenceCorrespondingVolume*( ).

3. **Support for vnode Operations in newer kernels**. The *vn_cmp*( ) and *_vnctl* **X**-*ops* were added to support more recent versions of Sun's SunOS kernels. The *vn_map*( ) and *vn_unmap* **X**-*ops* were added to support IBM's AIX kernel. The *vn_reclaim* **X**-*op* was added to support OSF's OSF/1 kernel. Additional additions for SunOS 5 include *vn_read*( ), *vn_write*( ), *vn_realvp*( ), *vn_rwlock*( ), *vn_rwunlock*( ), *vn_seek*( ), *vn_space*( ), *vn_getpage*( ), *vn_putpage*( ), *vn_addmap*( ), *vn_delmap*( ), *vn_pageio*( ) and *vn_frlock*( ), which is equivalent to *vn_lockctl*( ). Also, the following were added for HP/UX: *vn_pagein*( ), *vn_pageout*( ).

   As noted in *Overview of Interfaces to the LFS*, in Section 11.3 on page 235, these are LFS-specific facilities. They are thus not documented here.

4. **Support for ACLs**. The **X**-*ops* *vn_getacl*( ) and *vn_setacl*( ) **X**-*ops* are used by the Protocol Exporter to read and write **ACL**s. There is also a system call interface, implemented using these **X**-*ops*, which makes **ACL**s available to local users.

5. **Support for VM**. **Hold**, **Rele**, **Lock**, **Unlock**, **Pvnlock**, **Getlength**, **Setlength**, **Pin**, **Unpin**, **Nlrw**, **Cached**, **Update**, **Readahead**, **Delaywrite**.

   The idea of supporting VM by calling VM-related **X**-*ops* from the **N**-*op* wrapper functions is no longer being used in new code, but not all old code has been revised at this writing, and so the VM-related **X**-*ops* are still included in the **X**-*op* specifications.

NAME

vn_getvolume — Get the vnode's fileset

SYNOPSIS

```
int vn_getvolume(
    /* IN */   struct vnode  *avp,
    /* OUT */  struct volume *avolpp
    );
```

ARGUMENTS

avp	Pointer to the vnode whose fileset is desired.
avolpp	Pointer to place to put fileset pointer.

DESCRIPTION

If the **vnode**'s fileset is in the *Fileset Registry*, put a pointer to the fileset's entry in the registry in the location indicated by *avolpp*. Otherwise, return an error code.

DISCUSSION

This operation extracts a fileset ID from the **vnode** and uses it to obtain and return the volume structure for the fileset on which it resides.

Note: This implies that LFS vnodes either contain or point to a fileset ID.

The actual fileset structure (**struct volume**) should be obtained by an up-level call to the *Fileset Registry* routine for that purpose, *volreg_Lookup*( ) defined in Section 15.15 on page 399.

RETURN VALUE

This function will return the value that *volreg_Lookup*( ) returned to it. If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[ENODEV]	The object does not reside on an exported fileset (there is no fileset for the **vnode**).

SEE ALSO

volreg_Lookup( ) defined in Section 15.15 on page 399.

NAME

vn_afsfid — Return file ID suitable for exporter

SYNOPSIS

```
int vn_afsfid(
    /* IN */   struct vnode   *avp,
    /* OUT */  struct afsFid  *af,
    /* IN */   int           wantv
    );
```

ARGUMENTS

avp	Pointer to the vnode whose file ID is desired.
aafsfidp	Pointer to the file ID structure as used by the Protocol Exporter.
wantv	Flag indicating whether the *volume* and *cell* fields in the file ID structure should be filled in.

DESCRIPTION

vn_fid( ) and *vn_afsfid*( ) have to be separate **X**-*ops*. Given a pointer to a vnode *avp*, set *af* to point to a file ID pointer describing the file ID associated with the vnode.

This **X**-*op* is called only on behalf of Protocol Exporter.

vn_fid( ) is called by the NFS server.

DISCUSSION

Fields within the returned **afsFid** are constructed as follows:

.*Vnode*	Set to the LFS-dependent (non-canonical) anode number.
.*Unique*	Set to the anode generation number.
.*Volume*	If *wantv* is non-zero, set to the fileset ID of the fileset on which this object resides.
.*Cell*	If *wantv* is non-zero, the *.high* and *.low* subfields should be set to 1 and 0, respectively.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

vn_fid( ) which does a similar function.

NAME

vn_getacl — Get the vnode's access control list

SYNOPSIS

```
int vn_getacl(
    /* IN */   struct vnode   *avp,
    /* OUT */  struct dfs_acl *aaclp,
    /* IN */   long           aw,
    /* IN */   struct ucred   *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode whose **ACL** is desired.
aaclp	Pointer to the buffer to which the contents and length of the chosen **ACL** are written.
aw	Specifies which of a file's **ACL**s is desired.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Read the **ACL** associated with the file whose **vnode** pointer is *avp*. A pointer to the caller's credentials is contained in *acred*. Argument *aw* describes which of a file's **ACL**s is desired. For regular files, there is only one possbility, but for directories, either the directory's own **ACL**, or the default **ACL** for its subdirectories, or the default **ACL** for its subfiles may be specified.

The format of the **ACL** is described in the topic, **ACL** *Storage Format*, Chapter 9 on page 165.

DISCUSSION

The **ACL** designated by the *aw* argument is encoded as described in the above mentioned document and returned. The *aw* argument can take the following values.

VNX_ACL_REGULAR_ACL
: The **ACL** for the object itself is returned.

VNX_ACL_DEFAULT_ACL
: The *Initial Container* **ACL** for a directory object is returned. If the object in question is not a directory, an [EINVAL] error is returned.

VNX_ACL_INITIAL_ACL
: The *Initial File* **ACL** for a the object in question is not a directory; an [EINVAL] error is returned. If the specified **ACL** exists for the object in question, it is returned. If *aw* equals VNX_ACL_REGULAR_ACL, it may be necessary to coerce the "required" (user_obj, group_obj, other_obj) entries to agree with the mode bits on the file. Consult the "Relationship between an **ACL** and the UNIX protection mode bits" section of the LFS **ACL** specification.

If the requested **ACL** does not exist, matters are somewhat more complicated. A minimal **ACL** is constructed as is described in **ACL** *Creation From Mode Bits Algorithm*, Section 12.10 on page 257. The actual specifics differ slightly depending on the type of **ACL** requested (the *aw* argument):

VNX_ACL_REGULAR_ACL
: The mode bits used in the **ACL** synthesis are taken from the object's mode.

VNX_ACL_DEFAULT_ACL
: The default is used.

VNX_ACL_INITIAL_ACL

These initial **ACL**s are synthesized from mode bits of 0777 (**Read-Write-Execute** for all).

If the object in question does possess a regular **ACL**, the default realm in the constructed initial **ACL** is set to the value from that regular **ACL** instead of the LOCAL-REALM (local cell, that is).

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL] Returned if the *aw* argument is not one of the values VNX_ACL_REGULAR_ACL, VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL.

Also returned if *aw* is either VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL and the object (*avp*) is not a directory.

SEE ALSO

ACL *Storage Format*, Chapter 9 on page 165, and **ACL** *Creation From Mode Bits Algorithm*, Section 12.10 on page 257.

NAME

vn_setacl — Set the vnode's access control list

SYNOPSIS

```
int vn_setacl(
    /* IN */   struct vnode *avp,
    /* IN */   char         *aaclp,
    /* IN */   struct vnode *asvp,
    /* IN */   long         adestw,
    /* IN */   long         asrcw,
    /* IN */   struct ucred *acred
    );
```

ARGUMENTS

avp	Pointer to the vnode whose **ACL** is to be overwritten.
aaclp	Pointer to the buffer from which to read the new length and contents of the chosen **ACL**.
asvp	Pointer to the vnode structure (if any) to use as the ''shared'' **ACL** object.
adestw	Indication of which of the file's **ACL**s is to be set.
asrcw	Indication of which of the source file's **ACL**s is to be copied.
acred	Pointer to the caller's credential structure.

DESCRIPTION

Set the contents of the **ACL** associated with the file whose **vnode** pointer is *avp*. If *aaclp* is non-null, write the **ACL** directly from that structure. If *aaclp* is null and *asvp* is non-null, then copy the **ACL** of *avp* from the **ACL** associated with the file whose **vnode** pointer is *asvp*. This copying gives the underlying file system the opportunity to detect sharing of **ACL**s. However, the **ACL**'s are not *semantically* shared; if a subsequent call to *vn_setacl*( ) modifies the **ACL** of *avp*, the **ACL** of *asvp* is not modified with it. Exactly one of *alen* and *asvp* must be non-zero. A pointer to the caller's credentials is passed via *acred*. The format of the information in the **ACL** is identical to that in *vn_getacl*( )

DISCUSSION

This operation sets an **ACL** on the target object (*avp*). The *destw* argument indicates which type of **ACL** is being set and can take the following values:

VNX_ACL_REGULAR_ACL
: The **ACL** on the object itself is set.

VNX_ACL_DEFAULT_ACL
: The initial directory **ACL** on a directory object is set.

VNX_ACL_INITIAL_ACL
: The initial file **ACL** on a directory object is set.

The **ACL** to be applied to the target is specified by either the *asvp* or *aaclp* arguments, exactly 1 of which should be non-NULL:

- **If** *aaclp* **is non-NULL**:

  The **ACL** has been supplied as an argument to this call. It's representation is described in the above mentioned LFS **ACL** specification.

  If the supplied **ACL** is "empty" (.*dfs_acl_len* == 0), any existing target **ACL** is removed.

If the supplied **ACL** is present (*.dfs_acl_len* > 0) and the *destw* argument equals VNX_ACL_REGULAR_ACL, it may be necessary to update the mode bits of the object.

The *srcw* argument is unused.

- **If** *asvp* **is non-NULL**:

The **ACL** to be applied to the target is copied from the source object indicated by *asvp*. The *srcw* argument indicates which **ACL** on the source is to be used and will have one of the VNX_ACL_XXX values listed above. There is no requirement that the source and target **ACL** types agree.

Note: If either *srcw* or *destw* equal VNX_ACL_REGULAR_ACL, the internal Episode implementation of **aCL**s makes itself apaprent here.

Normally, **ACL** operations under Episode behave as follows. When one sets an **ACL**, the file's mode bits are changed to agree with the required **ACL** entries. When a file's mode bits are changed (**chmod**), its **ACL** is not physically changed on disk. Instead, the mode bits are "merged" into the **ACL** dynamically during **ACL** checks and when **ACL**s are returned to user space.

If *srcw* equals VNX_ACL_REGULAR_ACL, the *vn_setacl*( ) operation fetches the source **ACL** as it resides on disk, without coercing the **ACL**'s required entries to agree with the file's mode bits. If *destw* equals VNX_ACL_REGULAR_ACL, the **ACL** is applied to the target object without changing the file's mode bits.

Finally, the ctime on the target object is set to the current time.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[EINVAL] Returned if both *asvp* and *aaclp* are non-NULL.

Returned if both *asvp* and *aaclp* are NULL.

Returned if *adestw* is not equal to one of the legal values VNX_ACL_XXX listed above.

Returned if *asvp* is non-NULL and *asrcw* is not one of the legal VNX_ACL_XXX values listed above.

Returned if *adestw* is either VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL and the target object (*avp*) is not a directory.

Returned if *asrcw* is either VNX_ACL_DEFAULT_ACL or VNX_ACL_INITIAL_ACL, *asvp* is non-NULL and the source object is not a directory.

[EINVAL] If *aaclp* is non-NULL and the supplied **ACL** is in an illegal or inconsistent state. Specifically, a valid **ACL** requires that:

- The **ACL**'s length is consistent with its actual contents.
- The *.mgr_type_field* is correct.
- The required entries are all present.
- The *entry_type_foreign_user* and *entry_type_foreign_group* entries must contain a realm UUID that differs from the default one.

- The *user_obj* entry grants **perm_control** rights.

[EPERM] Returned if The caller does not have administrative rights to the target object (*avp*).

SEE ALSO

See *vn_getacl*( ), **ACL** *Storage Format*, Chapter 9 on page 165, and **ACL** *Creation From Mode Bits Algorithm*, Section 12.10 on page 257.

Chapter 17

DCE DFS VFS+ Extended Credential Package

17.1 The xcred Package

This chapter describes the interface used for authentication within the VFS+ layer. It presents an extensible and upwards-compatible interpretation of the VFS UNIX **credential** structure. Section 17.2 provides a short overview of this extended credential package, with Section 17.3 on page 470 specifying the exported interface. Important data structures are detailed in *Extended Credential Data Types*, Section 13.4 on page 289. The set of reserved attributes for the user's property list is described in Section 13.4.3 on page 290.

17.2 Package Overview

This chapter, describing VFS client context extensions, does not describe new VFS interface functions. Rather, it describes what is essentially a subroutine library allowing VFS functions to interpret the VFS credentials structure in a more general manner.

Specifically, different file systems may have different authentication requirements, and one system may not be satisfied with the authentication guarantees made by a particular file exporter.

17.2.1 Interface Overview

To give an idea of the problems must be solved here, consider some of the possible authenticated identities a DCE fileset might have to evaluate. First, a call might enter via a authenticated RPC call, in which case the UUID would be used as the user's identity. A call might also enter via a kernel--uthenticated access on the File Exporter itself. In this mode of authentication, the user is identified by his or her UNIX UID to the DCE fileset. A third user accessing the file system might come in via an essentially unauthenticated NFS call, also specifying a UNIX UID for the accessing user. However, this access path, while it results in a UNIX UID identifying the user, probably should be treated as less secure than a kernel-authenticated access.

So, a client context is created and associated with a typed authentication structure. There are various calls in the interface to access these objects. One call might ask for the string name for the authenticated user. Another might ask for a 32- bit UID from the authentication library (and the interface might call the protection server to map a name to an ID if necessary to provide the information). Since the authentication level for a 32-bit UID might differ considerably (compare getting that UID from the kernel versus getting it from an unprotected NFS RPC call), some hint as to what level of authentication is permitted needs to be provided.

17.3 xcred Functions

This section describes the functions exported by the credential package.

NAME

xcred_Init — Initialize the module

SYNOPSIS

```
long  xcred_Init(
    /* No parameters */
    );
```

ARGUMENTS

No parameters.

DESCRIPTION

This function initializes the internal state of the xcred module. It must be called at least once before any of the other exported routines are invoked. Failure to do so will cause unpredictable results from the xcred module. Only the first call will actually perform initialization, all others will be no-ops.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_ALLOC_FAILED]
There was a memory allocation failure.

NAME

xcred_Create — Create an extended credential structure

SYNOPSIS

```
long  xcred_Create(
    /* OUT */  xcred_t   **anewXCredPP
    );
```

ARGUMENTS

newXCredPP Address of xcred pointer to fill.

DESCRIPTION

This function returns the location of a new (empty) xcred structure.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_ALLOC_FAILED]
There was a memory allocation failure.

NAME

xcred_Hold — Increment the reference count on a given xcred structure

SYNOPSIS

```
long  xcred_Hold(
    /* IN */  xcred_t   *axcredP
    );
```

ARGUMENTS

axcredP Pointer to the xcred structure to update.

DESCRIPTION

This function simply increments the reference count on the relevant structure.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_INVAL_PARAM]
The xcred was invalid.

[XXCRED_EC_BAD_REFCOUNT]
The refcount was already less than or equal to zero.

NAME

xcred_Delete — Mark an xcred structure for deletion

SYNOPSIS

```
long  xcred_Delete(
    /* IN */  xcred_t   *axP
    );
```

ARGUMENTS

axP Pointer to the xcred structure to delete.

DESCRIPTION

This function marks the xcred structure for deletion. This deletion will occur when the *xcred_Release*( ) function is called with this xcred structure as an argument.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

NAME

xcred_Release — Decrement the reference count on a given xcred structure

SYNOPSIS

```
long  xcred_Release(
    /* IN */  xcred_t   *axcredP
    );
```

ARGUMENTS

axcredP Pointer to the xcred structure to update.

DESCRIPTION

This function decrements the reference count on the relevant structure. If the reference count drops to zero, or a call to *xcred_Delete* was previously issued against the xcred structure, then the structure is freed.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_INVAL_PARAM]
The xcred was invalid.

[XCRED_EC_BAD_REFCOUNT]
The refcount was already less than or equal to zero.

NAME

xcred_AssociateCreds — Associate the given xcred and UNIX ucred

SYNOPSIS

```
long  xcred_AssociateCreds(
    /* IN */   xcred_t       *axcredP,
    /* OUT */  struct ucred  **aucredPP,
    /* IN */   long          apag
    );
```

ARGUMENTS

axcredP Pointer to the xcred structure for the association.

aucredPP Pointer to the related UNIX credential structure.

apag pag to allocate for linkage, if any.

DESCRIPTION

This function associates the given xcred and ucred structures so that having a pointer to the ucred structure will permit the location of the xcred structure with which it is associated. If *apag* is not-specified, (equal to zero), this function will generate one.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[CRED_EC_ALLOC_FAILERED_EC_INVAL_PARAM]
An invalid structure address was passed in.

[XCRED_EC_BAD_REFCOUNT]
The refcount was already less than or equal to zero.

[XCRED_EC_CRED_FULL]
The UNIX cred can't handle additional xcreds.

[OSI_NOPAG] The *apag* value is invalid.

SEE ALSO

xcred_UCredToXCred( ) .

NAME

xcred_UCredToXCred — Obtain pointer to ucred associated with the given xcred

SYNOPSIS

```
long  xcred_UCredToXCred(
    /* IN  */  struct ucred  *aucredP,
    /* OUT */   xcred_t      **axcredPP
    );
```

ARGUMENTS

aucredP Pointer to the UNIX credential structure to examine.

axcredPP Pointer to the xcred structure for the association.

DESCRIPTION

This function obtains the corresponding xcred structure from the given UNIX cred, increments its reference count by 1, and returns its location. If an xcred structure has not been associated with the ucred, *axcredPP* will be set to null and the appropriate error code is returned.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_INVAL_PARAM]
An invalid structure address was passed in.

[XCRED_EC_BAD_REFCOUNT]
The refcount on the xcred was already less than or equal zero.

[XCRED_EC_NO_ENTRY]
The given UNIX cred had no associated xcred.

NAME

xcred_FindByPag — Obtain pointer to xcred

SYNOPSIS

```
long  xcred_FindByPag(
    /* IN */   long          apag
    );
```

ARGUMENTS

apag pag to match against.

DESCRIPTION

This function returns a pointer to the xcred structure that contains the pag equal to *apag*. If no match is found then NULL is returned.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[non_NULL] This function returns a non_NULL pointer to the xcred structure upon success.

ERRORS

[NULL_Pointer] No match was found. This is a FAILURE.

NAME

xcred_PutProp — Enter a property list attribute-value pair into an xcred

SYNOPSIS

```
long  xcred_PutProp(
    /* IN */   xcred_t       *axcredP,
    /* IN */   char          *aattributeP,
    /* IN */   long          aattrLength,
    /* IN */   char          *avalueP,
    /* IN */   long          aLength,
    /* IN */   char          *abaseAttrP,
    /* IN */   long          abaseAttrLength
    );
```

ARGUMENTS

axcredP	Pointer to the xcred structure to modify.
aattributeP	Name of the property list entry.
aattrLength	Number of bytes pointed to by *aattributeP*.
avalueP	A counted array of bytes comprising the value portion.
alength	The length of the *avalueP* field in bytes.
abaseAttrP	Name of the attribute from which this plist entry is derived. **Note:** This parameter is now ignored.
abaseAttrLength	Number of bytes pointed to by *abaseAttrP*. **Note:** This parameter is now ignored.

DESCRIPTION

This function modifies the contents of the specified attribute associated with the given xcred. If *avalueP* is null, the attribute is deleted from the property list. The *avalueP* parameter's type is a counted array of bytes, not a character string, despite the declaration, and thus may contain null characters that would otherwise terminate a C language character string. This is also true of *aattributeP*.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_INVAL_PARAM]
An invalid parameter was passed in.

[XCRED_EC_BAD_REFCOUNT]
The refcount on the xcred was already less than or equal zero.

[XCRED_EC_NO_BASE_ATTRIBUTE]
The base attribute specified doesn't exist.

NAME

xcred_GetProp — Retrieve attribute value from an xcred's property list

SYNOPSIS

```
long  xcred_GetProp(
    /* IN */   xcred_t       *axcredP,
    /* IN */   char          *aattributeP,
    /* IN */   long          aattrLength,
    /* OUT */  char          **avaluePP,
    /* OUT */  long          *arealLengthP
   );
```

ARGUMENTS

axcredP	Pointer to the xcred structure to examine.
aattributeP	Name of the property to retrieve.
aattrLength	Number of bytes pointed to by *aattributeP*.
avalueP	Pointer to counted array of bytes in which to store the retrieved value, if any (may be null).
arealLengthP	The number of bytes placed into **avaluePP*.

DESCRIPTION

This function returns the value from the specified attribute for an xcred. The value is placed in *avaluePP*, and its length is placed in *arealLengthP*.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_BAD_REFCOUNT]
The reference count on the xcred was less than or equal zero.

[XCRED_EC_NO_ENTRY]
There was no entry named attribute associated with the xcred.

[XCRED_EC_NOT_ENOUGH_ROOM]
There wasn't enough room in the caller's buffer to copy out the value component.

[XCRED_EC_ALLOC_FAILED]
There was a memory allocation failure.

NAME

xcred_GetUFlags — Get user flags from an xcred

SYNOPSIS

```
long  xcred_GetUFlags(
    /* IN */  xcred_t   *xcredP
    );
```

ARGUMENTS

xcredP Pointer to the xcred structure from which the flags are to be obtained.

DESCRIPTION

This function returns the value of the user defined flags for the given xcred structure. Currently this is the value of the uflags field.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always returns success. This function succeeds if:

[value] This function returns success. The value of the uflags field is returned (here).

ERRORS

None.

NAME

xcred_SetUFlags — Set user flags in an xcred

SYNOPSIS

```
long  xcred_SetUFlags(
    /* IN */  xcred_t   *xcredP,
    /* IN */  long      orflags,
    /* IN */  long      andflags
    );
```

ARGUMENTS

xcredP	Pointer to the xcred structure from which the flags are to be set.
orflags	Flags that are to be set (turned on).
andflags	Flags that are to be unset (turned off).

DESCRIPTION

This function sets the value of the user defined flags in the xcred structure. Currently this is the value of the uflags field. Those flag bits set in the *orflags* parameter are set, and the flag bits set in the *andflags* parameter are unset.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

None.

NAME

xcred_EnumerateProp — Apply a function to all elements of the given xcred's property list

SYNOPSIS

```
long  xcred_EnumerateProp(
    /* IN */  xcred_t   *axcredP,
    /* IN */  long      (*aprockP)(),
    /* IN */  long      *arockP
    );
```

ARGUMENTS

axcredP Pointer to the xcred structure being operated on.

aprocP Ptr to function to call on each member of the xcred's property list.

arockP The first parameter passed to **aprocP*( ). The given routine is actually given three parameters, the second being *axcredP* and the third a pointer to the parameter list entry currently being operated upon.

DESCRIPTION

This call applies the function given in *aprocP* to all the elements of an xcred's property list. The function will be passed three arguments, the first is *arockP*, the second is *axcredP*, and last is a pointer to the plist entry currently being operated upon. The function should return a long value which is 0 for success, and any non-zero value to indicate falure. Note that the xcred is read-locked for the duration of the call, implying that the function being applied should not try to lock the xcred itself.

This function will halt at the first unsuccessful invocation of the *aprocP* function, and a non-zero value will be returned in this case.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function succeeds if:

[error_status_ok] This function returns success.

ERRORS

[XCRED_EC_INVAL_PARAM]
A parameter was illegal.

[XCRED_EC_BAD_REFCOUNT]
The xcred's reference count was less than or equal zero.

[any_other_non_zero]
This function did not complete successfully. The function was not applied to all elements of the xcred's property list.

NAME

xcred_DeleteEntry — Mark a property list entry as deleted

SYNOPSIS

```
long  xcred_DeleteEntry(
    /* IN */  xcred_t                  *axcredP,
    /* IN */  struct xcred_PlistEntry  *aplp
    );
```

ARGUMENTS

axcredP Pointer to the xcred structure being operated on.

aplp The property list entry to mark as deleted.

DESCRIPTION

This function marks a property list entry as deleted. This function is only to be called from within a function that is invoked via the *xcred_EnumerateProp*( ) function (the only legal source for xcred_PListEntry objects). It assumes this is the case and does not lock the xcred structure.

RETURN VALUE

If this function succeeds, it returns a value of zero. This function always succeeds. This function succeeds if:

[error_status_ok] This function always returns success.

ERRORS

None.

SEE ALSO

Section 13.4.1 on page 289.

Index