HPSS Programmer's Reference

Generated on Fri Aug 9 2024

Contents

Chapter 1

Introduction

The *HPSS Programmer's Reference* describes areas of HPSS which can be used in custom or site applications.

Preface Contains contractual and trademark information.

Overview The Overview contains a brief description of the HPSS Client API and general concepts of programming for HPSS.

Client API Tutorials and Best Practices Client API Best Practices The Best Practices Guide contains a tutorial-style discussion of the HPSS Client API with examples.

API Reference Links:

- Client API Describes available data structures and functions for the Client API.
- HPSS Networking Library Describes available data structures and functions for the HPSS network library.
- Hash Interface Describes available data structures and functions for the HPSS hashing library.
- HPSS XML Interface Describes available data structures and functions for the HPSS XML module.
- HPSS Environment Variable Module Describes available data structures and functions for the HPSS Environment module.
- 64-bit Math Library Describes available data structures and functions for the HPSS 64-bit math library.
- Gatekeeper Site Library Describes available data structures and functions for the HPSS Gatekeeper Site library.
- Account Validation Site Library Describes available data structures and functions for the HPSS Accounting Site library.
- HPSS Access Control List APIs Describes available data structures and functions for the HPSS ACL Library.
- HPSS Client API Extensions Describes available data structures and functions for the HPSS Configuration Extensions.

Glossary of Terms and Acronyms A brief glossary of technical terms and acronyms.

Further Reading Additional reading, papers, specifications, etc.

Developer Acknowledgements Developer Acknowledgements

The *HPSS Programmer's Reference* is generated from HPSS source and should generally be accurate, but sometimes mistakes sneak through, spelling errors happen, or additional information may be useful. Please contact HPSS support if you find errors in this reference.

Chapter 2

Preface

2.1 Copyright notification

2.2 Trademark usage

Microsoft Windows is a registered trademark of Microsoft Corporation.

Other brands and product names appearing herein may be trademarks or registered trademarks of third parties.

Chapter 3

Overview

The High Performance Storage System (HPSS) provides scalable parallel storage systems for highly parallel computers as well as traditional supercomputers and workstation clusters. Concentrating on meeting the high end of storage system and data management requirements, HPSS is scalable and designed for large storage capacities and to use network-connected storage devices to transfer data at rates up to multiple gigabytes per second. Listed below are the programming interfaces for accessing data from HPSS.

3.1 Client API

3.1.1 Purpose

The purpose of the Client API is to provide an interface which mirrors the POSIX.1 specification where possible to provide ease of use for the POSIX application programmer. In addition, extensions are provided to the programmer to take advantage of the specific features provided by HPSS, such as storage/access hints passed on file creation, parallel data transfers, migration, and purge.

Consult the Client API Tutorials and Best Practices for a discussion of programming with the Client API.

3.1.2 Components

The Client API consists of these major parts:

- Authentication
- File Open, Create, and Close
- File Data
- Buffered Data
- Parallel I/O
- Directory Creation and Deletion
- Directory Data
- File Attribute
- File Name
- Working Directory

- Data Structure Conversion
- User-defined Attributes
- Fileset and Junction
- ACLs
- HPSS Object
- HPSS Statistics
- Client API Control
- Trashcan

3.1.3 Constraints

The following constraints are imposed by the Client API:

- The validity of open files and directories at the time of fork is undefined in the child process.
- The validity of open files and directories is lost across calls to any of the family of exec calls.
- The designed client API works only with applications that make use of the Core Server. In particular, this API is not designed to meet the needs of clients that will perform the Name Service functionality internally or will bypass the Core Server when performing storage operations.
- The client API does not support the file locking interfaces (for example, fcntl(2) & flock(2)) found in most UNIX operating systems. Because of this, it is the responsibility of the application to prevent interceding update problems by handling any necessary serialization.
- Multithreaded applications should take care to serialize calls to exit(). Per the ISO C standard, multiple calls to exit() are undefined.

3.1.4 Libraries

Applications issuing HPSS Client API calls must link the following library:

libhpss.a HPSS client library

Applications issuing requests to the HPSS POSIX API must link the following library:

libhpssposix.a HPSS POSIX client library

3.1.5 User-defined Attributes Search APIs

Searching should only be done on user-defined attributes which have an index defined over them. Indexes can be defined on the system running DB2 using the xmladdindex utility. Additionally, because of speed concerns when searching, if the XML for each result is not required, the hpss_UserAttrGetObjQuery() interface should be used instead of the hpss_UserAttrGetXMLObj() interface.

3.1.6 HPSS Checksum

HPSS checksum functions provide a method of error detection. Changes in a file can be detected; however, neither the location of the change nor the amount of change can be determined from the checksum. Checksumming does not enable recovery from data errors. Mirrored copies, RAIT, or other error correcting mechanisms are required to recover from data.

The checksum functions work by generating a checksum engine context for the chosen algorithm. That engine context is then used by several generic functions to append information, finalize the hash, reset the engine context, or free the engine context.

To use the HPSS checksum functions, follow the following model:

1. Authenticate with HPSS
2. Open a file
3. Create a hash engine (hpss_HashCreateXXX)
4. Read the file and append each data buffer to the Hash Engine (hpss_HashAppend(), hpss_HashAppendStr())
5. Close the file.
6. Finalize the checksum (hpss_HashFinish() or hpss_HashFinishHex())
7. At this point you have a finished checksum which can be used at a later time to determine if the contents of the file have changed.

The following algorithms are supported: SHA1, SHA224, SHA256, SHA384, SHA512, MD5, Crc32, and Adler32.

3.1.7 PIO Model

A description of a simple programming model for PIO is provided in this section. The PIO functions use a programming model of one coordinator and one or many participants. The coordinator organizes and manages the data transfer, while the participants collect the data. Here is an outline of a simple PIO program:

1. Authenticate with HPSS
2. Open a file
3. (Coordinator) hpss_PIOStart()
4. (Coordinator) hpss_PIOExportGrp()
5. Spawn participants
6. (Participants) hpss_PIOImportGrp()
7. (Participants) hpss_PIORegister()
8. (Participants) hpss_PIOFinalize()
9. (Coordinator) hpss_PIOExecute()
10. (Coordinator) hpss_PIOFinalize()

Note the dual role of hpss_PIOFinalize(). It is used to clean up the participants and also to complete the I/O and clean up the coordinator. This is a very simple overview of how to structure a PIO program, but it is accurate in the roles and order in which the functions need to be called to get PIO to function properly.

3.1.8 Trashcans

Trashcans (also called "soft delete") in HPSS are currently enabled or disabled on a system-wide basis. In general, applications should not need to consider whether trashcans are on or off when doing deletion; the functionality is transparent to the application.

When trashcans are enabled, object deletion is considered "soft". Any delete operation will place the object into a trashcan. The objects in this trashcan will then eventually be deleted by the system based upon a system-wide policy. A delete on an object which is already in the trashcan permanently deletes it and frees up the associated space, while moving a file out of a trashcan will keep it from being deleted by the system, as will "undeleting" the file (see hpss_Undelete()).

Moving a file into the trash will not result in a soft deletion; the file must actually be deleted. When a file is deleted into a trashcan it is renamed to append its object ID onto the end of it. For example, myfile.out with object ID 1234 becomes myfile.out.1234. Similarly, moving a trashcan directory does not undelete its contents.

There are two types of trashcans: home directory and fileset. A typical user will have one home directory trashcan and one fileset trashcan per additional fileset. When a user deletes an object in the same fileset as their home directory, the system will move that object into the user's home directory trashcan. This is ~/.Trash. When a user deletes an object in a fileset which differs from their home directory, or if the user has no existing home directory, the object will be placed in the user's fileset trash. This is [fileset root]/.Trash/[user name]. These trashcan directories are automatically created and do not need to be set up by the application. For example, in the root fileset with the hpss user it may be /.Trash/hpss. In a fileset under /sys1 it would be /sys1/.Trash/hpss. Files cannot be soft deleted across filesets.

3.1.8.1 Getting Information on Trashcans

Information about whether trashcanning is on or not, and additional settings, can be obtained by priviledged users using the hpss_GetTrashSettings() API. In general, developers should assume that the trashcan behavior could be changed by the administrator at any time.

Information about a trashcanned object's original location can be obtained using normal attribute functions. See the data structure reference for more infomation.

3.1.8.2 Limitations, Caveats, and Warnings

Currently, operations are not restricted within a trashcan; however it is not advised to develop applications which take any special advantage of this fact as operations on soft-deleted objects may be restricted later.

In its current implementation, HPSS trashcans are a flat directory. This means that while perusing a trashcan, it is impossible to determine the relationships between directories and files in the trashcan because all directories and files reside at the top of the trashcan directory. At this time there is no API which identifies the trashcan containing a deleted file or contents across multiple trashcans.

The Client API caches trashcan directories to reduce unnecessary lookups. In certain cases this may cause the Client API to issue a retry on a delete operation because the cache entry became invalid and had to be refreshed. In some cases a client instance may continue deleting files into a .Trash directory which has been moved elsewhere due to caching. The Client API will not discover that the .Trash has been moved until it is restarted. It is not recommended to move or alter trashcan directories.

3.2 64-bit Arithmetic Library

3.2.1 Purpose

Some HPSS Client APIs require 64-bit fields. The operating system and C compiler on many workstation platforms may not support 64-bit integer operations. As a result, in order to support large integer fields, a set of math libraries has been supplied until 64-bit support is available on all supported vendor platforms.

The Client API has been compiled and tested in 64-bit mode on the four supported platforms. This results in the 64-bit Arithmetic Library components being converted into true 64-bit operations.

3.2.2 Constraints

The following constraints are imposed by the 64-bit arithmetic functions:

- 64-bit signed arithmetic operations are not supported.
- Multiply functions are limited to 64-bit by 32-bit unsigned operations. For example, a 64-bit unsigned integer may be multiplied by a 32-bit unsigned integer. No 64-bit by 64-bit operations are supported for this category of functions.

3.2.3 Libraries

The 64-bit arithmetic functions are included in the **libhpss.a** library.

3.3 Storage Concepts

This section defines key HPSS storage concepts which have a significant impact on the usability of HPSS. Configuration of the HPSS storage objects and policies is the responsibility of your HPSS system administrator.

3.3.1 Class of Service

Class of Service (COS) is an abstraction of storage system characteristics that allows HPSS users to select a particular type of service based on performance, space, and functionality requirements. Each COS describes a desired service in terms of characteristics such as minimum and maximum file size, segment allocation method, transfer rate, access frequency, latency, and valid read or write operations. A file resides in a particular COS and the class is selected when the file is created. Underlying a COS is a storage hierarchy that describes how data for files in that class are to be stored in HPSS.

You specify a COS at file create time by using COS hints and priority structures which are passed to HPSS in the hpss_Open() function. You can later change the COS by using the hpss_FileSetCOS() or hpss_FileSetCOSHandle() functions. Contact your HPSS system administrator to determine the Classes of Service which have been defined. Use the following command to list the defined Classes of Service:

```
lshpss -cos
```

Refer to Chapter 4 of the *HPSS User's Guide* for information on the lshpss command. A class of service is implemented by a storage hierarchy of one to many storage classes. Storage hierarchies and storage classes are not directly visible to the user, but are described below since they map to a Class of Service. The relationship between storage class, storage hierarchy, and COS is shown in Figure 3-1.

3.3.2 Storage Class

An HPSS storage class is used to group storage media together to provide storage with specific characteristics for HPSS data. The attributes associated with a storage class are both physical and logical. Physical media in HPSS are called physical volumes. Physical characteristics associated with physical volumes are the media type, block size, the estimated amount of space on volumes in this class, and how often to write tape marks on the volume (for tape only). Physical media are organized into logical virtual volumes. This allows striping of physical volumes. Some of the logical attributes associated with the storage class are virtual volume block size, stripe width, data transfer rate, latency associated with devices supporting the physical media in this class, and storage segment size (disk only). In addition, the storage class has attributes that associate it with a particular migration policy and purge policy to help in managing the total space in the storage class.

3.3.3 Storage Hierarchy

An HPSS storage hierarchy consists of multiple levels of storage with each level representing a different storage media (i.e., a storage class). Files are moved up and down the storage hierarchy via stage and migrate operations, respectively, based upon storage policy, usage patterns, storage availability, and user request. For example, a storage hierarchy might consist of a fast disk, followed by a fast data transfer and medium storage capacity robot tape system, which in turn is followed by a large data storage capacity, but relatively slow data transfer tape robot system. Files are placed on a particular level in the hierarchy depending on the migration policy and staging operations. Multiple copies of a file may also be specified in the migration policy. If data is duplicated for a file at multiple levels in the hierarchy, the more recent data is at the higher level (lowest level number) in the hierarchy. Each hierarchy level is associated with a single storage class.

3.3.4 File Family

A file family is an attribute of an HPSS file that is used to group a set of files on a common set of tape virtual volumes. Grouping of files only on tape volumes is supported. Families can only be specified by associating a family with a fileset, and creating the file in the fileset. When a file is migrated from disk to tape, it is migrated to a tape virtual volume assigned to the family associated with the file. If no family is associated with the file, the file is migrated to the next available tape not associated with a family (actually to a tape associated with family zero). If no tape virtual volume is associated with the family, a blank tape is reassigned from family zero to the file's family. The family affiliation is preserved when tapes are repacked. Configuring file families is an HPSS System Administrator's function.

3.3.5 User IDs

After the HPSS system is configured, the necessary accounts must be created for HPSS users. Contact your HPSS system administrator to add an account. For the Client API, either a UNIX or Kerberos account must be created. The HPSS system administrator can use the following commands to add a new account.

```
hpssuser -add user -krb
hpssuser -add user -unix
```

3.4 Access Control List API

3.4.1 Purpose

The access control list API is a set of routines for managing access control lists (ACLs). The routines provide a way to convert ACLs from string format into a form suitable for use by the client API routines. They also provide a way to call the client API routines using ACLs and string format, and a way to convert ACLs back from client API format

to string format. In particular, the string conversion routines take care of translating user, group, and realm names into UIDs, GIDs and Realm IDs respectively.

3.4.2 Constraints

None.

3.4.3 Libraries

The access control list APIs are available in the HPSS Client Library, **libhpss.a**.

Chapter 4

Client API Tutorials and Best Practices

This chapter provides guidance on using the HPSS client programming interface (Client API). In addition, short examples and snippets are provided to facilitate new development. Distinctions are made between the characteristics of various API functions which may differ from their typical file system counterparts.

4.1 Client API Concepts

The HPSS Client API provides both POSIX-like interfaces into the HPSS system as well as interfaces specific to managing hierarchical storage. Given the unique nature of the underlying storage system and its metadata, many functions vary from POSIX to some degree, which is why they are described as POSIX-like. HPSS itself is not POSIX conformant, it is POSIX compliant, which means that partial support for POSIX.1 is available, and the features which are supported are documented (in this document). You may assume that any POSIX features not explicitly described in this document are unavailable.

A number of variants of POSIX functions as well as additional functions are provided to allow the development of useful applications which can take advantage of the special features of a hierarchical storage management system, and in some cases these functions have characteristics which may differ from a typical file system implementation. In this section some of these special characteristics will be discussed as well as other more general characteristics of the HPSS Client API, hereafter referred to simply as the Client API or the API.

4.1.1 How the Client API Works

The Client API works by contacting any HPSS Core Server, retrieving the information needed to contact the Core Server associated with the client request, and then issuing requests to that Core Server. Each Core Server, when contacted for location information, reports the location of each Core Server and Gatekeeper server. The Client API detects the correct Core Server for each request based upon either a user-input subsystem id or the subsystem id of the request's object handle. The Core Server handles all requests from the API, with I/O requests being handled by the Mover. The Client connects to the Mover, or the Mover to the client, depending upon the type of I/O, with the Core Server handling the passing of initial connection information. The Core Server does not directly transfer any data. Core Server requests are made via RPC, and data connections to the Mover are made via direct TCP/IP connection.

4.1.2 Client Platforms and Considerations

Consult your HPSS version release notes for proper platform and library prerequisites for the Client API.

4.1.3 Limitations and Known Issues

1. HPSS does not support links which cross junctions.
2. The Client API's default open file limit is 4096. This is a per process limitation, and the Core Server has its own (configurable) limitation.
3. The **HPSS_API_DISABLE_JUNCTIONS** flag must currently be set to '4' in order to work properly.
4. Running the Client API under some memory checking programs such as Valgrind may result in a large number of undefined value errors due to undefined values passed in HPSS RPC interfaces as outputs.
5. Purging a login credential with hpss_PurgeLoginCred() will fail with an error if a login has already been established.

4.1.4 Standard Practices

Many applications choose to spawn a separate process for each Client API I/O operation. Threaded usage of the API is also supported. Having separate processes has a few advantages such as reducing the chance of hitting the open file limitations.

4.1.5 HPSS Levels

Some applications may wish to define certain actions based on the **HPSS_LEVEL** macro. It is a five-digit number built using the major, minor, patch, and build levels of HPSS:

```
(10000*HPSS major level)+(1000*HPSS minor level)+(100*HPSS maintenance level)+HPSS patch level
```

For example, the HPSS 7.4.2 release has **HPSS_LEVEL** of 74200, and HPSS 7.4.1 Patch 3 has **HPSS_LEVEL** of 74103.

4.2 Installation and Configuration

4.2.1 Installation

This section will discuss the necessary steps required to install the Client API. It is important to rebuild your application client binaries following any upgrade as there is no guarantee of backwards compatibility with older versions of HPSS. Failure to do this can result in undefined behavior, failures, or program crashes due to incompatibilities between the HPSS client and server. It is important to thoroughly test your application prior to running it in production with a new version of HPSS. This is especially true when moving up several versions or doing a major upgrade.

4.2.2 Install from RPM

Installing the HPSS Client API from RPM requires identifying the proper client RPM. RPMs are available for all supported architectures and operating systems.

4.2.2.1 Client API Binary Package

The hpss-clnt RPM installs HPSS Client API tool binaries.

```
% rpm -i hpss-clnt-<version>.<platform>.<arch>.rpm
```

Either the hpss-lib or hpss-clnt-devel RPM must be installed as prerequisite for hpss-clnt.

4.2.2.2 Client API Development Package

The hpss-clnt-devel RPM installs HPSS Client API libraries and headers.

```
% rpm -i hpss-clnt-devel-<version>.<platform>.<arch>.rpm
```

4.2.2.3 Client API Mover Protocol Development Package

The hpss-clnt-devel-mvrprot RPM installs HPSS Client API headers for implementing mover protocol.

Warning

> The headers in the mvrprot RPM are considered confidential intellectual property and may not be distributed for any reason without approval from IBM. These headers are available to HPSS licensees upon request.

```
% rpm -i hpss-clnt-devel-mvrprot-<version>.<platform>.<arch>.rpm
```

4.2.3 Install from Source

Normally, any install on a supported platform will use the RPM install process (Install from RPM). The install from source method should only be used by special arrangement.

The Client API source is distributed along with all other HPSS source code. In order to build the Client API on a client machine, it is necessary to extract it and compile it. This is called the client bundle.

4.2.3.1 Extraction from HPSS Source Tree

In order to build the Client API it first must be extracted from the HPSS source tree along with its dependencies and client tools. This is accomplished with a single make command issued from the root of an HPSS source bundle.

```
% make build-clnt BUILD_ROOT=/path/to/client/build
```

This will copy the necessary files to the specified location. Keep in mind that any Makefile customization specific to that source tree will be retained when the files are copied; for instance, the build platform, HPSS root path, library or include locations, etc. These may need to be modified once the bundle is moved to the client environment.

4.2.3.2 Compilation

Compiling the Client API is straightforward, but there are several Makefile options which may need to be modified depending upon the environment. The primary Makefile is Makefile.macros, and it has several important flags and values which may need to be changed depending on your desired configuration.

```
BUILD_PLATFORM = LINUX
```

Set the BUILD_PLATFORM to the O/S the client will run on. The legal values are AIX, SOLARIS, and LINUX. This value in turn sets a number of variables based upon defaults found in the Makefile.macros.[OS] Makefiles. The system-specific defaults are not typically changed, but can control things such as library locations and binaries for use in the Makefiles.

```
BUILD_TOP_ROOT = /opt/hpss
```

This is the HPSS source directory. It is highly recommended that this be a link to some other area so that multiple source code directories can coexist and be maintained without confusion, and if necessary the client source can be backed up to an older version.

```
% make clnt
```

This builds the Client API, its dependencies, and some Client API tools (most notably **scrub**). You should at a minimum see the HPSS client libraries in **$HPSS_ROOT/lib**: **libhpss.so** and **libhpsscs.so**. Depending upon your security settings you will see **libhpssunixauth.so** and possibly **libhpsskrb5auth.so** as well.

4.2.4 Using pkg-config To Compile

Both the source and RPM installs of the Client API include a package config file. This file can be used to assist in compiling your application with the HPSS Client API by providing a set of compile flags and library dependencies which are commonly required for HPSS applications. The pkgconfig file is located in: **${HPSS_ROOT}/lib/pkgconfig/hpss.pc**.

Note that if the install path is not the final path for the libraries and headers, the 'prefix' path in hpss.pc can be modified to allow the continued use of pkgconfig.

Note also that the pkgconfig file is only valid for the platform associated with the RPM. In order to use the Client API on a different platform, architecture, operating system, or bit-width, the RPM for that specific environment must be obtained and installed.

4.2.5 Testing

There are a several tools and utilities which are either included with the HPSS software package or available as support tools which can test the Client API. The first and foremost Client API test utility is **scrub**. **scrub** provides a simple shell interface which can be used to test connectivity, namespace operations, file I/O, and much more. The **hpsssum** utility is also included in the client build, and can be used to read existing files out of HPSS or generate / modify file checksum metadata stored with the UDAs feature.

4.2.6 Configuration

Client API settings can be configured in three ways: environment variables, at run-time, and through other HPSS and system files. All HPSS environment variables and the most common alternate configurations will be described.

4.2.7 Client API Environment Variables

Each of these environment variables is defined in hpss_env_defs.h, which is appended onto the HPSS Management Guide, and are also described in other reference material such as the *HPSS Programmer's Reference*. For the sake of completeness this guide will provide a few short examples of practical applications of each of these environment variables.

HPSS_API_HOSTNAME

This is the data hostname that the API will use when supplying an address to the Mover when initiating a transfer. Typically this is used to specify an alternate hostname which maps to a high speed interconnect, while the host will continue to use a hostname mapped to a slower control network when communicating with the Core Server. It is important to note the Mover must be able to connect to this alternate interface.

HPSS_API_DEBUG / HPSS_API_DEBUG_PATH

The **HPSS_API_DEBUG** and **HPSS_API_DEBUG_PATH** environment variables are described in some detail in the Troubleshooting section; they control the level of detail and output path of API logging.

HPSS_API_MAX_CONN This value is currently unused; the total maximum connections across all Client API applications can be configured using the Core Server.

HPSS_API_MAX_OPEN The Client API instance open file limit is configurable. The default value is 4096 open files. Applications may increase this by setting an environment variable, **HPSS_API_MAX_OPEN**. This may be set either in the env.conf or in the application environment. Hitting this limit will cause further opens to fail. The lower limit for the configured Client API maximum open files is 1; the upper limit for the configured Client API maximum open files is (2^31-1).

HPSS_API_RETRIES

This value limits the total number of retries which will be done by the Client API where the class of the error is "Retryable". "Retryable" errors will be retried right away. Retries can be turned off by setting this to '-1'. The following Core Server errors are considered retryable:

- HPSS_EINPROGRESS, -EINPROGRESS
- HPSS_ETIMEDOUT
- HPSS_EBUSY
- HPSS_ENOTREADY

HPSS_API_BUSY_DELAY

This value is the time (in seconds) to wait between retries when the class of error is "delay retryable". The following Core Server errors are considered delay retryable:

- HPSS_ENOCONN
- HPSS_BUSY_RETRY (only applies when manipulating a bitfile, e.g., open, close, stage)
- BFS_STAGE_IN_PROGRESS (if API_RetryStageInp is turned on)

HPSS_API_TOTAL_DELAY

This value is the total time (in seconds) across all retries that the request may be retried.

HPSS_API_LIMITED_RETRIES

This value is the number of times to retry when the class of error is "limited retryable". This class of retryable operations will be retried immediately with no delay a limited number of times. The following Core Server errors are considered limited retryable:

- HPSS_EPIPE
- HPSS_ECONN
- HPSS_EBADCONN

HPSS_API_DMAP_WRITE_UPDATES

This environment variable is defunct and has been removed from the API.

- HPSS_API_REUSE_CONNECTIONS

If this option is enabled the Client API will request that storage resources be held until the connection is closed, and reuse data connections.

HPSS_API_USE_PORT_RANGE

This value causes the Client API to request that the Mover connect back using the port range specified in the mover port range configuration. However this value has no meaning when PDATA PUSH is used because the Client connects to the Mover rather than the Mover connecting back to the Client.

HPSS_API_RETRY_STAGE_INP

This value, when nonzero, will interpret the BFS_STAGE_IN_PROGRESS error as retryable. This translates into errors caused by staging being retried rather than immediately returning with an error. This is on by default.

HPSS_API_DISABLE_CROSS_REALM

This is used to reject requests to initialize the Client API when one of the realms is not local. Cross-realm support is a special configuration of HPSS which allows HPSS installations to communicate, and is not generally useful for most customers.

HPSS_API_DISABLE_JUNCTIONS

This option will disable the crossing of junctions by the Client API. This essentially limits any client applications to the root fileset and a single subsystem, and can be useful in restricting access.

HPSS_API_SITE_NAME

This value is not used by the Client API, but may be useful for client applications which connect to multiple HPSS systems.

HPSS_API_AUTHN_MECH

This option will set the default authentication mechanism used by the Client API. Applications can specify a different mechanism at run time. In sites which use both Unix and Kerberos this can be useful in defaulting a certain set of clients to one authentication mechanism or the other.

HPSS_API_RPC_PROT_LEVEL

This value controls the RPC protection level of the API connections made to the Core Servers. The effects of these values are documented in the HPSS Installation Guide; the allowed values are: connect packet, packet integrity, and packet privacy.

HPSS_API_SAN3P

This environment variable determines whether the Client API will use SAN3P. In certain circumstances it may be beneficial to turn this off and use the network interface for data transfer instead; for example, the SAN interface is down or a certain client may not need access to the disk resources on the SAN.

HPSS_API_TRANSFER_TYPE

This is the default transfer type which will be used by the API. The three values are TCP, MVRSELECT, and IPI. IPI is defunct, TCP implicitly disables SAN3P, and MVRSELECT allows SAN3P (if available).

HPSS_API_OBJ_BUF_LIMIT

This represents the maximum number of objects which can be returned by a single hpss_UserAttrGetObjs() or hpss_UserAttrReadObjs() operation. The Core Server will adhere to this limit when handling the UDAs calls which retrieve Object Ids. This is meant to be an administrative limitation placed on all API clients at the Core Server layer; modifying this value on Client machines has no effect.

HPSS_API_XML_BUF_LIMIT

This represents the maximum number of XML strings which can be returned by a single hpss_UserAttrGetXML() or hpss_UserAttrReadXML() operation. The Core Server will adhere to this limit when handling the UDAs calls which retrieve XML strings. This is meant to be an administrative limitation placed on all API clients at the Core Server

layer; modifying this value on Client machines has no effect.

HPSS_API_XMLOBJ_BUF_LIMIT

This represents the maximum number of objects which can be returned by a single hpss_UserAttrGetXMLObj() or hpss_UserAttrReadXMLObj() operation. The Core Server will adhere to this limit when handling the UDAs calls which retrieve Object Id and XML string pairs. This is meant to be an administrative limitation placed on all API clients at the Core Server layer; modifying this value on Client machines has no effect.

HPSS_API_XMLSIZE_LIMIT

This represents the maximum size of a single XML string which will be returned by an hpss_UserAttrGetXML(), hpss_UserAttrGetXMLObj(), hpss_UserAttrReadXML(), or hpss_UserAttrReadXMLObj() request. Any operation requesting strings longer than this limit will be rejected. This is meant to limit memory consumption by the Core Server as it is possible to request XML strings up to 2 GB (provided appropriate infrastructure exists).

HPSS_API_XMLREQUEST_LIMIT

This represents the maximum total size of a single request returning XML strings. Any operation with a total request size longer than this limit will be rejected. The total request size is calculated as (size of request strings $*$ number of request strings). This is meant to limit memory consumption by the Core Server as it is possible to request multiple XML strings, each having a very large size.

4.2.8 Client API Runtime Configuration

Most of the environment variables enumerated above correspond to a configurable field or flag within the Client API Configuration data structure (see hpss_api.h). These fields and flags can be retrieved from the API by using the hpss_GetConfiguration() function, and can be set using the hpss_SetConfiguration() function. This allows a more finely grained and manageable configuration that does not necessarily rely upon external configuration files which may be shared between different applications.

4.2.9 Other System Configuration

Some applications have their own configuration files which change Client API behavior when used within that application. For example, the HPSS.conf file is used by FTP to allow user configuration of many parameters, including parameters related to the Client API.

4.3 Tuning and Troubleshooting

4.3.1 Tuning

Tuning is as much art as science, and in general almost no specific tuning parameter is a best fit for everything. This section describes what kinds of things impact the performance of the Client API.

4.3.2 Expectations

Some technologies within HPSS may have some surprising characteristics for new users. Application programmers should be aware of some of the characteristics of these technologies so that they can write applications that utilize the strengths of the underlying technology.

4.3.2.1 Tape

Tape is a sequential I/O technology with relatively high seek times to get to a random position on tape. Tape has the additional characteristic of not being "always on" like a disk - a tape cartridge must first be mounted into a tape drive before it can be read, and also a tape drive has a start-up time before it can reach its maximum I/O rate.

HPSS systems make heavy use of tape storage for long-term archival. It is important to recognize when constructing an application that data on tape may not be readily available due to factors like location of the cartridge and availability of tape drives - a read request from tape could take minutes or hours between the time the request is generated and the time the first byte is received. Additionally, assuming that an application is reading directly from tape, a short read of one part of a file followed by a short read of another part of a file may cause a noticeable delay

4.3.2.2 Metadata

Metadata is data about the data. It may describe where it's located, how it's laid out on media, who can access it, etc. In HPSS, metadata is located in a relational database. Some of the characteristics of a relational database apply to the access and retrieval of HPSS metadata. For example, a simple directory listing typically translates into an SQL select, and multiple reads from the database. Using this knowledge, it should come as no surprise to learn that a short delay could be expected before the first results appear in a very large directory. After all, the relational database will fetch all the results when the SQL is run, and the time required to do that is dependent upon the number of entries.

4.3.3 Testing Procedures

It is important to test that the Client API deployment was successful. HPSS support maintains a repository of example programs which can be used to test functionality, and the **scrub** tool can also be used to test functionality. It is appropriate to test both namespace and I/O operations. When testing client applications, consider how the HPSS APIs you are using are impacting performance. Doing this frequently can help identify possible performance issues early. Perhaps there's a better API for the task.

4.3.4 Tuning Concepts

The important information to know are what you're trying to optimize, and some common things that can be done to optimize those things. Depending upon specific deployments and specific hardware the specific steps and chokepoints may be completely different; however the optimization points are still the same. The Client API will perform best when:

1. A reliable connection exists between the machines running the Client API, the Movers, and the Core Server machines. Avoid public WANs which can drop long-running connections.
2. The Core Server is configured in anticipation of the planned client and system administrative load. The server Maximum Requests size should be larger than the anticipated number of clients to accommodate administrative programs, such as various tools including **repack**.
3. High speed interconnects are used between the Client API and Mover. The API data hostname should be configured properly to take advantage of high speed interconnections. For SAN3P, it's important to verify that the SAN3P disks are usable by the client.
4. Storage class definitions are appropriate for the data being transferred. Misconfigured storage classes can lead to network and storage inefficiencies due to large numbers of protocol messages and transactions. I/O performance is sensitive to storage segment sizes due to an HPSS internal limit of 32 descriptors in a request at a time.
5. Parallelism. HPSS is meant to scale very well at the data layer. Multiple client machines and many client processes is considered normal. Application developers should expect to use multiple processes in order to get maximum performance.

6. Software Tuning. Poorly written software can have a significant detrimental impact to overall application performance. Some of the most common issues are inappropriate buffer sizes, misuse of APIs, or making assumptions about performance of various HPSS APIs based upon past experience programming for file systems. For example, simple namespace operations such as renaming or creating a hardlink is more expensive in HPSS than a typical file system.

4.3.5 Troubleshooting

HPSS and the Client API provide various ways of troubleshooting Client API applications. Among these are configurable logging subsystems on the HPSS servers and within the Client API as well as system files for the security subsystem. This section will describe various methods and issues that may arise during troubleshooting.

4.3.5.1 Debug and Trace Levels

HPSS has eight types of logging which can be configured per server. When doing API development it is often useful to turn REQUEST and TRACE logging on for the Core Server in order to more easily identify where certain errors come from. Additional logging on other servers (for example, the Mover) may be appropriate depending upon the nature of the error. The Client API has three types of debugging which can be controlled through the **HPSS_API_DEBUG** environment variable: ERROR, REQUEST, and TRACE. These are controlled by an octal value in the API Logs section below. Within the context of the Client API, ERROR logs are due to any error code being returned by the Core Server - this can include so-called 'informative error codes' which do not result in operational failure. REQUEST logs are application requests to the Client API, and TRACE level will log information during normal operations describing API activities or state values. Other logs mentioned typically have no configurable logging level; that is, they are either on or off.

4.3.5.2 API Logs

The HPSS API logs are a very useful first step in troubleshooting HPSS Client API problems. They can be turned on within the shell by setting the **HPSS_API_DEBUG** environment variable, by the application (using hpss_SetAPILogLevel() / hpss_SetAPILogPath()), or by using the HPSS API logging resource file.

Below is a table which indicates which logging types will be enabled based upon the value of **HPSS_API_DEBUG**. This debug output will go to stdout by default, and can be redirected by specifying an alternate path using the **HPSS_API_DEBUG_PATH** environment variable.

Value	Trace	Request	Error
0	Off	Off	Off
1	Off	Off	On
2	Off	On	Off
3	Off	On	On
4	On	Off	Off
5	On	Off	On
6	On	On	Off
7	On	On	On

Consult the Client API function references for hpss_SetAPILogLevel() and hpss_SetAPILogPath() for details on setting the log level and path programmatically. These log level / path updates can be done while the application is running post-initialization.

The HPSS API Resource file allows system-level manipulation of HPSS Client API logging, overriding application-supplied log parameters. This file must be created as:

${HPSS_PATH_TMP}/hpss.api.resource (e.g. /var/hpss/tmp/hpss.api.resource)

Once the file exists and has valid log level and path information, all Client API logging will redirect there after the applications detect the change (within 30 seconds). If the application cannot read the resource file due to file or directory permissions, then the resource file cannot be used.

The format of the file is:

```
<log level> <log path>
```

The log path has a limit of 1023 characters and the log level is the same as the table above. The new logging will persist until the resource file is removed, after which time the log levels and paths will return to their original levels and locations. Remember that this resource file controls output for the entire sytem on which it is placed, so logs can become large very quickly. The resource file cannot be disabled; it must be removed in order for resource-based log levels to be removed.

It is worth noting that some error log entries within the API may be expected and may not constitute a hard error. For example, on initialization the API will attempt to place the authenticated user in their home directory. If that home directory does not exist you may see an error log even though the application will continue to function. Other error messages are logged but serve as informational errors from the Core Server back to the API, and provide useful information for understanding the program flow. An example of this is an API error called **HPSS_EMOREPATH** which will be returned and logged whenever a junction is crossed. In general the API debug log should only be turned on when troubleshooting a specific problem; the logging mechanism has no roll-over mechanism and so the log file can grow without bound.

4.3.5.3 HPSS Logs

Sometimes the information provided in the API log is not enough to understand what went wrong. At that point it is useful to start looking through the HPSS log messages. It is possible to trace a request from the Client API through the HPSS logs using the Request Id. This Request Id is presented in the API log as a hex value, for example:

```
trc:02/27/2012 14:22:18.389320[0x5f500001]: entering API_TraversePath: Path=/,Handle=0xf7f5fac0
```

0x5f500001 is the Request Id as a hex string; the decimal representation which will be logged by the Core Server is 1599078401. If you have an API log which shows an error being returned from the Core Server you can easily find the HPSS logs which correspond to it by converting the hex string and finding that Request Id (1599078401) in the HPSS log. The Request Ids generated by the API are very likely, but not guaranteed, to be unique within a time window of hours or even days, depending upon the level of system activity.

Certain errors, such as authentication errors, are not always logged by the HPSS server, or the details needed to resolve the problem aren't logged in all cases. In order to get additional information about authentication and authorization related issues, use the /var/hpss/tmp/gasapi.diag and /var/hpss/tmp/secapi.diag files. These files can be created on the client machine or the Core Server to begin logging security and authentication information, and they will log information about every authentication request. These files do not timestamp or trim themselves and so they should be turned on just long enough to debug the issue. In all cases it's necessary to stop each process which was using the log files before the space is truly freed - for example all clients must be stopped on the client side, and all servers must be stopped on the Core Server side. This can cause service disruption and so its typically desirable to plan the use of these log files around a window where a small outage would be acceptable.

4.3.5.4 Crash Dumps

Occasionally you may get an application crash due to a software or hardware error. In such cases, you should use the logs and the generated core file to get an understanding of what caused the crash. There's nothing special about debugging the Client API in a core dump. If the problem appears to originate within the Client API, contact HPSS support with the core dump trace and a description of what the application was doing so that the issue can be analyzed.

4.4 Client API Uses

4.4.1 Overview and Applications

The Client API is the most general interface into HPSS. Every client interface packaged with HPSS (PFTP, VFS) and many other interfaces use the Client API as a lower layer.

PFTP and VFS have very different deployment models for the Client API. PFTP runs a daemon on the HPSS Core Server, and requests from PFTP clients funnel through that daemon, which in turn uses the Client API. This allows PFTP to be deployed with a minimum of prerequisite software and configuration data. VFS, on the other hand, deploys the Client API on every VFS client machine, which includes all of its libraries and configuration files.

4.5 Tutorials

This section provides examples and discussion of the most-used APIs including an in-depth tutorial showing a complete view of creating an API program and compiling it from scratch. Additional tutorials will discuss small snippets centered around single functions or logical groups of functions.

4.5.1 Overview

The following tutorial will provide a thorough examination of the creation of a Client API program from start to finish. Each additional tutorial focuses on a small topic such as creating a file and present a snippet of code which accomplishes that task. Those tutorials also discuss related issues and common variations on the task. There are over 200 APIs in the HPSS Client API, and not all of them can be covered in this level of detail, therefore only the APIs most likely to be used are covered by this document. The Client Interfaces section of this document does provide a basic description of every API, its parameters, and its expected return codes. These tutorials should not be misinterpreted as a replacement for the API references for understanding specific Client API functions.

4.5.2 A Short Example: Current Working Directory

Assumptions: This tutorial (and other tutorials in this section) should be run on the Core Server as it uses HPSS system user IDs to log in. It may be run on a properly configured client system if the HPSS system account is created there, or if the program is modified to use an account that is available on both the Core Server and the client host. This tutorial assumes that a directory with the compiled HPSS client source exists, and we'll refer to it as **HPSS_ROOT**. It will also assume, in the case a client machine is used, that the prerequisite configuration files have been copied over and the Client API has been successfully tested for connectivity and working configuration with the Core Server and the Mover using **scrub** or a similar utility. Consider the following simple program:

```
00
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <hpss_errno.h>
#include <hpss_api.h>
#include <hpss_Getenv.h>
#include <hpss_limits.h>

int
main(int argc, char** argv)
{
 int rc;
 char cwd_path[HPSS_MAX_PATH_NAME];

 char keytab_file[HPSS_MAX_PATH_NAME] = "/var/hpss/etc/hpss.unix.keytab";

 unsigned char* env, *keytab;
 hpss_authn_mech_t mech;
 int i;

 env = hpss_Getenv("HPSS_PRIMARY_AUTHN_MECH");
```

```
 if(env != NULL) {
 if(strcasecmp(env, "UNIX") == 0) {
 mech=hpss_authn_mech_unix;
 env = (char*)hpss_Getenv("HPSS_UNIX_KEYTAB_FILE");
 strcpy(keytab_file, env);
 }
 else if(strcasecmp(env,"KRB5") == 0) {
 mech=hpss_authn_mech_krb5;
 env = (char*)hpss_Getenv("HPSS_KRB5_KEYTAB_FILE");
 strcpy(keytab_file, env);
 }
 else {
 printf("Unknown authentication mechanism.  Defaulting to Unix.\n");
 mech=hpss_authn_mech_unix;
 }

 // remove auth_keytab prefix
 for(i = 0; i < strlen(keytab_file); i++)
 if(keytab_file[i] == ':')
 break;
 i++;
 keytab=keytab_file+i;

 }
 else {
 printf("Primary authentication not defined.\n");
 exit(1);
 }



 if((rc = hpss_SetLoginCred("hpssssm",
 mech,
 hpss_rpc_cred_client,
 hpss_rpc_auth_type_keytab,
 keytab)) < 0)
 {
 printf("could not authenticate.\n");
 exit(1);
 }


 rc = hpss_Getcwd(cwd_path, HPSS_MAX_PATH_NAME);

 printf("Path is %s\n", cwd_path);

 return 0;
}
```

The following gcc command will successfully compile this Client API program on Linux:

```
gcc `pkg-config --cflags --libs /opt/hpss.75/lib/hpss.pc` -c hpss_test_program.c
gcc `pkg-config --cflags --libs /opt/hpss.75/lib/hpss.pc` hpss_test_program.o -o ../bin/hpss_test_program
```

This is done in a two step process (compile and link). For simple programs the two can easily be combined in gcc. HPSS provides a pkgconfig file for the Client API which is based upon the environment the Client API was compiled in. By default it is in the $HPSS_ROOT/lib folder as 'hpss.pc'. It includes a number of useful pieces of information such as the HPSS version, the compile flags, and the libraries required.

Users of the client bundle (client source) will generate a new pkgconfig file when the tree is compiled. Users of the build-clnt-devel bundle (binaries and headers only) will receive a pkgconfig file with the bundle.

The pkgconfig file can be customized to some extent by the developer as necessary. pkgconfig files are not guaranteed to be compatible across HPSS releases.

Now, let's discuss the headers in the example above:

```
00
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <hpss_errno.h>
#include <hpss_api.h>
#include <hpss_Getenv.h>
#include <hpss_limits.h>
```

The HPSS error codes are contained in hpss_errno.h. HPSS error codes are defined as negative numbers, and try to conform to their errno equivalent where possible. For example, HPSS error code HPSS_EIO is -5; Linux error code EIO is 5. The hpss_api.h header defines the data structures and function prototypes for the HPSS Client

API. The hpss_Getenv.h defines the hpss_Getenv() function, which is used to retrieve HPSS environment variables from the default list, the override file, or the environment. Finally, hpss_limits.h defines various limits within HPSS including the max path size.

```
00
int
main(int argc, char** argv)
{
 int rc;
 char cwd_path[HPSS_MAX_PATH_NAME];

 char keytab_file[1024] = "/var/hpss/etc/hpss.unix.keytab";

 unsigned char* env, *keytab;
 hpss_authn_mech_t mech;
 int i;
```

This section defines the main function and creates some local variables which will be used later. It is worth noting that HPSS_MAX_PATH_NAME is used here to hold the current working directory path and the keytab file name - using HPSS_MAX_PATH_NAME for HPSS paths is generally a good idea - and a variable of hpss_authn_mech_t type will be used to hold the authentication mechanism.

```
00
env = hpss_Getenv("HPSS_PRIMARY_AUTHN_MECH");
if(env != NULL) {
 if(strcasecmp(env, "UNIX") == 0) {
 mech=hpss_authn_mech_unix;
 env = (char*)hpss_Getenv("HPSS_UNIX_KEYTAB_FILE");
 strcpy(keytab_file, env);
 }
 else if(strcasecmp(env,"KRB5") == 0) {
 mech=hpss_authn_mech_krb5;
 env = (char*)hpss_Getenv("HPSS_KRB5_KEYTAB_FILE");
 strcpy(keytab_file, env);
 }
 else {
 printf("Unknown authentication mechanism.  Defaulting to Unix.\n");
 mech=hpss_authn_mech_unix;
 }
 // remove auth_keytab prefix
 for(i = 0; i < strlen(keytab_file); i++)
 if(keytab_file[i] == ':')
 break;
 i++;
 keytab=keytab_file+i;
}
else {
 printf("Primary authentication not defined.\n");
 exit(1);
}
```

This section obtains the default authentication mechanism and the associated keytab. This is useful because we're using keytab authentication for this program. It's also possible to use a password, and also possible to create your own HPSS keytab file with non-HPSS system users. hpss_Getenv() pulls its environment variables from the environment , the overrides file, and finally a set of default defined values, and will prefer them in that order - that is, if a variable is set in the environment, hpss_Getenv() will use that, and then check the overrides file, and finally the defaults.

```
00
if((rc = hpss_SetLoginCred("hpssssm",
 mech,
 hpss_rpc_cred_client,
 hpss_rpc_auth_type_keytab,
 keytab)) < 0)
{
 printf("could not authenticate.\n");
 exit(1);
}
```

Now we finally get to do something. The hpss_SetLoginCred() function logs into HPSS and creates a security credential with the information we provide. The user "hpssssm" can be useful for testing purposes, or you can substitute your own user who has an entry in the appropriate keytab file. This user should exist on the system that the program is running on as well as the HPSS Core Server. Using a user which already exists on the test system is done to make this tutorial shorter and simpler, but in general your application should use your own user rather than an HPSS system user name. Logging in with a non-system user name is done in the same way, although you may have to provide a password or a custom keytab rather than the HPSS default keytab.

```
00
 rc = hpss_Getcwd(cwd_path, HPSS_MAX_PATH_NAME);
 printf("Path is %s\n", cwd_path);
 return 0;
```

```
}
```

Since we're now logged in it's possible to use the Client API. Prior to this the application would have seen errors such as HPSS_EPERM (-1) when attempting to access HPSS through the Client API. Our simple tutorial application merely calls hpss_Getcwd(), which returns the current working directory. In this case your application may print "/" or "/home/hpssssm" depending upon what the hpssssm user's home directory is set to. If you change the program and run with a different user you'll likely see the current working directory as that user's home directory, unless it has not been created. Client API initialization will happen the first time the Client API is called after logging in, and creates the settings the API will run under (for instance, the environment variables described in the Client API Environment Variables section).

The remainder of the tutorials within this document will contain only the information necessary to understand the usage of each API rather than a full program. This tutorial program can be used as a base from which to test and manipulate these other snippets. The code required to log into HPSS described above will be moved to a function called "login" for brevity.

4.5.3 Namespace Metadata

The HPSS namespace presents a way to navigate and organize files within directories. Every file within HPSS has an entry in the namespace. In order to create a file, it must be have namespace metadata associated with it. The following tutorials focus on creating, retrieving, and manipulating namespace metadata. The characteristics of the namespace are such that a working mover is not required in order to satisfy its operations; all of the namespace metadata resides on the Core Server.

4.5.3.1 Create

The following snippet will create a new HPSS file, assuming that appropriate permissions exist. There are variants of this function such as hpss_Create(), hpss_Creat(), and others which provide alternative methods for creating and optionally opening an HPSS file.

```
00
int hfd;
hpss_cos_hints_t hints_in, hints_out;
hpss_cos_priorities_t hints_pri;
char *hfile;
login();
memset(&hints_in, 0x0, sizeof(hints_in));
memset(&hints_out, 0x0, sizeof(hints_out));
memset(&hints_pri, 0x0, sizeof(hints_pri));
hfile=argv[1];
hfd = hpss_Open(hfile, O_CREAT, 0777,
 &hints_in, &hints_pri, &hints_out);
if (hfd < 0)
{
 fprintf(stderr, "Could not open/create:  %s, error %d\n", hfile, hfd);
 exit(1);
}
hpss_Close(hfd);
```

The hints are an important addition to the open API. They allow the caller to specify several criteria and requirements for the HPSS Class of Service (COS) that will be used to store the new file's data. One common usage is to simply require a specific Class of Service be used, and another may require a certain File Family be used. The resulting COSId will be presented via the hints_out parameter in the example above.

There are a few other notable features of hpss_Open(). One of them is that the file remains open until it is closed, and the HPSS Client API has a file table limit defined by **HPSS_API_MAX_OPEN** open files at a single time. The default is 4096 open files. Across multiple instances, each instance could have up to **HPSS_API_MAX_OPEN** files open with an appropriately configured HPSS Core Server. hpss_Open() also allows the mode permissions to be supplied, however it is important to note that when testing, the umask will be applied to these mode bits; use hpss_Umask() to change the umask settings.

4.5.3.2 Rename

The following snippet will rename an existing HPSS file, assuming that appropriate permissions exist for both the old and new locations. There are other ways to do this operation, for example by creating and removing hard links.

```
00
int rc;
char *oldfile, *newfile;
login();
oldfile=argv[1];
newfile=argv[2];
rc = hpss_Rename(oldfile, newfile);
if (rc < 0)
{
 fprintf(stderr, "Could not rename:  %s, error %d\n", oldfile, rc);
 exit(1);
}
```

The rename operation is very simple; one thing to watch out for troubleshooting rename issues is to be sure that both the old and new areas should be checked for proper conditions when a rename fails.

4.5.3.3 Unlink

The following snippet will demonstrate the hpss_Unlink() function. The hpss_Unlink() function is similar to the unlink(2) function in that it will always result in a namespace entry being removed, but may or may not actually remove any data. For example, regular files will have their data removed, but a hardlink may not remove any data if additional hardlinks to that bitfile exist. Unlinks on a directory will succeed only if the directory is empty - to recursively delete directory contents, it's necessary to traverse the directory subtrees and empty the directories before deleting them. Unlinks on symbolic links will likewise not delete any file data.

```
00
int rc;
char *file;
login();
file=argv[1];
rc = hpss_Unlink(file);
if (rc < 0)
{
 fprintf(stderr, "Could not unlink:  %s, error %d\n", file, rc);
 exit(1);
}
```

Unlinks in HPSS are performed in two stages where in the first stage the namespace object is removed, and in the second the bitfile is removed. The bitfile is placed on a list of files to be removed and will be cleaned up as allowed by the system. The upshot from this is if you are deleting a large number of files, or a number of heavily used files, you may not see space being freed for some time even though the namespace entries have been removed.

4.5.3.4 Get Attributes

This snippet will retrieve file attributes for a namespace entry. This can be used, for example, to retrieve the number of entries in a directory, the size of a file, the last time the file was written to, and more:

```
00
int rc;
char *file;
hpss_fileattr_t attrs;
login();
memset(&attrs, 0x0, sizeof(attrs));
file=argv[1];
rc = hpss_FileGetAttributes(file, &attrs);
if (rc < 0)
{
 fprintf(stderr, "Could not get attributes:  %s, error %d\n", file, rc);
 exit(1);
}
printf("File name:%-20s\n", file);
printf("COS:%-20d\n", attrs.Attrs.COSId);
printf("EntryCount:%-20d\n", attrs.Attrs.EntryCount);
printf("Family:%-20d\n", attrs.Attrs.FamilyId);
```

The great thing about the hpss_FileGetAttributes() function is that it's relatively fast and in many cases its results could be returned from cache rather than being read from the database. Its downsides are mostly that there is a lot

more information which can be retrieved and can be extremely useful - that information is the domain of the HPSS "extended attributes", not to be confused with the concept of linux extended attributes.

4.5.3.5 Get Extended Attributes

The HPSS extended attributes contain additional bitfile and storage server information about the location of the file data within the HPSS storage hierarchy and storage class. This information can be useful in making decisions in user applications because it's possible to tell, for example, whether the I/O request will require a tape mount.

```
00
int rc;
char *file;
hpss_xfileattr_t attrs;
char buf[255];
int i, j, k;
login();
memset(&attrs, 0x0, sizeof(attrs));
file=argv[1];
rc = hpss_FileGetXAttributes(file, API_GET_STATS_FOR_ALL_LEVELS, 0, &attrs);
if (rc < 0)
{
 fprintf(stderr, "Could not get attributes:  %s, error %d\n", file, rc);
 exit(1);
}
printf("File name:%-20s\n", file);
printf("COS:%-20d\n", attrs.Attrs.COSId);
printf("EntryCount:%-20d\n", attrs.Attrs.EntryCount);
printf("Family:%-20d\n", attrs.Attrs.FamilyId);
for(i=0;i<HPSS_MAX_STORAGE_LEVELS;i++)
{
 printf("Level %d Stripe Width:  %d\n", i, attrs.SCAttrib[i].StripeWidth);
 printf("Level %d Flags:  %d\n", i, attrs.SCAttrib[i].Flags);
 printf("Level %d VVs:  %d\n", i, attrs.SCAttrib[i].NumberOfVVs);
 for(j=0;j<attrs.SCAttrib[i].NumberOfVVs;j++)
 {
 printf("VV %d RelativePosition:  %d\n", j, attrs.SCAttrib[i].VVAttrib[j].RelPosition);
 printf("VV %d BytesonVV: %s\n", j, u64tostr_r(attrs.SCAttrib[i].VVAttrib[j].BytesOnVV, buf));
 if (attrs.SCAttrib[i].VVAttrib[j].PVList != NULL)
 {
 for(k=0; k < attrs.SCAttrib[i].VVAttrib[j].PVList->List.List_len; k++)
 {
 printf("PV Name %d:  %s\n", k, attrs.SCAttrib[i].VVAttrib[j].PVList->List.List_val[k].Name);
 }
 free(attrs.SCAttrib[i].VVAttrib[j].PVList);
 }
 }
}
```

This snippet will present a few of the fields present in each level of detail within the extended attributes. The main concepts here are that there are three lists to go through, one for the levels of an HPSS hierarchy, one for the VV metadata within that hierarchy, and finally one for the list of PVs associated with those VVs. Using the extended attributes you can determine, for example, if the requested data might be on the second level of the hierarchy, but also that it's on disk, or maybe that the tape the data is on is shelved (not in the robotic library).

Of course getting all of this information can take some time. Most benchmarks place getting the extended attributes at anywhere from 100% to 400% slower than a simple file attributes retrieval. The performance can be improved if some scenarios can be removed and the requested information can be more targeted - the API allows for a specific level to be retrieved, for example. And of course, the PVList expects to be freed, so don't forget that.

4.5.3.6 Directory Listing

At a certain level, listing a directory just means getting the attributes for every namespace object in that directory. Within HPSS it is actually a fairly complex operation because an individual directory listing might have its ongoing database handle and results. This provides a consistent view of the chosen directory at a moment in time, and allows the request to be split across several requests from the Client API to the Name Server portion of the HPSS Core Server. Directories can also be extremely large and though directory results are normally sorted, in HPSS the default is not to sort them as that can require a lot of extra time which may be wasted if the user doesn't want or need a sorted listing. There is an option to retrieve a sorted listing for those who desire such output.

4.5.3.7 Short Listing

A short listing is relatively fast and simple to implement. The following program will list the objects contained in the provided directory name:

```
00
int rc;
char *file;
hpss_dirent_t dirs;
int dirfd;
int i, j, k;
char* directory;
login();
memset(&dirs, 0x0, sizeof(dirs));
directory=argv[1];
dirfd = hpss_Opendir(directory);
if(dirfd < 0)
{
 fprintf(stderr, "Could not open directory %s, rc=%d\n", directory, dirfd);
 exit(1);
}
while(1)
{
 rc = hpss_Readdir(dirfd, &dirs);
 if (rc < 0)
 {
 fprintf(stderr, "Could not get attributes:  %s, error %d\n", file, rc);
 (void) hpss_Closedir(dirfd);
 exit(1);
 }
 if(dirs.d_namelen == 0)
 break;
 printf("%s\n", dirs.d_name);
}
rc = hpss_Closedir(dirfd);
if(rc != 0)
{
 fprintf(stderr, "Could not close directory %d, rc=%d\n", dirfd, rc);
 exit(1);
}
```

Really not a lot to write home about here. Initially the directory is opened, and then we use the file descriptor returned by hpss_Opendir() to read the directory entries one by one. Once we see an entry with no name we'll stop reading them and close the directory - closing the directory will free up memory that would otherwise be left open until the open entry became stale as well as the file table entry associated with the open directory. Looking at the hpss_dirent_t structure, you may notice that there's not a lot of information here. Well, they don't call it a short listing for nothing. Next..

4.5.3.8 Long Listing

If you just read the previous section, "Short Listing", then this will look familiar to you. This is essentially the same program, but each entry in the directory has a hpss_GetAttrHandle() called on it to get its basic attributes.

```
00
int rc;
char *file;
hpss_dirent_t dirs;
hpss_vattr_t attrs;
hpss_fileattr_t dir_attrs;
int dirfd;
int i, j, k;
char* directory;
ns_ObjHandle_t null_handle, cwd_handle;
char buf[255];
login();
memset(&dirs, 0x0, sizeof(dirs));
memset(&dir_attrs, 0x0, sizeof(dir_attrs));
directory=argv[1];
if(rc=hpss_FileGetAttributes(directory, &dir_attrs) < 0)
{
 fprintf(stderr, "Could not get attributes for %s, rc=%d\n", directory,
 rc);
}
cwd_handle = dir_attrs.ObjectHandle;
dirfd = hpss_Opendir(directory);
if(dirfd < 0)
{
 fprintf(stderr, "Could not open directory %s, rc=%d\n", directory, dirfd);
 exit(1);
}
```

```
printf("%20s %10s %10s %10s\n", "Name", "Size", "COS", "Account");
while(1)
{
 rc = hpss_Readdir(dirfd, &dirs);
 if (rc < 0)
 {
 fprintf(stderr, "Could not get directory error %d\n", rc);
 exit(1);
 }
 if(dirs.d_namelen == 0)
 break;
 rc = hpss_GetAttrHandle(&cwd_handle, dirs.d_name, NULL, NULL, &attrs);
 if(rc < 0)
 {
 fprintf(stderr, "Could not get attributes error %d\n", rc);
 exit(1);
 }
 printf("%20s %10s %10d %10d\n", dirs.d_name,
 u64tostr_r(attrs.va_size, buf), attrs.va_cos, attrs.va_account);
}
rc = hpss_Closedir(dirfd);
if(rc != 0)
{
 fprintf(stderr, "Could not close directory %d, rc=%d\n", dirfd, rc);
 exit(1);
}
```

This is a simple way to get file attributes. The entries come back one at a time and you simply get their attributes and print them out. This is one way to get file attributes, but another is to use the hpss_ReadAttrs() or hpss_Getdents() function calls. These will typically yield better performance than readdir because they can supply larger batches (hpss_Readdir() will read 128 entries at a time) and less overhead; for example, the directory entry retrieval and attribute retrieval operations occur within the same function rather than your application calling them for every entry.

4.5.4 File Input and Output

File I/O in HPSS at its most basic is familiar to anyone who has used the POSIX write and read functions before; however, there are really three modes of supported I/O in HPSS applications. One is the unbuffered APIs, another is the buffered APIs, and the third is PIO. The unbuffered APIs are simple to use and can provide good performance for small files where the overhead of setting up a complex I/O framework might take as long as just doing the I/O directly. However doing numerous small I/O within a small section of a file can be very expensive with the unbuffered APIs as each request actually writes data and generates a Core Server request.

The buffered APIs are built on top of the unbuffered APIs and to some extent hide the I/O latency by buffering write and read commands. The downside is that the buffered APIs can report success when writing data that was never actually written - the only way to know if the data made it to HPSS or not is by flushing the buffers periodically. The buffered APIs are styled after the C file stream APIs, although many options of those APIs are not currently supported by HPSS. Finally, PIO provides an abstraction of the basic protocol used to transfer data in HPSS rather than presenting a POSIX-like interface. This has the advantage of being faster than the POSIX interface as less requests must be made to the Core Server; PIO can stream hundreds of gigabytes of data between the client and the mover off a single Core Server request. The downside to PIO is that it's relatively expensive to set up and more complicated to implement. PIO also supports the PUSH protocol which is useful for WAN or high latency data movement.

It is important to remember that the underlying media will have a large impact on the performance characteristics of the I/O. Many customers use disk to tape hierarchies and so the tape is often no longer in use during I/O due to staging, but in the case where data is being written or read directly from tape the behavior of seek times or small I/O operations may have a noticeable delay due to tape positioning and startup times.

4.5.4.1 hpss_Write

```
00
int hfd;
hpss_cos_hints_t hints_in, hints_out;
hpss_cos_priorities_t hints_pri;
char *hfile;
char *buf;
int rc;
login();
```

```
memset(&hints_in, 0x0, sizeof(hints_in));
memset(&hints_out, 0x0, sizeof(hints_out));
memset(&hints_pri, 0x0, sizeof(hints_pri));
hfile=argv[1];
buf=argv[2];
hfd = hpss_Open(hfile, O_RDWR, 0777,
 &hints_in, &hints_pri, &hints_out);
if (hfd < 0)
{
 fprintf(stderr, "Could not open:  %s, error %d\n", hfile, hfd);
 exit(1);
}
rc = hpss_Write(hfd, buf, strlen(buf));
if(rc != strlen(buf))
{
 fprintf(stderr, "Could not write buf, rc=%d\n", rc);
 hpss_Close(hfd);
 exit(1);
}
fprintf(stderr, "Wrote %d bytes\n", rc);
hpss_Close(hfd);
```

hpss_Write() behaves similarly to the write system call. This snippet will write data passed in from the command line to an HPSS file specified at the command line. The file must already exist since the O_CREAT flag is not passed to hpss_Open(). hpss_Write() will return the number of bytes written. Since hpss_Write() is unbuffered the number of bytes written is the number of bytes that made it to the physical media and were committed by the database.

4.5.4.2 hpss_Read

```
00
int hfd;
hpss_cos_hints_t hints_in, hints_out;
hpss_cos_priorities_t hints_pri;
char *hfile;
int rc;
char buf[255];
login();
memset(&hints_in, 0x0, sizeof(hints_in));
memset(&hints_out, 0x0, sizeof(hints_out));
memset(&hints_pri, 0x0, sizeof(hints_pri));
hfile=argv[1];
hfd = hpss_Open(hfile, O_RDWR, 0777,
 &hints_in, &hints_pri, &hints_out);
if (hfd < 0)
{
 fprintf(stderr, "Could not open:  %s, error %d\n", hfile, hfd);
 exit(1);
}
rc = hpss_Read(hfd, buf, sizeof(buf));
if(rc <= 0)
{
 fprintf(stderr, "Could not read buf, rc=%d\n", rc);
 hpss_Close(hfd);
 exit(1);
}
fprintf(stderr, "Read %d bytes:  %s\n", rc, buf);
hpss_Close(hfd);
```

hpss_Read() behaves similarly to the read system call. This will read up to 255 bytes of data from an HPSS file specified at the command line. hpss_Read() will return the number of bytes read into the supplied buffer.

4.5.4.3 PIO Concepts

PIO does not have a common analog, and requires at least two threads. The complexity lends PIO a great deal of flexibility at the price of being more difficult to use. A PIO application will typically outperform an application that uses the unbuffered or buffered API sets, especially for very large files or files which have a large number of segments.

Although PIO can be configured to use more than one client machine to transfer files, the following explanation will focus on the case where the only one client machine is used. The two roles within the PIO framework are the Coordinator and the Participant. The Coordinator role is given file I/O requests by the client application, forwards those requests to the HPSS Core Server, and manages the Participants which have registered with it. The Participant, which is a thread on the client machine, registers itself with the Coordinator's PIO group context and actually sends

or receives the data, which is available to the client application in the form of a callback. The first step is to start the PIO coordinator thread and create the PIO group context. The options field can be used to specify whether to use PUSH or have PIO handle gaps. This is shown below:

```
00
hpss_pio_params_t params;
params.Operation = HPSS_PIO_READ;
params.ClntStripeWidth = 1;
params.FileStripeWidth = 1;
params.BlockSize = 4096;
params.IOTimeOutSecs = 10;
params.Transport = HPSS_PIO_TCPIP;
params.Options = 0;
hpss_pio_grp_t sgrp;
int rc = hpss_PIOStart(&params, &sgrp);
```

Operation is either HPSS_PIO_READ or HPSS_PIO_WRITE for reads and writes to HPSS, respectively. ClntStripeWidth is the number of participant threads. Typically, you would want to create enough participants so that there is at least one participant for each stripe of the HPSS file, but this is not required. FileStripeWidth is the stripe width of the file on HPSS. This is typically determined by the COS. If you have a file on a COS with a top level storage class with a stripe width of four, then the FileStripeWidth of files created in that COS will be four. This information should probably be obtained when you open the HPSS file. The hints_out parameter of hpss_Open() will provide the file's stripe width. Alternatively, you can specify HPSS_PIO_STRIPE_UNKNOWN if you do not know and do not want to get this information elsewhere.

BlockSize is the buffer length that is passed into hpss_PIORegister() by each participant. Transport is either HPSS_PIO_TCPIP or HPSS_PIO_MVR_SELECT. HPSS_PIO_MVR_SELECT is used for SAN3P, mainly.

There are several options which can be OR'd together and stored in Options. They are HPSS_PIO_PUSH, HPSS_PIO_PORT_RANGE and HPSS_PIO_HANDLE_GAP. HPSS_PIO_PUSH is used to make the clients connect to the movers on writes to HPSS, which might be more efficient since it has fewer network connections. HPSS_PIO_PORT_RANGE is used to specify that a range of ports can be used for the transfer on the client side. The range of ports is configured elsewhere. This has no effect when PUSH is used since the client is connecting to the mover. One thing to note is that a single file transfer using PIO can use several ports. Especially if the client is coming or going from a highly striped COS and thus has many participant threads. We have seen instances where over 30000 ports were used to transfer 10000 small files and ephemeral ports limits were reached. It is almost as easy to transfer many files with the one PIO setup as it is to transfer one file. This should be considered whenever many files are being transferred, especially if each transfer takes less than a minute. HPSS_PIO_HANDLE_GAP forces HPSS to handle the gaps which are encountered when transferring files. HPSS will zero-fill the buffer that is passed to the callback when it handles the gap. When you don't specify HPSS_PIO_HANDLE_GAP a call back will not be given for the gap. Instead, you must the hpss_pio_gapinfo_t pointer that was passed to hpss_PIOExecute(). The hpss_pio_gapinfo_t struct contains on any gaps which interrupted the hpss_PIOExecute(). You must use this information to handle the gap. This might include just seeking to the end of the gap and continuing with the read or write or writing the zeros over the gap. Here is an example of how you might handle the gaps:

```
00
 if(neqz64m(gap.Length) ||
 (neqz64m(gap.Offset)&&lt64m(currentOffset, pio->size)))
{
 U64_TO_ULL(gap.Length));
 inc64m(pio->total_moved, gap.Length);
 inc64m(io_bytes_moved, gap.Length);
 inc64m(currentOffset, gap.Length);
}
```

You can see that we did not manipulate our file in any way instead we only incremented our accounting variables. There is no need to zero fill buffers and then write those buffers although it is allowed. This section of code would be called after each time hpss_PIOExecute() returns without errors.

From there, the PIO group that was just created can be exported into a form useful by the participants. This is shown below:

```
00
rc = hpss_PIOExportGrp(sgrp, (void **)&group_buf, &group_buf_len);
```

Both group_buf and group_buf_len are output parameters - group_buf will be allocated by the hpss_PIOExportGrp() function and should be freed when the buffer is no longer needed. sgrp is the group that was output from hpss_PIOStart() function. This provides a group context that can be used by participants to register with the coordinator. This group context which comes out of hpss_PIOExportGrp() is suitable for network transfer so the participants could even be on a separate machine.

At this point in the PIO process the coordinator is ready, but the participant(s) need to be created. These could be separate processes or threads. Once the participant has the group buffer (output from hpss_PIOExportGrp() as group_buf) it can then import the group buffer as shown below:

```
00
hpss_pio_grp_t sgrp;
int rc = hpss_PIOImportGrp(group_buf, group_buf_len, &sgrp);
```

This takes the network portable group_buf and decodes it back into a group context. The group context also contains the address of the coordinator. Now we're ready to register the participant with the coordinator. Once all of the participants are registered, the coordinator can begin issuing I/O. The coordinator knows how many participants there are from the ClntStripeWidth parameter supplied to hpss_PIOStart(). If you created fewer than that number of participants, then the coordinator thread will hang and timeout without issuing any I/O. Each call to hpss_PIORegister() is done in a separate thread, so you would typically create a thread which allocates the buffer and does whatever other setup needs to be done and then calls hpss_PIORegister() once its ready to do I/O. The buffer length should be the same size as the BlockSize set in the params. If the buffer is larger than BlockSize then you will have wasted the difference, and if its smaller then hpss_PIORegister() will return an error.

```
00
rc = hpss_PIORegister(participant_index,
 NULL,
 buf,
 buflen,
 sgrp,
 pio_read_dump_buffer,
 NULL);
```

The participant is registered and requires the index of the participant (0-(N-1), where N is the client stripe width specified in the original hpss_PIOStart() parameters). The second parameter is an alternate data address to use for data connections - if NULL then the participant will listen as INADDR_ANY on a random port. The participant then supplies its data buffer, data buffer length, an I/O callback, I/O callback arguments, plus the group context it just imported.

Typically, you would create a struct that contains all the necessary information for each participant. This would include information such as local and HPSS file descriptors, buffer pointer, buffer len, etc. It might also be convenient to have a parent struct which contains transfer information such as file descriptors. This would allow you to transfer several files with only one call to hpss_PIOStart(). You also would not need to register your participants more than once. The example we use below has this type of design.

The I/O callback is important - it allows the application to access the data that is being read or written. Basically once PIO fills up a data buffer, it makes the finished buffer available to the application through the callback. Here's the full callback definition that was just registered by the hpss_PIORegister():

```
00
static int pio_read_dump_buffer(
 void *arg,
 u_signed64 offset,
 unsigned int *length,
 void **buffer)
{
 pio_proc_t *proc = (pio_proc_t *)arg;
 pio_t *pio = proc->parent;

 memcpy(proc->buf, *buffer, proc->buflen);
 unsigned char *readbuf = proc->buf;

 off_t noffset;
 CONVERT_U64_TO_LONGLONG(offset, noffset);
 if (lseek(pio->local_fd, noffset, SEEK_SET) == (off_t)-1) {
 fprintf(STDERR, "pio_read_dump_buffer():  lseek to %llu:  %s",
 (long long)noffset, strerror(errno));
 return errno;
 }

 int total = 0;
 int left = *length;
 unsigned char *bufPosition = readbuf;
 while (left) {
 int rc = write(pio->local_fd, bufPosition, left);
 if (rc < 0) {
 fprintf(STDERR, "pio_read_dump_buffer():  write:  %s",
 strerror(errno));
 return errno;
 }
 total += rc;
 bufPosition += rc;
 left -= rc;
 }
```

```
 return 0;
}
```

The first argument to the callback is the last argument that was given to hpss_PIORegister(). This void pointer could be different for each participant. Thus each participant can have its own buffer, or rather, should have its own buffer. You receive the void pointer and cast it as the proper struct. In this example, we cast arg as a pio_proc_t pointer, which contains a pointer to a base struct, pio_t, which contains the local and HPSS file descriptors that are being read to or from. This means you can change the place that the callback writes to just by replacing file descriptors in the pio_t struct.

The callback specifies a buffer and an offset; it's important to seek to the specified offset before using the data. This callback function copies the buffer containing the data read into the participant buffer and then writes it to the local file; a similar callback would be used for file writes where the callback would read a buffer from a source and supply it to PIO, and it would be registered in the same way.

Okay, so now we have a coordinator started, participants registered with appropriate callbacks. All that's left is to actually execute the I/O. This is done with the hpss_PIOExecute() function:

```
00
rc = hpss_PIOExecute(pio->hpss_fd, add64m(starting_offset, current_offset), current_iosize,
 sgrp, gap_ptr, &bytes_moved);
```

PIOExecute should be called in a loop until all of the I/O is received. In many cases, depending upon the storage class configuration, a single hpss_PIOExecute() will retrieve all of the data, however, that can never be counted on. Here you see the first mention of an open HPSS file. hpss_fd is an HPSS file descriptor that will be read / written. Following each hpss_PIOExecute() make sure to increment the current offset and decrement the current_iosize by the number of bytes moved. When a gap is encountered, the gap size should also be added to the offset.

This example has been largely about read, however performing PIO writes is essentially the same - the only differences are the operation passed in by the group parameters (HPSS_PIO_WRITE) and the callback (read from disk and load PIO buffer rather than load local buffer and write to disk).

4.5.5 HSM Functions

The HSM functions allow the application developer to control where the data resides in a class of service. When used in conjunction with the extended attributes, which provide the ability to see which levels data resides on, these functions can be used to ensure data is at the desired level. These functions can be used, for example, to force HPSS to immediately migrate a file to tape and then purge it from disk. It can also be used to pre-stage data that the application is aware will be needed soon from tape on to disk and force it to stay on disk.

It should be noted that these HSM functions, when improperly used, can result in inefficiency and wasted resources. For best results, migration and stage requests should be pooled and if possible ordered to reduce thrashing.

4.5.5.1 Migrate

The hpss_Migrate() function requires a user to have a special ACL with the Core Server. hpssmps is an authorized caller of the Core Server and so it may be used as an example. The reason for this is that migrations cause tape mounts and issuing migrations in an inefficient way could cause inefficient use of resources within the system vs. allowing the Migration Purge Server to manage migrations. Users who do not have the special ACL cannot issue migrations and their requests will be rejected.

The following code migrates a file and prints out the number of bytes moved. If a file has already been migrated from level 0, zero bytes will be moved. Keep in mind that hpss_Migrate() will not return until the file has been migrated or an error occurs, and so the program may take some time depending upon the size of the file and the characteristics of level 1 of the file's class of service.

```
00
int rc;
char buffer[255];
int i;
char* hfile;
int hfd;
u_signed64 bytes = cast64m(0);
```

```
login();
hfile = argv[1];
hfd = hpss_Open(hfile, O_RDWR, 000, NULL, NULL, NULL);
if(hfd < 0)
{
 printf("Failed to open file %s, rc=%d\n", hfile, rc);
 return;
}
rc = hpss_Migrate(hfd, 0, 0, &bytes);
if(rc != 0)
{
 printf("Failed to Migrate %s, rc=%d\n", hfile, rc);
 return;
}
else
{
 printf("Migrated %s bytes\n", u64tostr_r(bytes, buffer));
}
```

4.5.5.2 Purge

The hpss_Purge() API is used to unlink data at one level of the hierarchy and allow that space to be reclaimed for use by other files. hpss_Purge() always requires that data exist at some other level - it will not allow data to be lost by purging. The most common usage is shown below, where a whole file is purged from the top level of the class of service. Unlike hpss_Migrate() which involves the movement of data and potentially tape mounts, etc, Purge is a metadata-only operation and so will be faster than Migrate for essentially all cases.

```
00
int rc;
char buffer[255];
int i;
char* hfile;
int hfd;
u_signed64 bytes = cast64m(0);
login();
hfile = argv[1];
hfd = hpss_Open(hfile, O_RDWR, 000, NULL, NULL, NULL);
if(hfd < 0)
{
 printf("Failed to open file %s, rc=%d\n", hfile, hfd);
 return;
}
rc = hpss_Purge(hfd, cast64m(0), cast64m(0), 0, BFS_PURGE_ALL, &bytes);
if(rc != 0)
{
 printf("Failed to Purge %s, rc=%d\n", hfile, rc);
 return;
}
else
{
 printf("Purged %s bytes\n", u64tostr_r(bytes, buffer));
}
```

Like hpss_Migrate(), hpss_Purge() will report 0 bytes purged if no bytes existed at the level where the purge was requested.

4.5.5.3 Purge Lock

When scheduling data it can be important to make sure that the purge policies do not cause certain files to be purged off fast, top-level media. The following example opens a file, purge locks it, and then purge unlocks it. Remember that purge locks can have an expiration time set in the purge policy so it's important to keep in mind that a purge lock can be removed automatically, which can be a double-edged sword. The expiration time can be disabled if desired; consult the *HPSS Management Guide* for more information.

```
00
int rc;
char buffer[255];
int i;
char* hfile;
int hfd = 0;
u_signed64 bytes = cast64m(0);
login();
hfile = argv[1];
hfd = hpss_Open(hfile, O_RDWR, 000, NULL, NULL, NULL);
if(hfd < 0)
{
```

```
 printf("Failed to open file %s, rc=%d\n", hfile, rc);
 return;
}
printf("Opened file %d\n", hfd);
rc = hpss_PurgeLock(hfd, PURGE_LOCK);
if(rc != 0)
{
 printf("Failed to purge lock %s, rc=%d\n", hfile, rc);
 return;
}
else
{
 printf("%s purge locked\n", hfile);
}
rc = hpss_PurgeLock(hfd, PURGE_UNLOCK);
if(rc != 0)
{
 printf("Failed to purge lock %s, rc=%d\n", hfile, rc);
 return;
}
else
{
 printf("%s purge unlocked\n", hfile);
}
```

Purge locks are only available for the top level of the hierarchy; if no data exists at the top level of the hierarchy an error is returned. This example immediately purge unlocks the file; in a real application you would likely wait for some condition to occur (such as the file being read by the waiting application) before doing the purge unlock.

4.5.5.4 Stage

As previously mentioned, when an application is aware that many stages may be upcoming, it is important to order the stage requests in such a way as to make the most efficient use of the tape drives involved. Unordered stages can lead to tapes being dismounted just before a stage request for a file on that tape is issued, or for a file at one end of a tape to be retrieved, followed by a file at the other end, followed by a file at the other end, etc. This can cause unnecessary delay and also wear and tear on the physical media.

Staging copies the file from a lower level up to a higher level of a storage hierarchy. The data still exists at the lower level, however. The following example opens a file and stages it. Keep in mind that if the COS is set up to stage on open you may need to include the O_NONBLOCK flag to keep the file from being staged.

```
00
int rc;
char buffer[255];
int i;
char* hfile;
int hfd = 0;
u_signed64 bytes = cast64m(0);
login();
hfile = argv[1];
hfd = hpss_Open(hfile, O_RDWR | O_NONBLOCK, 000, NULL, NULL, NULL);
if(hfd < 0)
{
 printf("Failed to open file %s, rc=%d\n", hfile, rc);
 return;
}
printf("Opened file %d\n", hfd);
rc = hpss_Stage(hfd, 0, cast64m(0), 0, BFS_STAGE_ALL);
if(rc != 0)
{
 printf("Failed to stage %s, rc=%d\n", hfile, rc);
 return;
}
else
{
 printf("%s staged\n", hfile);
}
```

It's also important to keep in mind that tape is serial; it's possible that stages could be delayed significantly by some sort of long, ongoing I/O in progress on the same tape. That condition can be easily seen in the HPSS RTM screen where the stage request wait string will show it as blocked by another request.

4.5.6 User-defined Attributes (UDA)

The HPSS UDAs allow the arbitrary tagging of files with user-defined metadata in a hierarchical format. A full description of UDAs is out of the scope of this guide, however a few common guidelines are appropriate. -

1. If you plan to use an XML schema to manage the UDAs, it is important to define it early since any attributes stored before the schema is in place which violate the schema will keep that object from being updated once the schema is installed.
2. Batching UDAs operations is typically somewhat faster than issuing individual operations
3. Using the XQuery interface yields the best performance at the cost of additional design complexity.
4. Be very careful and test custom XQueries as it's easy to make mistakes and wind up with multiple entries of the same attribute name
5. Identify any attributes that should be searchable up front in your application design and create an XML index for each of them - searching attributes which do not have an index is very poor performing

4.5.6.1 Set UDAs

```
00
char *hfile;
int rc;
hpss_userattr_list_t inAttr;
login();
hfile=argv[1];
// set up UDA structure
inAttr.len = 1;
inAttr.Pair = malloc(inAttr.len * sizeof(hpss_userattr_t));
inAttr.Pair[0].Key = "/hpss/bestpractice/snippet";
inAttr.Pair[0].Value = "snip value";
// set the attribute
rc = hpss_UserAttrSetAttrs(hfile, &inAttr,NULL);
if(rc == 0)
 printf("Successfully set attribute\n");
else
 printf("Failed to set attribute, rc=%d\n", rc);
free(inAttr.Pair);
```

This snippet will create the attribute called "/hpss/bestpractice/snippet" if it does not exist and assign it the value of "snip value". If any value previously existed it is removed. The way these attribute names work is that they are hierarchical; they are referred to as "attribute paths" or "XPaths" because the UDA technology is based upon XQuery and XPath technology. The full UDA content for a file is stored within a single XML document within the database, and each update or retrieval references one or more paths within that XML document.

Large numbers of attributes can be set at once using the hpss_userattr_list_t structure, and those updates do happen within a single transaction.

4.5.6.2 Get UDA

Retrieving an attribute is even simpler than setting one. Just fill in the attribute pair with the key and the value will be retrieved. It's important to note that in the case where multiple attributes with the same name exist that the first attribute is retrieved by default. To specify a different attribute the one-based index must be provided (this is standard for XPath).

```
00
char *hfile;
int rc;
hpss_userattr_list_t inAttr;
int i;
int flag = UDA_API_VALUE;
login();
hfile=argv[1];
// set up UDA structure
inAttr.len = 1;
inAttr.Pair = malloc(inAttr.len * sizeof(hpss_userattr_t));
inAttr.Pair[0].Value = malloc(sizeof(char)*HPSS_XML_SIZE);
inAttr.Pair[0].Key = "/hpss/bestpractice/snippet";
```

```
rc = hpss_UserAttrGetAttrs(argv[1], &inAttr, flag, HPSS_XML_SIZE);
if(rc != 0)
{
 if(rc == -ENOENT)
 printf("No User-defined Attributes.\n");
 else
 printf("hpss_UserAttrGetAttrs returned %d\n", rc);
}
else
{
 for(i = 0; i < inAttr.len; i++) {
 char* xmlstr = hpss_ChompXMLHeader(inAttr.Pair[i].Value, NULL);
 printf("%s=\t\t%s\n", inAttr.Pair[i].Key, xmlstr);
 free(xmlstr);
 }
}
free(inAttr.Pair[0].Value);
free(inAttr.Pair);
```

This snippet will read the value of the attribute "/hpss/bestpractice/snippet" back and display it. Note that the hpss_ChompXMLHeader() function is called on the string prior to it being displayed; each retrieved value which comes back always includes an XML header which is currently not used for anything, so we simply remove it.

Also, this example provides the output as a value, but it's also possible to retrieve the XML. This allows this function to be used to potentially return larger groups of values all under the same base attribute. For example, you could retrieve "/hpss/widget/a", "/hpss/widget/b", "/hpss/widget/c", and "/hpss/widget/d" at the same time by retrieving "/hpss/widget" in XML and then parsing that XML.

4.5.6.3 List UDA

You may not always know the attributes on a file, and simply want to see the full file contents. This is where the convenience function, hpss_UserAttrsListAttrs() comes in. hpss_UserAttrsListAttrs() is essentially a wrapper around an XML get of "/hpss" on a certain file. As described in the previous section, retrieving XML retrieves each of the children as well. List works by retrieving the entire XML document and then parsing it out into attribute key-value pairs.

```
00
char *hfile;
int rc;
hpss_userattr_list_t inAttr;
int i;
int flag = XML_ATTR;
login();
hfile=argv[1];
rc = hpss_UserAttrListAttrs(argv[1], &inAttr, flag, HPSS_XML_SIZE);
if(rc != 0)
{
 if(rc == -ENOENT)
 printf("No User-defined Attributes.\n");
 else
 printf("hpss_UserAttrListAttrs returned %d\n", rc);
}
else
{
 for(i = 0; i < inAttr.len; i++) {
 printf("%s=\t\t%s\n", inAttr.Pair[i].Key, inAttr.Pair[i].Value);
 free(inAttr.Pair[i].Value);
 free(inAttr.Pair[i].Key);
 }
}
free(inAttr.Pair);
```

The List API is surprisingly fast due to the fact that it only issues a single request to HPSS. The most common problem when doing a List is running into the ERANGE error. This is due to the application using a buffer size that is too small for the result set. In some cases an EDOM may be returned, which signifies that the Core Server buffer size limitation has been reached. In this case the limit may need to be increased or the attributes culled for space.

4.5.6.4 Delete UDA

Deleting attributes is nearly identical to setting them. The only difference is that no value should be provided as it will be ignored.

```
00
```

```
char *hfile;
int rc;
hpss_userattr_list_t inAttr;
login();
hfile=argv[1];
// set up UDA structure
inAttr.len = 1;
inAttr.Pair = malloc(inAttr.len * sizeof(hpss_userattr_t));
inAttr.Pair[0].Key = "/hpss/bestpractice/snippet";
// set the attribute
rc = hpss_UserAttrDeleteAttrs(hfile, &inAttr, NULL);
if(rc == 0)
 printf("Successfully deleted attribute\n");
else
 printf("Failed to delete attribute, rc=%d\n", rc);
free(inAttr.Pair);
```

There are a few notable differences between some delete functions in HPSS such as unlink and the User Defined Attributes delete. The first is that the UDA delete function will return success even if the attribute in question never existed. There are two reasons for this: technical and functional. The technical reason for this behavior is because that's how the XQuery delete behaves; an XQuery to delete a non-existing attribute is successful. The functional reason is that since multiple attributes can be deleted at once, even if some of the attributes no longer exist they can still be deleted without causing the entire operation to fail. The other way the UDA delete is different from say hpss_Unlink() is that it's essentially recursive with respect to child attributes. For example, in a file which might have /hpss/a/a, /hpss/a/b, and /hpss/a/c attributes, deleting /hpss/a will delete /hpss/a/a, /hpss/a/b, and /hpss/a/c. The path /text() can be appended on to the attribute in order to remove this recursive behavior and only delete a value associated with a particular attribute. For example /hpss/a/text() would return any value associated with /hpss/a, but not any of its child nodes such as /hpss/a/a.

4.5.6.5 Using Custom XQueries

The custom XQuery interface is an incredibly powerful tool that allows the application programmer to directly manipulate the XML document associated with the file. The best way to reduce the risk of a custom XQuery doing something unintended is through good application testing using a variety of user metadata.

First, an example of a custom XQuery:

```
00
char *xquery;
char *hfile;
int rc;
login();
hfile=argv[1];
xquery = "copy $new := $DOC modify(if ($new/hpss/test) then do replace value of $new/hpss/test with \"b\"
      else if($new/hpss) then do insert <test>{xs:string(\"b\")}</test> into $new/hpss else ()) return
      $new";
// set the attribute
rc = hpss_UserAttrXQueryUpdate(hfile, xquery, NULL);
if(rc == 0)
 printf("Successfully ran xquery update\n");
else
 printf("Failed to run xquery update, rc=%d\n", rc);
```

XQuery is a broad topic beyond the scope of this guide, however let's briefly go through the XQuery string above in order to give a hint of the complexity. So first, the XML document is copied into a sort of local variable ($new). Next the command is to modify this document. The logic must be split here because we may either be inserting a new node into the XML document or updating an existing node. One very important consideration is that every level within the XML document may need to be handled separately. For instance, in order to set /hpss/really/long/path to foo it may be necessary to have one conditional which updates path to foo, another which inserts

```
<path>
```

into /hpss/really/long with the value of foo, another which inserts

```
<long><path>
```

into /hpss/really with the value of foo, and so on. One thing the standard APIs never do that XQuery can easily do (and was hinted at in the previous paragraph) is insert and manipulate duplicate entries. For example, if the first condition was not there and the XQuery was simply..

```
"copy $new := $DOC modify(if($new/hpss) then do insert \
<test>{xs:string(\"b\")}</test> into $new/hpss else ()) return $new"
```

then every invocation of that XQuery would add another element called "test" with the value of "b" into /hpss. That's one of the major ways that the XQuery interface can be accidentally misused.

For more information on XQuery consult the DB2 documentation on the subject; not all operations are supported within DB2, and DB2 has a few special characteristics that need to be considered beyond the normal scope of XQuery.

4.5.6.6 Using Custom XQueries for Retrieval

The UDA data within a single file can be retrieved or manipulated prior to its retrieval using the hpss_UserAttrXQueryGet() API. This API supports the FLWOR syntax which is a very nice way to search through the XML tree. The FLWOR language is again out of scope for this guide, but like the other technologies it is well-documented by DB2 and in its own right. Here's an example which will print the number of key/value pairs that follow and up to the first ten keys and the values underneath the /hpss/a attribute in a comma and space-delimited format. For example, if two attributes exist under /hpss/a (/hpss/a/a=2 and /hpss/a/b=3) then the output would be: "2 a,2 b,3".

```
00
char *xquery;
int rc;
char buffer[HPSS_XML_SIZE];
int i;
int bufsize=10;
char* hfile;
login();
hfile = argv[1];
xquery = "let $j := for $v in $DOC/hpss/a/* return string-join((string(node-name($v)),data($v)),\",\")
      return insert-before(subsequence($j, 0, 10), 0, count($j))";
rc = hpss_UserAttrXQueryGet(hfile, xquery, buffer, sizeof(buffer));
if(rc != 0)
{
 printf("Failed to get xquery results, rc=%d\n", rc);
 return;
}
else
{
 printf("XQuery returned:  %s\n", buffer);
}
```

Don't worry if the xquery doesn't make a lot of sense to you. Essentially FLWOR just allows the developer to iterate over a selection of nodes, perhaps apply some transformation, and return them. XQuery is an extremely flexible tool, and unlike the update function no updates occur so it's inherently less dangerous. XQuery is beyond the scope of this document, however there are plenty of resources on the Internet that describe XQuery and XPath.

4.5.6.7 Searching UDA

Searching UDAs is limited to users with special "trusted" status with the Core Server, such as root. Since the search capabilities could potentially include tens of millions of files tagged with UDA attributes it was important to restrict the harm that could be done by a single user or inefficient XQuery could do to the performance of the system.

That said, even administrators should consider their XQuery criteria carefully and follow the guidelines and advice of DB2 when crafting their XQuery searches. The two most important things in the author's experience are using wildcards sparingly, if ever, and having a lean XML index for your criteria. Now, an example:

```
00
char *xquery;
int subsys;
int rc;
object_id_list_t ObjectList;
char path[HPSS_MAX_PATH_NAME];
char idstr[24];
ns_ObjHandle_t fsroot;
u_signed64 id;
int i;
int bufsize=20;
hpss_cursor_id_t cursorid;
memset(&cursorid, 0x0, sizeof(cursorid));
login();
```

```
subsys=atoi(argv[1]);
xquery = "$DOC/hpss";
rc = hpss_UserAttrGetObjQuery(subsys, bufsize, &ObjectList,
 xquery, &cursorid);
if(rc != 0)
 return;
printf("\n");
printf("Found %d objects:\n", ObjectList.ObjectList.ObjectList_len);
printf("%24s\t\t%24s\n", "ID", "PATH");
printf("-------------------------------------------------------------------\n");
for(i = 0; i < ObjectList.ObjectList.ObjectList_len; i++) {
 id = ObjectList.ObjectList.ObjectList_val[i];
 rc = hpss_GetPathObjId(id, subsys, &fsroot, path);
 if(rc != 0)
 {
 sprintf(path,"invalid path");
 free(ObjectList.ObjectList.ObjectList_val);
 return;
 }
 u64_to_decchar(id, idstr);
 printf("%24s\t\t%24s\n",idstr, path);
}
free(ObjectList.ObjectList.ObjectList_val);
```

This example will list every file which has an entry in the UDA table, and is probably an example of a search you wouldn't want to run very often on a system which makes heavy use of UDAs. The search criteria is essentially limited to XPaths plus attribute matching, which is outside of the scope of this document and widely documented on the Internet.

Essentially any XPath can be searched on, and can include wild cards or value matches. For example, to find all files that have the /hpss/color attribute set to green, the query is $DOC/hpss[color="green"]. XPaths can contain conjunctive operations such as ands and ors as well, although it should be noted that these can increase processing time substantially.

There are three types of search. ObjId returns the object id of the matching file and has been by far the most useful in most user's experience. XML returns the matching XML, but not any file information. XMLObjId returns the object id and matching XML, and can also be used in a double constraint mode that works like "Find Object and XML matching Query 1 where Object also matches Query 2". The usage of these functions is essentially identical to the ObjId case but instead of Object Ids the search returns XML or XML and Object Ids.

The search APIs return data in sets, and in many or even most cases the result set may be larger than the set size specified. In this case an alternative search function may be called to retrieve additional results from the same query. Here's an example of that:

```
00
while(!done)
{
 rc = hpss_UserAttrReadObjs(Subsys, &objs, &cursorid);
 if(rc != HPSS_E_NOERROR)
 {
 (void)hpss_UserAttrCloseCursor(Subsys, &cursorid);
 break;
 }
 else
 {
 // Let's convert these new object ids to path names and print them
 for(i = 0; i < objs.ObjectList.ObjectList_len; i++)
 {
 rc = hpss_GetPathObjId(objs.ObjectList.ObjectList_val[i], Subsys, &fsroot, path);
 if(rc == HPSS_E_NOERROR)
 printf("%d) %s\n", idx++, path);
 else
 printf("%d) error parsing path name, obj=%u.%u, rc=%d", idx++,
 high32m(objs.ObjectList.ObjectList_val[i]),
 low32m(objs.ObjectList.ObjectList_val[i]), rc);
 }
 free(objs.ObjectList.ObjectList_val);
 }
}
```

The previous snippet will continue retrieving data from the original search using the cursor structure until an error occurs. When no more results are available the ReadObjs function returns a ESRCH. In this way millions of search result entries can be retrieved from HPSS, although in that case it may be prudent to use a buffer size larger than 20 entries.

4.5.7 Using UDAs with VFS

VFS uses the UDA namespace /hpss/fs/user. By default any attributes which match that pattern will be accessible via VFS using standard extended attributes tools. For example, setting an attribute "/hpss/fs/user.test" would be visible in VFS as "user.test". Attributes set in VFS will be updated in the same way. Any attributes which are not under the /hpss/fs/user namespace will not be visible via HPSS VFS, with a few exceptions described below.

In order to make use of the standard HPSS checksum attributes, VFS maps the HPSS checksum attribute names to names which are compatible with VFS; it maps them under the /hpss/fs/user XPath. Applications or users setting or getting these checksum attributes via VFS must use these translated attribute names rather than the standard HPSS checksum paths. The following table describes the mapping:

HPSS Checksum Attribute	VFS Mapped Checksum Attribute
/hpss/user/cksum/checksum	/hpss/fs/user.hash.checksum
/hpss/user/cksum/algorithm	/hpss/fs/user.hash.algorithm
/hpss/user/cksum/state	/hpss/fs/user.hash.state
/hpss/user/cksum/lastupdate	/hpss/fs/user.hash.lastupdate
/hpss/user/cksum/errors	/hpss/fs/user.hash.errors
/hpss/user/cksum/filesize	/hpss/fs/user.hash.filesize
/hpss/user/cksum/app	/hpss/fs/user.hash.app

4.5.8 HPSS to POSIX and Back Again

There are several structures returned by HPSS which do not provide what would be considered standard POSIX information. Luckily, there are convenience functions which take this information and put it into a POSIX format. While the information doesn't always match one-to-one with POSIX, it is as close a translation as possible.

4.5.8.1 HPSS Mode to POSIX Mode

```
00
int rc;
char *file;
hpss_fileattr_t attrs;
mode_t mode;
login();
memset(&attrs, 0x0, sizeof(attrs));
file=argv[1];
rc = hpss_FileGetAttributes(file, &attrs);
if (rc < 0)
{
 fprintf(stderr, "Could not get attributes:  %s, error %d\n", file, rc);
 exit(1);
}
API_ConvertModeToPosixMode(&attrs.Attrs, &mode);
```

This will translate the hpss_Attrs_t Type, ModePerms, UserPerms, GroupsPerms, and OtherPerms fields into a mode_t, with its bit fields set with appropriate values based upon the machine where the POSIX conversion function ran. The attributes returned by this conversion are OS-specific and machine-specific, and care should be taken when using these mode bits in a mixed environment. There's also a reciprocal function which takes a POSIX mode and sets the appropriate fields in an HPSS Attr structure.

4.5.8.2 HPSS Time to POSIX Time

HPSS has timestamps for creation time, modified time, write time, and read time while POSIX has access time, modify time, and change time. The API_ConvertTimeToPosixTime() function converts the HPSS timestamps as nearly as possible to the POSIX timestamps.

```
00
int rc;
char *file;
hpss_fileattr_t attrs;
```

```
timestamp_sec_t atime;
timestamp_sec_t mtime;
timestamp_sec_t ctime;
login();
memset(&attrs, 0x0, sizeof(attrs));
file=argv[1];
rc = hpss_FileGetAttributes(file, &attrs);
if (rc < 0)
{
 fprintf(stderr, "Could not get attributes:  %s, error %d\n", file, rc);
 exit(1);
}
API_ConvertTimeToPosixTime(&attrs.Attrs, &atime, &mtime, &ctime);
printf("a=%d, m=%d, c=%d\n", atime, mtime, ctime);
```

You can expect to see a few anomalies - HPSS does not strictly follow POSIX guidelines for updating timestamps. Some timestamps may be updated in a lazy fashion, for example. In addition, some timestamps could be updated based upon system activities such as migrations or purges.

4.5.9 More On UDAs

4.5.9.1 Guidelines, Best Practices, and Standard Attributes

There are several guidelines that should be followed when using UDAs in your application. The first is that, in order to maintain namespace uniqueness, the application should place all of its attributes under a common XPath name (perhaps some form of the application name). For example, the following is good usage which allows for uniqueness:

UDA Description	XPath
Date	/hpss/myapp/date
User	/hpss/myapp/user
Id	/hpss/myapp/id

Here's an example of some poor attribute usage:

UDA Description	XPath
Date	/hpss/data
User	/hpss/user
Id	/hpss/id

Without including an application identifier to put these application attributes inside, the attributes reside on the top level. If this is done by multiple applications, then there's a good chance that common attribute names could be overwritten by poor behaving applications. Using an application identifier safeguards these fairly common attribute names from accidental overwrites by other applications.

Another guideline is that, since many applications do checksumming, there a common location and set of attributes has been defined where checksums may be stored. Storing checksums in this way will allow the applications to be compatible with each other and HPSS tools such as **hpsssum**. These attributes and their meanings are listed below:

UDA Description	VFS Path	Comments
Checksum Hash	/hpss/user/cksum/checksum	The checksum hash in hex format. All lower case is preferred, but it should be case insensitive.

Algorithm Name	/hpss/user/cksum/algorithm	See the **hpsssum** man page for a list of checksum names that HPSS supports. If an application or tool does not support the specified checksum type it should error out. Algorithm names should be case insensitive.
Checksum State	/hpss/user/cksum/state	Legal values: "Valid" - No errors detected validating the checksum "Error" - The last validation attempt was unsuccessful due to an error. "Invalid" - Validation failed; the generated checksum during a previous read did not match the saved checksum. Files which are in "Error" or "Valid" should open normally, files which are "Invalid" should fail to open with EILSEQ or allow the user to determine whether to continue or not. Checksum state should be case insensitive.
Last time the checksum was updated	/hpss/user/cksum/lastupdate	This is the time of the last checksum hash attribute update in seconds since Epoch.
Number of errors reading the file	/hpss/user/cksum/errors	Informational only; may be reset to 0 following a successful read and verification of the file against the stored checksum.
Size of the check-summed file	/hpss/user/cksum/filesize	This is the size, in bytes, of the check-summed file.
Application Name	/hpss/user/cksum/app	This is the name of the application which last updated the checksum hash. Using an unsupported algorithm name will result in **hpsssum** being unable to verify the file. **hpsssum** can be used to identify files which are in the 'error' or 'invalid' state, although it is recommended that XML indexes be created if this feature is intended to be used in a production scenario. (See the xmladdindex man page).

4.5.10 Useful References

1. http://www.ibm.com/developerworks/data/library/techarticle/dm-0604saracco/
   - Using XQuery
2. http://www.ibm.com/developerworks/data/library/techarticle/dm-0710nicola/
   - Using XQuery Update
3. https://scholar.google.com/scholar?hl=en&as_sdt=0%2C44&q=Using+xml+and+xquery+for+d
   - Using xml and xquery for data management in hpss

Chapter 5

Developer Acknowledgements

HPSS is a product of a government-industry collaboration. The project approach is based on the premise that no single company, government laboratory, or research organization has the ability to confront all of the system-level issues that must be resolved for significant advancement in high-performance storage system technology.

HPSS development was performed jointly by IBM Worldwide Government Industry, Lawrence Berkeley National Laboratory, Lawrence Livermore National Laboratory, Los Alamos National Laboratory, NASA Langley Research Center, Oak Ridge National Laboratory, and Sandia National Laboratories.

We would like to acknowledge Argonne National Laboratory, the National Center for Atmospheric Research, and Pacific Northwest Laboratory for their help with initial requirements reviews.

We also wish to acknowledge Cornell Information Technologies of Cornell University for providing assistance with naming service and transaction management evaluations and for joint developments of the Name Service.

In addition, we wish to acknowledge the many discussions, design ideas, implementation and operation experiences we have shared with colleagues at the National Storage Laboratory, the IEEE Mass Storage Systems and Technology Technical Committee, the IEEE Storage System Standards Working Group, and the storage community at large.

We also wish to acknowledge the Cornell Theory Center and the Maui High Performance Computer Center for providing a test bed for the initial HPSS release. We also wish to acknowledge Gleicher Enterprises, LLC for the development of the HSI, HTAR and Transfer Agent client applications.

Finally, we wish to acknowledge CEA-DAM (Commissariat a l'Energie Atomique - Centre d'Etudes de Bruyeres-le-Chatel) for providing assistance with development of NFS V3 protocol support.

Chapter 6

Further Reading

1. *3580 Ultrium Tape Drive Setup, Operator and Service Guide* GA32-0415-00
2. *3584 UltraScalable Tape Library Planning and Operator Guide* GA32-0408-01
3. *3584 UltraScalable Tape Library SCSI Reference* WB1108-00
4. *HPSS Error Messages Reference Manual*, current release.
5. *HPSS Programmer's Reference* , current release.
6. *HPSS Programmer's Reference* - I/O Supplement , current release.
7. *HPSS User's Guide*, current release.
8. *IBM SCSI Device Drivers: Installation and User's Guide*, GC35-0154-01
9. *IBM Ultrium Device Drivers Installation and User's Guide* GA32-0430-00.1
10. Parallel and ESCON Channel Tape Attachment/6000 Installation and User's Guide, GA32-0311-02
11. POSIX 1003.1-1990 Tar Standard
12. Reference Guide AMU, Order no. DOC E00 005
13. STK Automated Cartridge System Library Software (ACSLS) System Administrator's Guide, PN 16716
14. STK Automated Cartridge System Library Software Programmer's Guide, PN 16718
15. J. Steiner, C. Neuman, and J. Schiller, "Kerberos: An Authentication Service for Open Network Systems," USENIX 1988 Winter Conference Proceedings (1988).
16. R.W. Watson and R.A. Coyne, "The Parallel I/O Architecture of the High-Performance Storage System (HPSS)," from the 1995 IEEE MSS Symposium, courtesy of the IEEE Computer Society Press.
17. T.W. Tyler and D.S. Fisher, "Using Distributed OLTP Technology in a High-Performance Storage System," from the 1995 IEEE MSS Symposium, courtesy of the IEEE Computer Society Press.
18. J.K. Deutsch and M.R. Gary, "Physical Volume Library Deadlock Avoidance in a Striped Media Environment," from the 1995 IEEE MSS Symposium, courtesy of the IEEE Computer Society Press.
19. R. Grossman, X. Qin, W. Xu, H. Hulen, and T. Tyler, "An Architecture for a Scalable, High-Performance Digital Library," from the 1995 IEEE MSS Symposium, courtesy of the IEEE Computer Society Press.
20. S. Louis and R.D. Burris, "Management Issues for High-Performance Storage Systems," from the 1995 IEEE MSS Symposium, courtesy of the IEEE Computer Society Press.
21. D. Fisher, J. Sobolewski, and T. Tyler, "Distributed Metadata Management in the High Performance Storage System," from the 1st IEEE Metadata Conference, April 16-18, 1996.

Chapter 7

Glossary of Terms and Acronyms

ACI Automatic Media Library Client Interface

ACL Access Control List

ACSLS Automated Cartridge System Library Software (Science Technology Corporation)

ADIC Advanced Digital Information Corporation

accounting The process of tracking system usage per user, possibly for the purposes of charging for that usage. Also, a log record type used to log accounting information.

AIX Advanced Interactive Executive. An operating system provided on many IBM machines.

alarm A log record type used to report situations that require administrator investigation or intervention.

AML Automated Media Library. A tape robot.

AMS Archive Management Unit

ANSI American National Standards Institute

API Application Program Interface

archive One or more interconnected storage systems of the same architecture.

attribute When referring to a managed object, an attribute is one discrete piece of information, or set of related information, within that object.

attribute change When referring to a managed object, an attribute change is the modification of an object attribute. This event may result in a notification being sent to SSM, if SSM is currently registered for that attribute.

audit (security) An operation that produces lists of HPSS log messages whose record type is SECURITY. A security audit is used to provide a trail of security-relevant activity in HPSS.

bar code An array of rectangular bars and spaces in a predetermined pattern which represent alphanumeric information in a machine readable format (e.g., UPC symbol)

BFS HPSS Bitfile Service.

bitfile A file stored in HPSS, represented as a logical string of bits unrestricted in size or internal structure. HPSS imposes a size limitation in 8-bit bytes based upon the maximum size in bytes that can be represented by a 64-bit unsigned integer.

bitfile segment An internal metadata structure, not normally visible, used by the Core Server to map contiguous pieces of a bitfile to underlying storage.

Bitfile Service Portion of the HPSS Core Server that provides a logical abstraction of bitfiles to its clients.

bytes between tape marks The number of data bytes that are written to a tape virtual volume before a tape mark is required on the physical media.

cartridge A physical media container, such as a tape reel or cassette, capable of being mounted on and dismounted from a drive. A fixed disk is technically considered to be a cartridge because it meets this definition and can be logically mounted and dismounted.

central log The main repository of logged messages from all HPSS servers. Usually /var/hpss/log/HPSS.log.

class A type definition in Java. It defines a template on which objects with similar characteristics can be built, and includes variables and methods specific to the class.

Class of Service A set of storage system characteristics used to group bitfiles with similar logical characteristics and performance requirements together. A Class of Service is supported by an underlying hierarchy of storage classes.

cluster The unit of storage space allocation on HPSS disks. The smallest amount of disk space that can be allocated from a virtual volume is a cluster. The size of the cluster on any given disk volume is determined by the size of the smallest storage segment that will be allocated on the volume, and other factors.

configuration The process of initializing or modifying various parameters affecting the behavior of an HPSS server or infrastructure service.

COS Class of Service

Core Server An HPSS server which manages the namespace and storage for an HPSS system. The Core Server manages the Name Space in which files are defined, the attributes of the files, and the storage media on which the files are stored. The Core Server is the central server of an HPSS system. Each storage sub-system uses exactly one Core Server.

daemon A UNIX program that runs continuously in the background.

DB2 A relational database system, a product of IBM Corporation, used by HPSS to store and manage HPSS system metadata.

debug A log record type used to report internal events that can be helpful in troubleshooting the system.

delog The process of extraction, formatting, and outputting HPSS central log records. This process is obsolete in 7.4 and later versions of HPSS. The central log is now recorded as flat text.

deregistration The process of disabling notification to SSM for a particular attribute change.

descriptive name A human-readable name for an HPSS server.

device A physical piece of hardware, usually associated with a drive, that is capable of reading or writing data.

directory An HPSS object that can contain files, symbolic links, hard links, and other directories.

dismount An operation in which a cartridge is either physically or logically removed from a device, rendering it unreadable and unwritable. In the case of tape cartridges, a dismount operation is a physical operation. In the case of a fixed disk unit, a dismount is a logical operation.

DNS Domain Name Service

DOE Department of Energy

drive A physical piece of hardware capable of reading and/or writing mounted cartridges. The terms device and drive are often used interchangeably.

event A log record type used to report informational messages (e.g., subsystem starting, subsystem terminating).

export An operation in which a cartridge and its associated storage space are removed from the HPSS system Physical Volume Library. It may or may not include an eject, which is the removal of the cartridge from its Physical Volume Repository.

file An object than can be written to, read from, or both, with attributes including access permissions and type, as defined by POSIX (P1003.1-1990). HPSS supports only regular files.

file family An attribute of an HPSS file that is used to group a set of files on a common set of tape virtual volumes.

fileset A collection of related files that are organized into a single easily managed unit. A fileset is a disjoint directory tree that can be mounted in some other directory tree to make it accessible to users.

fileset ID A 64-bit number that uniquely identifies a fileset.

fileset name A name that uniquely identifies a fileset.

file system ID A 32-bit number that uniquely identifies an aggregate.

FTP File Transfer Protocol

Gatekeeper An HPSS server that provides two main services: the ability to schedule the use of HPSS resources referred to as the Gatekeeping Service, and the ability to validate user accounts referred to as the Account Validation Service.

Gatekeeping Service A registered interface in the Gatekeeper that provides a site the mechanism to create local policy on how to throttle or deny create, open and stage requests and which of these request types to monitor.

Gatekeeping Site Interface The APIs of the gatekeeping site policy code.

Gatekeeping Site Policy The gatekeeping shared library code written by the site to monitor and throttle create, open, and/or stage requests.

GB Gigabyte (230)

GECOS The comment field in a UNIX password entry that can contain general information about a user, such as office or phone number.

GID Group Identifier

GK Gatekeeper

GSS Generic Security Service

GUI Graphical User Interface

HA High Availability

HACMP High Availability Clustered Multi-Processing - A software package used to implement high availability systems.

halt A forced shutdown of an HPSS server.

hierarchy See Storage Hierarchy.

HPSS High Performance Storage System

HPSS-only fileset An HPSS fileset that is not linked to an external filesystem.

IBM International Business Machines Corporation

ID Identifier

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

import An operation in which a cartridge and its associated storage space are made available to the HPSS system. An import requires that the cartridge has been physically introduced into a Physical Volume Repository (injected). Importing the cartridge makes it known to the Physical Volume Library.

I/O Input/Output

IOD/IOR Input/Output Descriptor / Input/Output Reply. Structures used to send control information about data movement requests in HPSS and about the success or failure of the requests.

IP Internet Protocol

IRIX SGI's implementation of UNIX

junction A mount point for an HPSS fileset.

KB Kilobyte (210)

KDC Key Distribution Center

LAN Local Area Network

LANL Los Alamos National Laboratory

LARC Langley Research Center

latency For tape media, the average time in seconds between the start of a read or write request and the time when the drive actually begins reading or writing the tape.

LDAP Lightweight Directory Access Protocol

LLNL Lawrence Livermore National Laboratory

local log An optional log on a remote system, for example on a Mover node under /var/hpss/log/HPSS.log. Remote system logs should be forwarded to the central logging system.

log record A message generated by an HPSS application and handled and recorded by the HPSS logging subsystem.

log record type A log record may be of type alarm, event, status, debug, request, security, trace, or accounting.

LRU Least Recently Used

LTO Linear Tape-Open. A half-inch open tape technology developed by IBM, HP and Seagate.

MAC Mandatory Access Control

managed object A programming data structure that represents an HPSS system resource. The resource can be monitored and controlled by operations on the managed object. Managed objects in HPSS are used to represent servers, drives, storage media, jobs, and other resources.

MB Megabyte (220)

metadata Control information about the data stored under HPSS, such as location, access times, permissions, and storage policies. Most HPSS metadata is stored in a DB2 relational database.

method A Java function or subroutine

migrate To copy file data from a level in the file's hierarchy onto the next lower level in the hierarchy.

Migration/Purge Server An HPSS server responsible for supervising the placement of data in the storage hierarchies based upon site-defined migration and purge policies.

MM Metadata Manager. A software library that provides a programming API to interface HPSS servers with the DB2 programming environment.

mount An operation in which a cartridge is either physically or logically made readable and/or writable on a drive. In the case of tape cartridges, a mount operation is a physical operation. In the case of a fixed disk unit, a mount is a logical operation.

mount point A place where a fileset is mounted in HPSS.

Mover An HPSS server that provides control of storage devices and data transfers within HPSS.

MPS Migration/Purge Server

MRA Media Recovery Archive

MSSRM Mass Storage System Reference Model

MVR Mover

NASA National Aeronautics and Space Administration

Name Service The portion of the Core Server that providesa mapping between names and machine oriented identifiers. In addition, the Name Service performs access verification and provides the Portable Operating System Interface (POSIX). name space The set of name-object pairs managed by the HPSS Core Server.

NERSC National Energy Research Supercomputer Center

NLS National Language Support

notification A notice from one server to another about a noteworthy occurrence. HPSS notifications include notices sent from other servers to SSM of changes in managed object attributes, changes in tape mount information, and log messages of type alarm, event, or status.

NS HPSS Name Service

NSL National Storage Laboratory

object See Managed Object

ORNL Oak Ridge National Laboratory

OSF Open Software Foundation

OS/2 Operating System (multi-tasking, single user) used on the AMU controller PC

PB Petabyte (250)

PFTP Parallel File Transfer Protocol

physical volume An HPSS object managed jointly by the Core Server and the Physical Volume Library that represents the portion of a virtual volume. A virtual volume may be composed of one or more physical volumes, but a physical volume may contain data from no more than one virtual volume.

Physical Volume Library An HPSS server that manages mounts and dismounts of HPSS physical volumes.

Physical Volume Repository An HPSS server that manages the robotic agent responsible for mounting and dismounting cartridges or interfaces with the human agent responsible for mounting and dismounting cartridges.

PIO Parallel I/O

PIOFS Parallel I/O File System

POSIX Portable Operating System Interface (for computer environments)

purge Deletion of file data from a level in the file's hierarchy after the data has been duplicated at lower levels in the hierarchy and is no longer needed at the deletion level.

purge lock A lock applied to a bitfile which prohibits the bitfile from being purged.

PV Physical Volume

PVL Physical Volume Library

PVR Physical Volume Repository

RAID Redundant Array of Independent Disks

RAIT Redundant Array of Independent Tapes

RAM Random Access Memory

reclaim The act of making previously written but now empty tape virtual volumes available for reuse. Reclaimed tape virtual volumes are assigned a new Virtual Volume ID, but retain the rest of their previous characteristics. Reclaim is also the name of the utility program that performs this task.

registration The process by which SSM requests notification of changes to specified attributes of a managed object.

reinitialization An HPSS SSM administrative operation that directs an HPSS server to reread its latest configuration information, and to change its operating parameters to match that configuration, without going through a server shutdown and restart.

repack The act of moving data from a virtual volume onto another virtual volume with the same characteristics with the intention of removing all data from the source virtual volume. Repack is also the name of the utility program that performs this task.

request A log record type used to report some action being performed by an HPSS server on behalf of a client.

RISC Reduced Instruction Set Computer/Cycles

RMS Removable Media Service

RPC Remote Procedure Call

SCSI Small Computer Systems Interface

security A log record type used to report security related events (e.g., authorization failures).

SGI Silicon Graphics

shelf tape A cartridge which has been physically removed from a tape library but whose file metadata still resides in HPSS.

shutdown An HPSS SSM administrative operation that causes a server to stop its execution gracefully.

sink The set of destinations to which data is sent during a data transfer (e.g., disk devices, memory buffers, network addresses).

SMC SCSI Medium Changer

SMIT System Management Interface Tool

SNL Sandia National Laboratories

SOID Storage Object ID. An internal HPSS storage object identifier that uniquely identifies a storage resource. The SOID contains an unique identifier for the object, and an unique identifier for the server that manages the object.

source The set of origins from which data is received during a data transfer (e.g., disk devices, memory buffers, network addresses).

SP Scalable Processor

SS HPSS Storage Service

SSA Serial Storage Architecture

SSM Storage System Management

SSM session The environment in which an SSM user interacts with the SSM System Manager to monitor and control HPSS. This environment may be the graphical user interface provided by the hpssgui program, or the command line user interface provided by the hpssadm program.

stage To copy file data from a level in the file's hierarchy onto the top level in the hierarchy.

start-up An HPSS SSM administrative operation that causes a server to begin execution.

status A log record type used to report progress for long running operations.

STK Storage Technology Corporation

storage class An HPSS object used to group storage media together to provide storage for HPSS data with specific characteristics. The characteristics are both physical and logical.

storage hierarchy An ordered collection of storage classes. The hierarchy consists of a fixed number of storage levels numbered from level 1 to the number of levels in the hierarchy, with the maximum level being limited to 5 by HPSS. Each level is associated with a specific storage class. Migration and stage commands result in data being copied between different storage levels in the hierarchy. Each Class of Service has an associated hierarchy.

storage level The relative position of a single storage class in a storage hierarchy. For example, if a storage class is at the top of a hierarchy, the storage level is 1.

storage map An HPSS object managed by the Core Server to keep track of allocated storage space.

storage segment An HPSS object managed by the Core Server to provide abstract storage for a bitfile or parts of a bitfile.

Storage Service The portion of the Core Server which provides control over a hierarchy of virtual and physical storage resources.

storage subsystem A portion of the HPSS namespace that is managed by an independent Core Server and (optionally) Migration/Purge Server.

Storage System Management An HPSS component that provides monitoring and control of HPSS via a windowed operator interface or command line interface.

stripe length The number of bytes that must be written to span all the physical storage media (physical volumes) that are grouped together to form the logical storage media (virtual volume). The stripe length equals the virtual volume block size multiplied by the number of physical volumes in the stripe group (i.e., stripe width).

stripe length The number of bytes that must be written to span all the physical storage media (physical volumes) that are grouped together to form the logical storage media (virtual volume). The stripe length equals the virtual volume block size multiplied by the number of physical volumes in the stripe group (i.e., stripe width).

stripe width The number of physical volumes grouped together to represent a virtual volume.

System Manager The Storage System Management (SSM) server. It communicates with all other HPSS components requiring monitoring or control. It also communicates with the SSM graphical user interface (hpssgui) and command line interface (hpssadm).

TB Terabyte (240)

TCP/IP Transmission Control Protocol/Internet Protocol

trace A log record type used to record procedure entry/exit events during HPSS server software operation.

transaction A programming construct that enables multiple data operations to possess the following properties: All operations commit or abort/roll-back together such that they form a single unit of work. All data modified as part of the same transaction are guaranteed to maintain a consistent state whether the transaction is aborted or committed. Data modified from one transaction are isolated from other transactions until the transaction is either committed or aborted. Once the transaction commits, all changes to data are guaranteed to be permanent.

TTY Teletypewriter

UDA User-defined Attribute

UDP User Datagram Protocol

UID User Identifier

UPC Universal Product Code

UUID Universal Unique Identifier

virtual volume An HPSS object managed by the Core Server that is used to represent logical media. A virtual volume is made up of a group of physical storage media (a stripe group of physical volumes).

virtual volume block size The size of the block of data bytes that is written to each physical volume of a striped virtual volume before switching to the next physical volume.

VV Virtual Volume

XDSM The Open Group's Data Storage Management standard. It defines APIs that use events to notify Data Management applications about operations on files.

XFS A file system created by SGI available as open source for the Linux operating system.

XML Extensible Markup Language