DeepSpace Storage Catalog Data Service Architecture Overview

Introduction

The Catalog Data Service (CDS) provides all services needed to connect any type of existing or future storage device to a requesting application on any host in the DS enterprise. To show the differences in implementation, at least at a high level, this document will describe the I/O processing on three different types of devices: an automated tape library, a Ceph Object Store, and a DeepSpace BlockOS device.

It should be noted the basic implementation is the same despite the significant differences in the form of the target storage architecture, this is not by accident. The DS Architecture is designed to take a lowest common denominator approach to interfacing to a storage target, allowing the integrator to create a simple "filter" to normalize the read/write/index functions of the storage target, while providing a site exit capability to take care of any external processing required to make the target ready for I/O.

Storage Presentation

The presentation of the storage target to the requestor is in the form of a "volumeset" abstraction whose creation includes rich attributes for long term systemic management of the data stored wherein (data curation).

Once the volumeset is presented to the requestor, it is provided for writing and reading files after authenticating and the CDS takes care of all low level, target specific functions. These include:

- The complete I/O path from the requestor to the correct target is created
- Device allocation is performed for removable media drive or mount point allocation
- Any device specific processing for removable media, power down, etc is run to bring the device online.
- The label processing (if applicable) is performed
- All volume spanning and needed rewind/hot-IO swap processing is taken care of automatically

A high level overview of this process will be described next, but will start assuming a volumeset has already been created with the appropriate attributes, these include as examples, technology type, location, ownership and permissions and a variety of expiration options based on either chronological or generational constructs. For additional details, please see either the dsvcreate command in the DeepSpace Command Reference Manual or look at the volumeset entry under the Configuration tab in the DeepSpace GUI.

Scratch Selection and Device Allocation

Volumesets are accessed through the dsvaccess command at the command line, or more typically by a client application making the request through the DS API. Existing DS client applications include the file system management, aka Systems Managed Storage (SMS), or the RDBMs data manager application. In any event, assume the volumeset exists and has been accessed by an authorized userid and refer to figure 1.

Catalog Service Object Read/Write Path

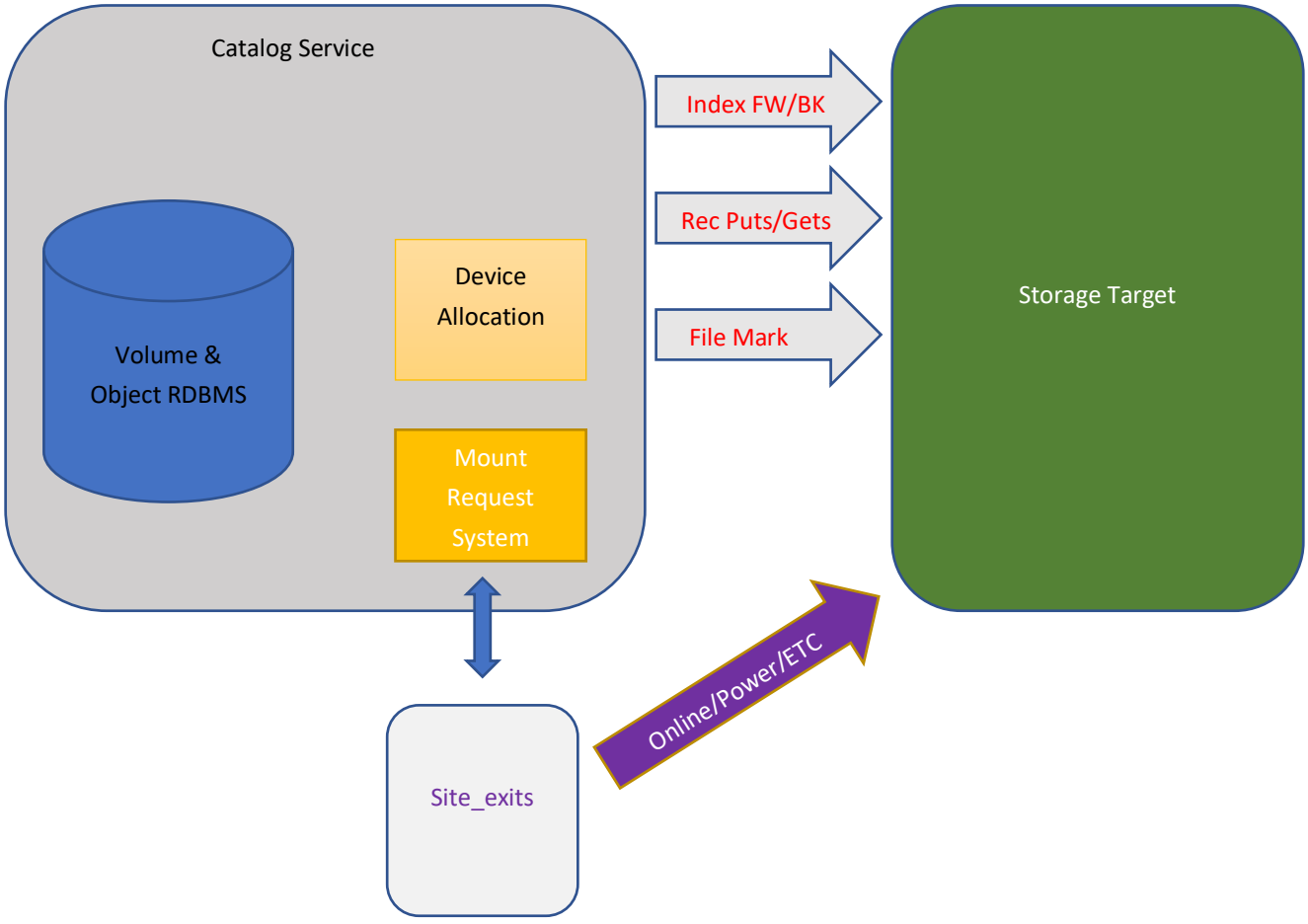

Figure 1. The catalog service encompasses the volume and file databases, mount request system and device allocation subsystem, it also includes a distributed I/O subsystem (not pictured).

The first time a volumeset is written to, a scratch volume needs to be selected as the first volume in the volumeset. These scratch volumes are drawn from pools, pools have user and group permissions. To simplify, let us assume everyone is using one pool, owned by root and called public, aka root/public. Further, the scratch volume must match the technology “type” keyword in the dsvcreate command, or from the “Type”

drop down list in the GUI. Example types would include LTO, RADOS (for Ceph) or one of the BlockOS drivers for conventional disk (DASD) or the ZBC type for host managed SMR disk drives.

To understand scratch entry, see dspsubmit in the Command Reference Manual, or see "Scratch Entry" under the "Operations" tab in the GUI.

Once a scratch volume of the correct type and pool are selected and the scratch volume id is added to the CDS Volume Record (VREC) entry for the named volumeset as part of the CDS internal housekeeping. The next step in writing a file is the device reservation system. The CDS will always perform each I/O operation to a mount point. If the type is LTO the mount point will be a regular device entry like /dev/st0. If it is a filter that translate the DS Sequential I/O stream into something suitable for another type of device, the mount point is typically one of many named pipes that act as an I/O gateway to the device unique filter. In either event, the device allocation system is responsible for finding a mount point that matches the type and matches a free mount point (or enters the wait queue) and finally in some cases makes necessary device naming translations.

On this last point consider a Fibre Channel attached tape drive on an un-zoned Storage Area Network. In this case a single tape drive may appear under many different /dev/st*N* device names depending on the system doing the I/O. The device reservation system understands the different naming conventions for each configured drive on each system and resolves the correct name to the correct drive on the host system the I/O is delegated to.

Mount Request Processing

Moving on to the point where both a scratch volume and a mount point is selected the next step is to implement the mount of the volume on the mount point and bring the device online and ready to write. Each mount point as mentioned is a specific I/O gateway to a real device or a filter that translates Sequential Access SCSI semantics to whatever commands the storage target needs to store and retrieve data. Let us look at 3 examples and contrast the logical requirements and how they are satisfied. But first we need to introduce the site exit facility called by the mount request system.

Site exits are a means of putting custom code into an otherwise COTS system to tailor the COTS solution to unique customer, hardware, and site challenges. In this case, we call a set of executable external programs, each is identified with a mount point, and each mount point has entries for scripts that bring the device ready as well as another that returns the device to the offline status, we just refer to them as mount and unmount scripts for each mount point. So each device type has a corresponding number of mount points configured to support the required number of parallel I/O streams being processed, and each has a mount/unmount site exit called with the following arguments which are provided by the CDS:

I - request_id

i – Volume Serial Number

v – Volume id

r - Rack

l - Location

t - Media_type

p - Pool

x - Capacity

d – Arbitrary device name

N – Mount point

R – device/FIFO

The above is an incomplete list, but generally only two or three of these command line arguments need to be processed to get the job done. The one attribute any site exit will need is the request id, which is passed to the site exit with 'I' argument. This is needed because once all of the steps necessary to prepare the device for I/O are complete, the site exit will have to call 'dsdone' and pass the request id integer to it in order to tell the CDS that it is ready to open the device or FIFO pointed to by the 'R' flag.

Automated Tape Library Example

```
$ cat /usr/local/ds/Librarian/site_exits
# Device_name Mount_exit Unmount_exit Offline_exit
#
# drive1 stkmount stkunmount none 0,0,10,0
# drive2 robo_mount.pl robo_unmount.pl none PV-122T,0
```

In the above example, there are site exits for 2 tape library transports. The site exits are different because the first example is for an enterprise class library that has a networked control platform that conforms to the IEEE Mass Storage Reference Model for the Physical Volume Repository (PVR) implementation. Without getting into details, the stkmount and stkunmount scripts are actually commercially produced binaries that implement the correct network protocol to effectuate mount/unmount actions with this type of library. The final 0,0,10,0 identifies the library control platform's device naming schema, in this case ACS, LSM, Panel, Drive and maps that to a specific CDS arbitrary device name, drive1. The stkmount and unmount binaries just need a Volume Serial Number (passed with the i flag) and this drive name 0,0,10,0 to make the mount happen.

The second line is for a standard SCSI attached library. The robo_mount.pl and robo_unmount.pl are PERL scripts that call the FOSS program mtx. This is the standard open source utility for media changer devices and its implementation is much more primitive then the previous example because media changer type libraries do not accept mount requests for tape volser's even when the tapes have barcode labels. Instead, media changers have a protocol that mounts a "storage slot" to a "data transfer element". Since we do not have the library catalog provided by a PVR type device in the example above, we leverage the CDS's own catalog to store the library and slot for each tape.

We do this by adding the CDS's rackid for each scratch volume submitted to the catalog with the library, slot string, then when the site exit is called, that rackid is passed to the PERL script which formats the command for mtx. The last entry, PV-122T,0 corresponds to the library device name and drive id as a "data transfer element" that specifically corresponds to the CDS notion of drive2 (library drive 0 on a library named PV-122T).

Ceph LibRADOS Example

The filter for doing I/O to Ceph does not need any action taken to prepare the Ceph cluster for a read or write, therefor the site exit only need to parse the request id when it is called and then execute dsdone with that request id. This is a no-op site exit and it tells the CDS the I/O is ready as soon as the volume (in this case an object) is ready to be opened for write.

BlockOS Example

The BlockOS filter in a sense does what a file system does with files, but it is based on SCSI commands rather than fopen()s. BlockOS understands where objects begin and end by block id, and vectors its read and write pointers based on SCSI sequential access skip commands to prepare the correct object or free space for read or write I/O.

In this case, assuming we are not implementing spin up/down processing to reduce operating power there would be no need for doing anything to prepare the device - and you might think a no-op site exit would be appropriate. However, because BlockOS devices are shared among clients in the DS enterprise using SCSI over RDMA, we do in fact have to create these iSER sessions between targets (the BlockOS LUN) and initiators (the DS client doing the I/O) dynamically as the need arises and then tearing them down when they are no longer needed (using the mount/unmount paradigm).

To do this we again use the rackid attribute of each volume to identify the iSCSI target device (either a SAS drive's WWN or a SATA drive's serial number + a partition or zone number in the case of an HM-SMR drive). Once we have the device parameters from the CDS rackid field indicating the device we need mounted, we then call the iscsiadm api to create a transient iSER session between the servers, and symbolically link the transient /dev/sd*N* iSCSI target to a /dev/ost*N* device before giving the CDS the all clear message with dsdone.

It should be noted this is a bit of a simplification, the session create/teardown is not necessarily a 1:1 correspondence to mount/unmount. In some situations, just leaving static iSER sessions left in place will work, in other circumstances a server can only support 256 initiator sessions on one host at a time, and the session pool needs to be managed below that level.

Label Processing and Volume Spanning

UNIX systems traditionally lack tape I/O subsystems, so tape processing would typically involve an application developer having to write all of the functions into the application code to handle low level tape I/O, especially the complexities of dealing with end of media handling and volume spanning. Mainframe computers, on the other hand, had a number of different labeling standards for wrapping user data with metadata as well as handling multi-volume data sets. Label processing enhances transportability of data and recovery of lost meta data so well that it is always used in DeepSpace regardless of the storage platform being written to.

The relevant label processing standards are the IBM and ANSI tape label formats, which we support. Both of these standards allow a single file, or a collection of files to span up to 10,000 volumes. However, these standards are based on short file names and a maximum of 10,000 files on a spanned volumeset. We modified the ANSI label type and renamed it DSCM label type to accommodate 256 character file names and over 4 billion files per volumeset.

Labels consist of Volume labels at the beginning and end of each volume, as well as file labels at the beginning and end of each file. All of the metadata associated with the volumeset, as well as each file in the volumeset, is included in these labels.

Volume spanning deals with reaching the end of volume in one of two ways depending on the capabilities of the target storage device. Ceph's LibRADOS API requires the client application write no more than 125MB of data in a single object, so we need to enter Ceph volumes (object storage targets) as having a fixed capacity of 125MB when we enter them as scratch volumes.

When writing to a fixed size volume like Ceph, the CDS tracks the free space as data is written until the free space is nearly exhausted, and then the partial file is closed. The following "file label" marks the partial file of segment 1 of a multi-segment file along with the rest of the file metadata. Then the closing "volume label" is written and the volumeset is spanned into the next scratch volume.

Tape volumes on the other hand never have fixed volume sizes because they all incorporate data compression, and compression rates can never be known ahead of the write. Tape drives instead give the device driver a hint as to when they are getting close to End of Media (EOM) and let the calling application know of this by posting a "short" write return at the early warning indication of EOM. This triggers the end of volume handling as in the above example. The tape model has the advantage of not requiring the media's capacity to be known when the volumes are added as scratches, the catalog service just knows how to use all of the capacity for writing data, until there is just enough space left over for label processing to logically close the volume at the end.

BlockOS devices follow the tape model for notification of EOM incorporated into the driver filter. When a BlockOS device is initialized, its geometry (and zone configuration if SMR) is read during the device initialization which creates the object map that is written to the device. The object map and write pointer information is then used by the BlockOS driver to provide the EOM warning back to the CDS exactly as in the tape example and the label processing is also the same. This has the distinct advantage of allowing the BlockOS disk farm to consist of any arbitrary collection of disk drives, each with any arbitrary capacity.

DeepSpace will just use and scratch capacity within the user's scratch pool until it is all exhausted and every device is at 100% utilization.

On-Device Data Structure

As previously noted, all DeepSpace Volumesets are written with standard label processing regardless of the storage target implementation. This helps secure the integrity of the data by allowing volumes to be validated before data is read, or overwritten. This also provides a recovery path in the event of catastrophic loss of the CDS catalog information stored in the customer's RDBMS.

For information on catalog recovery using label processing, see the DeepSpace Command Reference manual under the "dsvdisplay" command with the "report=scan" option.

Standard label processing is only applicable for IBM/ANSI/DSCM label types. The other label options supported by DeepSpace are intended for interoperability purposes, for example CPIO and tar support provide the ability to create tape sets that can read on any system, albeit with very limited metadata. These other formats generally do not support data spanning a single volume.

ANSI and IBM label types per their respective standards use a combination of 80 byte records at both the volume and file transitions, whereas the current (V1.0) DSCM label type uses a single 512 byte label for each.

Space Reclamation/Compaction

As indicated in the figure above, files are encapsulated with file metadata at both the beginning and end of each file. Part of the metadata includes a "segment" number that always begins with a 1. If the last file on the volume will not entirely fit, as is the expected case, the partial file is closed and the same label from the beginning of the file is rewritten at the end, followed by the volume closing label. The next scratch volume will get the same file label written to the first file write, but with the segment number incremented to 2.

This process can continue for up to 10,000 volumes, so that the file size of a single file can be as large as the size of the underlaying volume it is written to multiplied by 10,000. This should only be a concern to archiving to Ceph since RADOS volumes are limited to 125MB each, providing for a maximum volumeset and/or file size of 1.25TB. If you need to use very large files and Ceph, you will need to use the next revision of the DSCM label type which is in process and moves from a 512 byte to a 1024 byte label format with support for over 4 billion volumes in a volumeset and many other improvements including "out-of-order" processing and a URL embedded in the standard label to point to an external volume catalog for user defined metadata outside of the on-media record.

Free Space Collection/Compaction

The inherent assumption for any storage target is that it exposes a very minimal profile of capabilities, certainly these do NOT include the ability to update any record “in place”. In other words, every write is an append, and free space collection or “compaction” is only performed by reading the source volume one file at a time and rewriting it entirely to another target volume while skipping over and “expired” or stale data.

To understand how this works without disrupting the referential integrity of the data, it is necessary to understand Generation Data Groups (GDG). GDG’s apply to files as a means of providing version control, or the ability to roll back a file to any prior generation of the file.

They also apply to volumesets, and they provide the ability to have multiple generations of a volumeset. The significance of GDG’s in this respect lies in how generations are referenced in absolute or relative terms. As an example, if I create a volumeset it gets a generation number of 0. The second time I create the volumeset with the same name, it will increment to generation 2, and on until the maximum count of 9999 is reached and it wraps.

When a volumeset is opened without a specific generation reference, the last generation in existence is returned. Prior versions can be specified explicitly with a vsname:Gnnnn, where nnnn is replaced with a generation number between 0 and 9999. With relative addressing, the prior generation of a volumeset can be reference with a negative sign and an integer indicating how many versions back, i.e., vsname:G-1 indicates the second to last version of volumeset “vsname”.

Returning to the problem of compaction, we use the GDG construct to maintain referential integrity by creating another generation of the source volumeset that needs to be rewritten without the expired files it contains. The new volumeset and the original volumeset will coexist until the reclamation process completes by copying all non-expired files from the source to the target.

This reclamation process is a background process, it can be delegated to any DS client, and it will be pre-empted by any other process with a higher priority that needs to access data on the source volumeset. After the completion of the copy from source to target of all non-expired files, the reclamation process will simply scratch the source volume, returning it to its original scratch pool.

Properly coded DS client applications will always request the volumeset a needed file resides on by the generation number that the file was originally archived to. If that volumeset:generation no longer exists due to reclamation having been performed, the client application will respond to the error by reformatting the volumeset request without the generation specifier, automatically replacing the original volumeset with its updated replacement, and referential integrity is fully maintained.

Please send Comments/Corrections to scranage@deepspacestorage.com