Date: **December 13, 2012**	**High Level Design – Versioning Object Storage Device (VOSD)** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

Table of Contents

Revision History

Date	Revision	Author
Feb. 21, 2013	1.0	Paul Nowoczynski & Johann Lombardi
Mar. 03, 2013	1.1	Paul Nowoczynski
Mar. 29, 2013	1.21 (Comments and Responses)	Paul Nowoczynski

Introduction

This design documents details the integration of the Versioning Object Store into the existing Lustre Object Storage device.

Definitions

- **VOSD** – "Versioning Object Storage Device"
- **HCE** – Highest Committed Epoch.
- **HCE+1** – The next epoch on the commit horizon.
- **> HCE + 1** – Epochs which are further on the commit horizon. Writes to such epochs are generally directed into the Intent Log.
- **OI** – Object Index. An object index is a stateful Lustre OSD structure which enables the mapping of a Lustre based identifier to a file or directory in the underlying filesystem (such as ldiskfs or zfs).

Changes from Solution Architecture

During this last quarter it was concluded that writes into a VOSD on behalf of epoch *HCE+1* may be placed directly into the shard dataset without requiring mediation by the intent log. This technique does not violate VOSD semantics and hence, represents a significant breakthrough by providing means for avoiding intent log replay overhead in some situations.

Another method for addressing intent log and snapshotting overhead in VOSD was devised in the previous quarter. The method specifically addresses situations where applications may pathologically issue epoch commits such that the VOSD intent log flattening and snapshot overhead become dominant. Working in conjunction with the metadata server, which acts as a 2-phase commit coordinator, the VOSD may be instructed to 'flush' rather than 'commit' should the MDS determine that the commit frequency on the container is too high. Unlike a 'commit', which requires synchronous intent log flattening and a shard snapshot, 'flush' is a suggestive instruction which compels the VOSD to begin flattening an intents stored on behalf of the given epoch. This method is described in a later section called *DAOS Transaction Aggregation*.

Overview

To a large extent, VOSD functionality may be achieved by exposing functionality within existing copy-on-write filesystems. More specifically, the snapshot and rollback provide a basis for the VOSD framework in such a way that does not require DAOS to completely reinvent, or develop from scratch, a capable CoW filesystem. During this last development quarter the VOSD team decided to move forward with the ZFS filesystem for providing the basis of VOSD over btrfs. The reasoning behind this decision lied with ZFS's proven stability and its existing integration with Lustre. Btrfs was compelling due to its open-source backing and GPL licensing but ultimately proved too risky given the scope and time frame of the Fast Forward project. In the long term btrfs may prove to be the dominant CoW filesystem. Should this be the case in the future, we expect that the techniques used to implement a ZFS-based VOSD will be largely commutable to btrfs.

Starting with the current ZFS-based Lustre OSD provides a strong foundation for this work. For instance, the current ZFS Lustre OSD already fully supports I/O operations into ZFS volumes as well as Lustre specific functionality such as object index support. Despite the fact that ZFS on Lustre has proven to be successful, the means by which Lustre makes use of ZFS are relatively simplistic when considering the overall feature set of ZFS. The successful implementation of VOSD will require these functionalities to be exposed within the Lustre OSD framework in a manner which supports DAOS operation.

Establishing a VOSD within the context of ZFS and Lustre requires a moderate degree of modifications to both. The Fast Forward design team is going to great lengths to find a reasonable balance between the architectural integrity of VOSD and integration into the existing implementations (of ZFS and Lustre).

Characterization of Lustre Modifications

The implementation of VOSD improves upon the Lustre OSD by advancing the internal filesystem framework such that fundamental ZFS features, such as snapshot and rollback, may be exposed to DAOS. This work extends the current ZFS OSD implementation by allowing for more native ZFS capability to be exposed. Currently, the ZFS OSD implements the same API as the other OSD filesystem, ldiskfs (which is non-CoW).

The modifications required for VOSD align structurally to today's OSD implementation which supports multiple devices, or OSTs, running simultaneously. The difference between the current OSD stack and VOSD requirements lies in the granularity of the filesystem volume management. The current stack considers a 'device' to be an entire filesystem volume. This is case for both ldiskfs and ZFS. For the latter, a device consists of an entire ZFS pool. VOSD modifications will allow for a finer volume management granularity such that individual ZFS filesystems, of the same zpool, may be managed as devices within VOSD. Increasing the pliability of the underlying OSD devices will provide the necessary dynamics for creating, accessing, and destroying DAOS shards.

The current implementation supports the on-demand instantiation of devices, which can be utilized for this effort. Today, on-demand instantiation in the current implementation is done at the behest of the Lustre administrator primarily for starting and stopping individual OSTs on an I/O server. Modifications done on behalf of VOSD would allow for DAOS shards to be treated as devices and for devices to be loaded on-demand at the request of the application versus the command of the administrator.

There are several advantages in mapping DAOS shards to Lustre OST devices which should lead to a more rapid stabilization of the VOSD implementation. First, the Lustre OSD object index (OI) and OI cache structures remain unchanged. Alternately considered implementation methods required a DAOS object's epoch to be passed through the entire depth of the OSD stack so that version resolution could take place in the OI cache. The chosen method inherently separates versions by maintaining the current OSD model of an OI cache per device. This alleviates the need for major changes to the internal OI cache by insulating the epoch parameter from the lower layers of VOSD. It is our intention to use the current OSD implementation in this manner such that less time will be spent adapting Lustre internals and more effort may be focused on the VOSD itself.

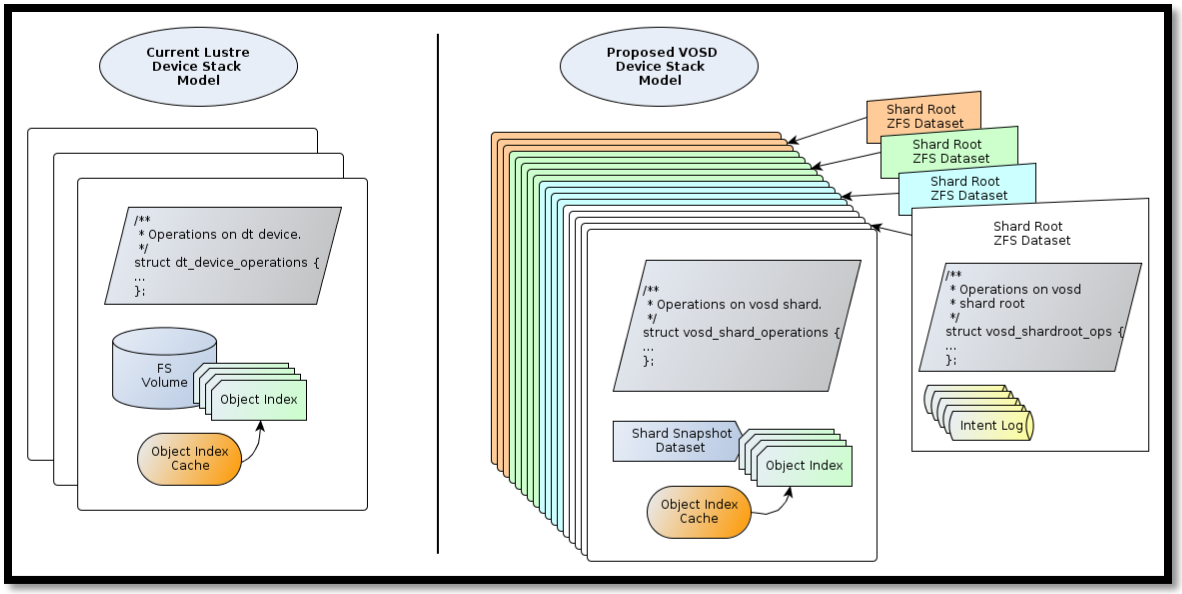


Figure 1 - Current Lustre Device Stack vs. VOSD

Figure 1 compares the current Lustre OSD stacking and the proposed, per-ZFS dataset, stack to be utilized by VOSD. The current device model is transferred largely intact to VOSD while an upper layer is inserted (Shard Root) to manage shard version presentation and intent logging. VOSD will utilize a much larger number of stack entities since every shard will expose a stack for every existing and accessed version of that shard. The diagram represents this by showing multiple stack entities per shard (one per snapshotted version).

Modifications for Supporting Intent Logging

Supporting the intent log functionality requires substantial engineering of ZFS and subsequent integration into VOSD. For instance, intent log writes must be seamlessly redirected from the upper layers of the Lustre OSD into their respective intent log objects. Furthermore, writes directed to the IL must be translated from pure Lustre writes to the intent log format.

Specification

DAOS Shards

Within a VOSD ZFS pool, a shard is represented by three different ZFS datasets. These are described immediately below.

1. Shard Root

The shard root is a child of the ZFS master object set (MOS) and is named according to the shard's sequence number as provided by the MDS. The shard root itself stores:

Intent logs for epochs > HCE+1 (Description to follow)

A shard configuration file(s) storing information about the current state of the shard, this includes:

- Current stable version (see item #2 below)
- Pointer to staging area (see item #3 below)
- Reference to HCE+1 snapshot if any (in case of failure after the 1st phase of commit)

2. HCE Snapshot

The second dataset composing a shard is the HCE snapshot which is the current stable version of the shard. This dataset will not exist for newly created or otherwise unused shards. By definition, per the VOSD specification, the HCE of a shard is read-only and as follows, reads which do not explicitly specify a previously committed epoch, are served from this dataset.

3. Staging Dataset (HCE+1)

The staging dataset is where writes for epoch HCE+1 land. It is a staging area where the next stable snapshot is groomed.

The staging dataset accumulates updates to HCE+1 through two methods. The first method is through direct write. By mandating the snapshot of HCE on commit, VOSD may allow for writes to HCE+1 to be applied directly into the staging dataset. As

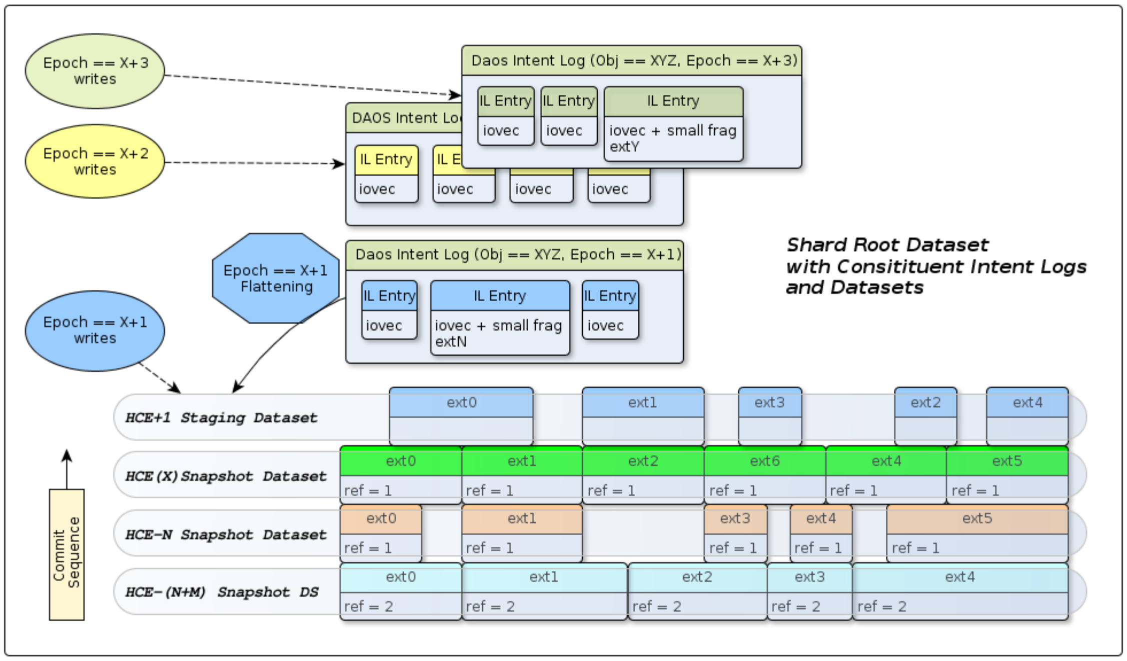


Figure 2 - Shard Root with Intent Logs and Datasets

described earlier in this document, this is an important technique for allowing the intent log to be bypassed in many circumstances.

The second method the staging dataset accumulates updates is through intent log flattening. It should be noted that DAOS does not guarantee ordering of writes within a single epoch. Therefore, the staging data set may be prepared through both methods - direct write and intent log flattening - simultaneously.

On successful commit, the staging dataset will be snapshotted and become the new stable version. While on failure, this dataset may be completely repealed leaving HCE intact.

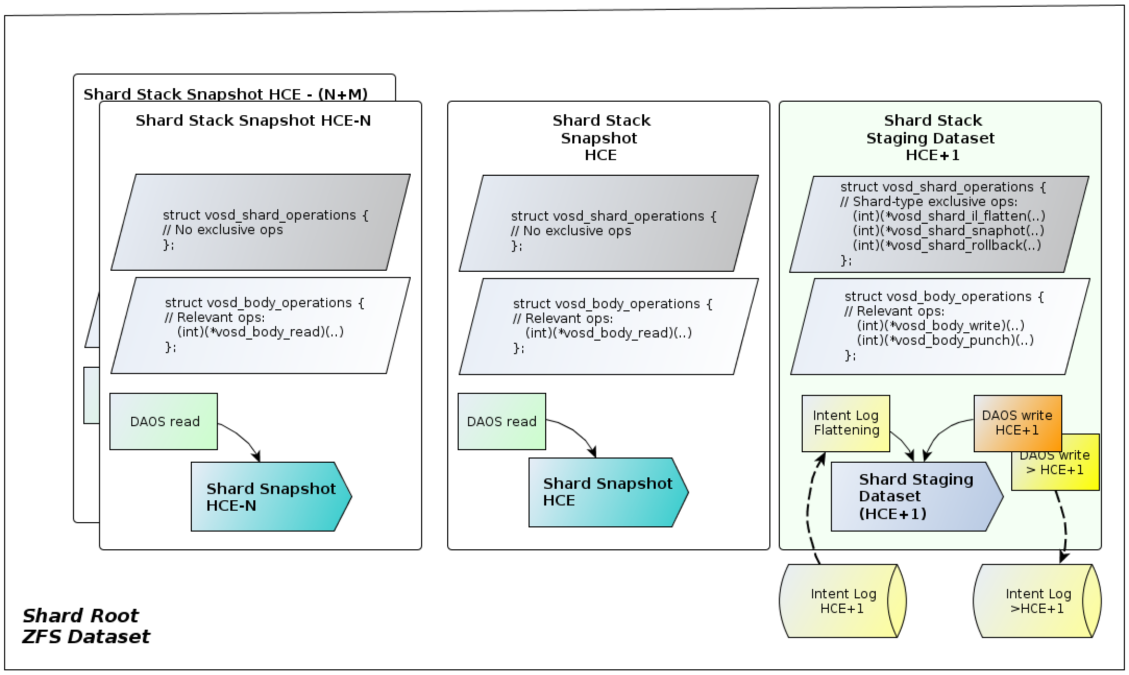

Shard Stack Snapshot HCE - (N+M)
Shard Stack Snapshot HCE-N
struct vosd_shard_operations { // No exclusive ops };
struct vosd_body_operations { // Relevant ops: (int)(*vosd_body_read)(..) };
DAOS read
Shard Snapshot HCE-N
Shard Stack Snapshot HCE
struct vosd_shard_operations { // No exclusive ops };
struct vosd_body_operations { // Relevant ops: (int)(*vosd_body_read)(..) };
DAOS read
Shard Snapshot HCE
Shard Stack Staging Dataset HCE+1
struct vosd_shard_operations { // Shard-type exclusive ops: (int)(*vosd_shard_il_flatten(..) (int)(*vosd_shard_snaphot(..) (int)(*vosd_shard_rollback(..) };
struct vosd_body_operations { // Relevant ops: (int)(*vosd_body_write)(..) (int)(*vosd_body_punch)(..) };
Intent Log Flattening
DAOS write HCE+1
DAOS write > HCE+1
Shard Staging Dataset (HCE+1)
Intent Log HCE+1
Intent Log >HCE+1
Shard Root ZFS Dataset

Figure 3 - Shard Stacks

Routing DAOS Requests to the Appropriate Shard Instance

As described in Figure 1, the Lustre storage service will be 'shard-aware' making it possible to use the shard's sequence number and the request's epoch number as parameters for selecting the appropriate shard stack. This design alleviates the need for the current Lustre OI layer to be aware of the epoch number thus insulating these parts of the stack from disruptive API changes. The shard root will provide routines for lookup of the available snapshots so that the Lustre storage service can initiate on-the-fly device configuration for requests which route to non-configured shard stacks. Once configured, a shard stack will operate in manner quite similar to the current Lustre ZFS OSD where each shard stack will be responsible for managing access to its underlying objects.

Shard stacks differ from the current Lustre devices in that only one shard stack per-shard (the staging dataset) may be accessed for direct writes. All other shard stacks are configured as read-only according to the DAOS coherency specification.

Routing to Intent Logs

Additional means must be provided for routing write requests to > HCE + 1. Here a special shard stack will be derived from the shard root's corresponding intent log object. This 'stack' will only support write or punch functionality and will not deal with any sort of OI functionality. Its only purpose is to stage shard updates through a mechanism which provides a uniform API to the upper layers of the Lustre storage device. In other words, the upper stack layers of the Lustre storage device are unaware that writes are being directed to an intent log within the shard.

Intent Log

An intent log is a specialized structure which is purposed for buffering writes into epochs beyond the commit horizon. Specifically, this means a write to any epoch > HCE + 1 must be staged into an intent log. If required, intent logs are manifested on a per-shard, per-epoch basis.

Intent logs are children of the shard root and thus do not reside in the shard root's child datasets. This separation is purposely maintained to prevent the log's temporal contents from being permanently stored in an HCE or named snapshot. The VOSD aims to prevent this sort of pollution within a shard's datasets.

An intent log is composed of at least a single ZFS dnode residing in the shard root. This primary dnode is responsible for storing a log of operation descriptions. The components of the intent log structure will be as follows:

```
struct vosd_intent_log_entry {

        enum vosd_il_op op_type;   /* write, punch, kv create / remove,etc. */

        daos_objid_t obj_id;       /* which DAOS object */

        off_t offset;              /* location of write or punch */

        size_t size;                        /* size of write */

        size_t il_data_len;        /* additional log data consumed by entry */

        union {

                zfs_blkptr_t blk;/* copy of zfs blkptr contents (zero copy) */

                char data[];     /* small embedded fragments or kv contents */

        };

};
```

One method under consideration for implementing the IL relies on a second dnode which is responsible for storing complete blocks which were delivered by DAOS clients. The purpose of this secondary log structure is to prevent the need to rewrite these blocks into the staging dataset. On flattening, VOSD will relocate the block pointers of these blocks into the corresponding DAOS object as it exists in the staging dataset. This capability is a deviation from the ZFS ZIL, which copies all logged data contents into its respective dataset. One advantage of this method is that logged blocks may be easily reclaimed in the event of a failure or rollback. Additionally, this method guarantees that the ZFS I/O pipeline has manicured the blocks in preparation for their association with a file. This means that the blocks' checksums may be reused without need for recalculation.

A second potential method for implementing the IL utilizes the block allocator directly. This mode utilizes the log structure detailed above but does not use a secondary dnode to hold complete blocks. Instead, blocks are allocated directly from the DMU and referenced from the intent log entries. This method must be mindful of leaking blocks on rollback or commit failure.

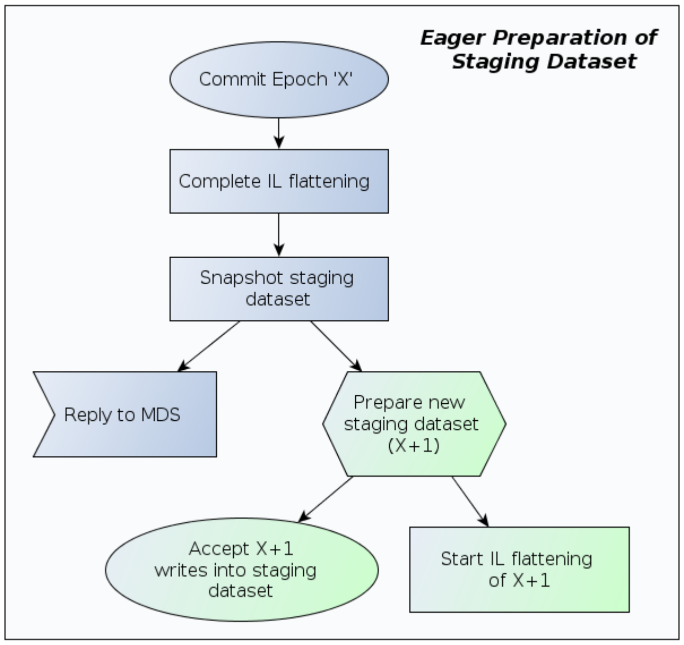
As the implementation of the intent log proceeds these methods will be weighed to determine which is more suitable.

Intent Log Flattening

Each entry self describes its type and size in manner which is consistent with most filesystem journal implementations. For DAOS writes smaller than a single ZFS block, the contents of the write are appended to the log entry and the intent log data length structure member is set to mark the length of the write. This length value also allows the next intent log entry to be easily located amidst variable length data. In the case of full block writes, the intent log entry stores a copy of the ZFS block pointer information so that the block may be incorporated into the destination dataset with zero-copy.

Flattening of the intent log is a relatively simple procedure which is prompted by the MDS in preparation for epoch commit. Upon receiving notification from the MDS, VOSD first locates the IL associated with the commit operation and then begins to iterate over the intent log entries encoded in the log[3]. Small writes are issued through the standard dmu_write() interface in a manner similar to ZFS's own intent log (ZIL). While large writes are processed without dmu_write() but rather by block pointer reassignment.

Block pointer reassignment must be cognizant of the ZFS internal structures needed for maintaining garbage collection and space accounting.



Eager Preparation of Staging Dataset
Commit Epoch 'X'
Complete IL flattening
Snapshot staging dataset
Reply to MDS
Prepare new staging dataset (X+1)
Accept X+1 writes into staging dataset
Start IL flattening of X+1

Figure 4 - Eager Preparation of Staging Dataset and Intent Log Flattening

Eager Preparation of the Staging Dataset to Reduce Commit Latency

Writes from DAOS clients are directed per their epoch to their respective shard root structure. When the epoch number of the incoming write is > HCE+1, the write is directed to its respective intent log. This element of the VOSD design has been in place

[3] Note that underlying DAOS objects and their respective OI structures are *created-on-write*. The create-on-write construct is carried on within the intent log replay such that intent log replay actions may implicitly create new DAOS objects.

since the beginning stages of the design.

However, in the case where a set of incoming writes' specified epoch is HCE+1, the deltas may be applied directly to the staging dataset without mediation by the intent log. This optimization is meant to reduce intent log replay for HCE+1 writes.

Figure 3 shows a shard root dataset with its sub-datasets and intent log structures. The diagram shows ingest from simultaneous epochs and flattening activity which occurs while the write window to HCE+1 is still open. This is another type of optimization which is meant to reduce commit latency by ensuring that intent log flattening begins at the earliest possible moment.

The Figure 4 flowchart shows how staging dataset preparations begin immediately following the commit of the previous epoch and facilitate eager flattening. Once epoch X+1 has been promoted to the staging dataset, writes from DAOS clients and intent log replay activities may occur simultaneously within the staging dataset. This is possible due to the transactional specification of DAOS which guarantees that updates within a given epoch are non-conflicting.

Epoch Commit and Rollback

Epoch commit is performed by a two phase operation which is coordinated by the MDS. Typically, a commit is executed by an application once all of the writes composing the given epoch have been submitted to DAOS. The commit request is presented to the MDS on behalf of an individual container. The MDS is aware of the container's layout and proceeds to setup a collective operation which encompasses the container's set of shards. Upon receiving a Phase 1 commit request, the VOSD prepares the staging dataset to be snapshotted. At this point, the VOSD may safely assume the application has closed the update window and that the epoch submitted for commit, and all lesser epochs, have quiesced. Before responding with a completion message to the MDS, VOSD must ensure that the any intent logs involved in the commit operation have been flattened. If the commit epoch is > HCE+1, intent logs of epochs falling between HCE+1 and the commit epoch must be flattened sequentially to preserve the cross-epoch ordering semantic.

Once flattening operations have completed, the VOSD snapshots the staging dataset and replies to the MDS. At this point the VOSD must ensure that the previous HCE snapshot remains intact until the MDS makes a final decision on the pending commit. This is done to ensure that rollback to a previously viable container state is possible.

The MDS initiates the latter half of the two phase commit after it has collected responses on behalf of each container shard. At this time, the MDS may choose to abort the transaction if a shard has reported an error or the owning OST has been deemed faulty by the gossip network[4].

Aborting an Epoch Due to Commit Failure

In the event where one or more shards fail to commit, the MDS must interact with both the DAOS clients and servers. Client interaction involves the revocation of the write capability which had been established at container open. This side of the abort operation guarantees the application will not assume that a given version dependency has been

[4] In this situation it's likely that the MDS had not received a response from the failed VOSD.

satisfied. The capability revocation is vital for notifying clients of the problem since the applications are programmed to use a single container opener who broadcasts a copy of the handle to his peers. Such a scheme makes it impossible for the storage system to inform every client who may access the container via the global handle. Therefore, capability revocation is necessary to ensure that no client may progress transactionally when an epoch has been aborted.

The MDS must also inform the responsive VOSD's to remove the snapshot associated with commit attempt. This step is accomplished using a collective composed of the remaining viable VOSD's listed in the container layout. Upon receiving the abort message each VOSD removes the snapshot created by the commit attempt. In situations where the container was consuming updates from concurrent epochs, the cleanup effort may go beyond the removal of this latest snapshot. Epochs, whose deltas had been gathering within their respective intent logs or the newly promoted staging dataset, must subsequently be aborted and their contents removed.

Committing an Epoch

If the MDS has decided to proceed with the commit a second collective will be issued instructing the constituent VOSD's to finalize commit activity for the given epoch. Since rollback to the previous HCE is no longer a possibility, removal of its snapshot is permitted if no open references to it persist.

DAOS Transaction Aggregation

DAOS applications which request frequent commits, by default, may invoke ZFS snapshots at rate which is not germane for high performance I/O to the underlying ZFS pool. This is because the snapshot forces synchronous activity at the ZFS layer. To deal with this phenomenon the DAOS metadata server may convert an application's commit request into VOSD *flush* operation which, in turn, is handled differently by the VOSD. The result of this behavior is that application commit actions may not result in a container snapshot. Hence, an application commit is merely notifying the storage system that the update window for that epoch is closed (since a snapshot is no longer guaranteed). Depending on the decision taken by the metadata server, with consideration to the application's commit frequency, a container snapshot may or may not be created.

When VOSD receives a flush operation, it proceeds as if a normal commit has been instantiated with the exception of the snapshot itself. Intent log flattening for the *flushed* epoch must complete before the VOSD replies to the MDS. Once this flattening has completed the staging dataset, which would have been snapshotted on a typical commit, is ready to accept direct writes from DAOS applications and begin intent log flattening for the following epoch.

The MDS collects the results from the *flush* collective but does not acknowledge the commit operation to the DAOS application. Acknowledgement is delayed until the next snapshot is taken to ensure the application's semantic integrity. The asynchronous nature of the DAOS clients' event queues allow for the applications to progress without needing to block on a typical commit operation. Since commit no longer guarantees a container snapshot, the DAOS application may call daos_epoch_commit() with a 'sync' flag which overrides any MDS decision to convert a commit into a flush, thus restoring some control to the application regarding explicit snapshotting.

While this approach spares the VOSD from snapshotting too frequently, it comes with the caveat that the staging dataset now contains updates from two or more epochs – making it impossible to separate the deltas applied on behalf those epochs.

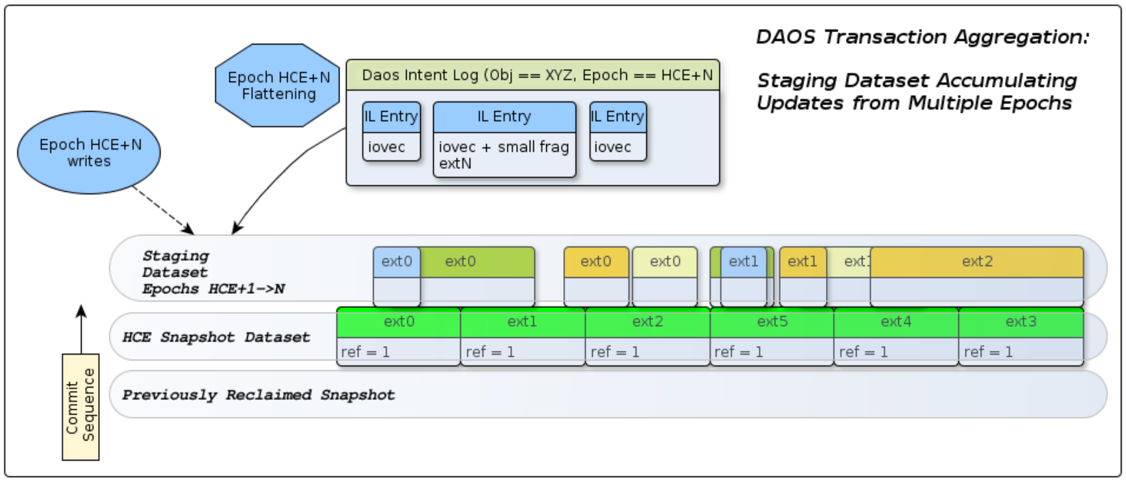


Figure 5 - Accumulation of Writes from Multiple Epochs into the Staging Dataset

API and Protocol Additions and Changes

- **`static int vosd_shard_snapshot(struct vosd_shard_dev *vsd, daos_epoch_t epoch)`**

  `Create a ZFS snapshot to transition the provided vosd_shard_dev from HCE+1 to HCE.`

- **`static int vosd_shard_il_flatten(struct vosd_shard_dev vsd*, daos_epoch_t epoch)`**

  `Issued after the commit of the previous epoch, this call handles the process of replaying writes from the intent log into the staging dataset. In some cases vosd_shard_commit() may block on the completion of this call (and a subsequent call to vosd_shard_snapshot()). The epoch parameter should correspond to the next epoch in the commit sequence.`

- **`static int vosd_shard_flush(struct vosd_shard_dev *vsd, daos_epoch_t epoch)`**

  `Where vosd_shard_dev is a pointer to the staging dataset, call denotes that the update window associated with 'epoch' has closed. Eager flattening of epoch+1 and acceptance of epoch+1 direct writes may now be accepted into the staging dataset.`

- **`int vosd_shard_commit(struct vosd_shard_dev vsd*, daos_epoch_t epoch, int flags)`**

Vosd_shard_commit() is issued by the MDS on behalf of a DAOS application. Depending on various circumstances, the MDS will decide whether or not to instruct the VOSD to create a snapshot for the shard.

Either VOSD_SHARD_FLUSH or VOSD_SHARD_SYNC will be set in the 'flags' parameter to specify the MDS's snapshot instruction. In the case where VOSD_SHARD_FLUSH is issued, VOSD will issue vosd_shard_flush() which will flatten the appropriate intent log(s) but will not call snaphot. Otherwise, both intent log flattening and snapshotting of the vosd_shard_device will occur through vosd_shard_snapshot().

- **int vosd_shard_rollback(struct vosd_shard_dev *vsd, daos_epoch_t epoch)**

  This call is issued when the MDS instructs the VOSD to abort a commit. The epoch parameter tells the VOSD which snapshot to target as the rollback point. This provided epoch number should correspond to a snapshotted version in the shard. If this is not the case, the container has faulted and may be corrupt.

- **struct vosd_shardroot * vosd_shardroot_create(daos_shard_seq_t seq)**

  Create the necessary ZFS structures needed to compose a shard.

- **int vosd_shardroot_destroy(daos_shard_seq_t seq)**

  Destroy shard datasets.

- **int vosd_shardroot_il_create(struct vosd_shardroot *vsr, daos_epoch_t epoch)**

  Intent logs reside in the shard root to prevent them from being involved in shard snapshotting activity. This call instructs the shard root to create an intent log object for the given epoch.

- **int vosd_shardroot_il_destroy(struct vosd_shardroot *vsr, daos_epoch_t epoch)**

  Once flattening has completed, or a dependent transaction has been aborted an intent log may be destroyed.

Open Issues

The exact implementation path for the intent log has yet to be determined.

Risks & Unknowns

Container Snapshot Frequency

At this point it is unknown what impact a high container snapshot frequency will have on the VOSD. Therefore, we have begun to put in place measures which will allow the snapshot frequency threshold to be configurable parameter.

Implementation Complexity of the Intent Log

Modifying ZFS internals to support zero-copy behavior for the VOSD intent log will require complex modifications to the ZFS layer. It is likely that issues may be found which impede progress in this area. This had come to light after our decision to use ZFS and the subsequent analysis which revealed that the ZFS intent log (ZIL) does not utilize zero-copy operation for full block writes.

Comments and Responses

Question Regarding ZFS Implementation Complexity

***Do we have any estimation as to the level of complexity of changes to ZFS? While this is research, it would be good to have some notion of how required changes within both ZFS and Lustre will be able to make it upstream at some point*. [GMS] -- Page 2, Paragraph 2**

Modifications outside of the Zero-Copy Intent Log implementation are considered attainable and therefore, low risk. The reason for this is that no new ZFS internal APIs are needed to implement the core VOSD functionality. The team is leveraging existing ZFS APIs for managing ZFS datasets to implement checkpointing and rollback functionality.

Routing of Writes Belonging to Epochs >HCE+1

(Note both question are addressed with the answer below)

***Are all writes destined for >HCE+1 always directed to the intent log or are direct "extent writes" supported for >HCE+1?* [GMS] – Page 5, Diagram 3**

***Same question above, all writes to >HCE+1 are directed to the Intent log?* [GMS] – Page 5, Paragraph 4**

At the point of the document where this comment was presented we had described the scenario where direct writes for >HCE+1 were allowed into the staging dataset. To answer this question directly, the answer is 'yes'. The section titled 'DAOS Transaction Aggregation' describes the condition in which direct writes for >HCE+1 are able to occur.

Handling of Conflicting Writes within an Epoch

***I assume there is no issue with writes within the intent log conflicting with writes already applied directly to the staging dataset? That is to say, the last applied write will "win".* [GMS] – Page 8, Paragraph 2**

Correct. The last write *applied* will win (not the last write issued). DAOS assumes that applications are properly managing their own consistency within an epoch. Therefore, if an application issues a conflicting or overlapping write, DAOS makes no guarantees which write will appear in the object.

***This may be better stated that DAOS makes NO guarantees as to ordering of conflicting writes within a single epoch.* [GMS] – Page 8, Paragraph 3**

Noted. Thank you.

Question about DAOS Transaction Aggregation and Snapshot Bypass

***So if the next epoch commit fails, they roll back two versions as HCE+1 was never truly committed and they have moved forward to HCE+2. [GMS]* – Page 9, paragraph 5**

In the case where a 'commit' was changed into a 'flush' a rollback may revert back 2 or more epochs. The number of versions involved is related to the rate of commits the application was performing. This is a trade-off between commit frequency (which affects the entire ZFS pool) and the application's rollback granularity.

***So the application was not acknowledged so they have no guarantees as to the durability of that epoch. Failure of their current epoch could result in rollback to a version two epochs ago. [GMS]* – Page 9, Paragraph 6**

Correct. When a 'commit' mutates to a 'flush' the application has not been guaranteed durability for the epoch which it committed. From a semantic perspective this should be a tenable approach.

Regarding the Implementation of the Zero Copy Intent Log

***What is the long term view for these ZFS modifications? [GMS]* – Page 12, Paragraph 2**

The long term view is to approach this problem once the core VOSD implementation has been sorted out. While this is a critical optimization, it is still an optimization which doesn't affect the core implementation of the VOSD core. At this point the team is highly committed to implementing this feature into ZFS though our current focus has been on the VOSD / Lustre integration. The goal is to have the zero copy intent log patches pushed upstream into the ZFS tree.