| Date:<br>June 25th, 2013 | # High Level Design – Epoch Recovery<br>## 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.<br>Santa Clara, CA 95052 |

**LIMITED RIGHTS NOTICE.** THESE DATA ARE SUBMITTED WITH LIMITED RIGHTS UNDER PRIME CONTRACT NO. DE-AC52-07NA27344 BETWEEN LLNL AND THE GOVERNMENT AND SUBCONTRACT NO. B599860 BETWEEN LLNL AND INTEL FEDERAL LLC. THIS DATA MAY BE REPRODUCED AND USED BY THE GOVERNMENT WITH THE EXPRESS LIMITATION THAT IT WILL NOT, WITHOUT WRITTEN PERMISSION OF INTEL, BE USED FOR PURPOSES OF MANUFACTURE NOR DISCLOSED OUTSIDE THE GOVERNMENT.

THE INFORMATION CONTAINED HEREIN IS CONFIDENTIAL AND PROPRIETARY, AND IS CONSIDERED A "TRADE SECRET" UNDER 18 U.S.C. § 1905 (THE TRADE SECRETS ACT) AND EXEMPTION 4 TO FOIA. RELEASE OF THIS INFORMATION IS PROHIBITED.

## Table of Contents

## Revision History

| Date | Revision | Author |
|---|---|---|
| 03/26/2013 | v0.1: outline & first draft | Johann Lombardi |
| 05/01/2013 | v0.2: document capability revocation | Johann Lombardi |
| 05/23/2013 | v0.3: document new commit framework | Johann Lombardi |
| 05/25/2013 | v0.4: several clean-ups including split of the "container cleanup" section into sub-sections | Johann Lombardi |
| 05/27/2013 | v0.5: integrate feedback from Alex & Liang | Johann Lombardi |
| 05/28/2013 | v0.6: add flush section | Johann Lombardi |
| 05/29/2013 | v0.7: introduce HSE | Johann Lombardi |
| 05/29/2013 | v0.8: add "Changes from Solution Architecture" section | Liang Zhen |
| 05/29/2013 | v0.9: merge contribution from Alex on the first chapter | Alexey Zhuravlev |
| 05/29/2013 | v1.0: several cleanups | Johann Lombardi |
| 05/31/2013 | v2.0: add layout flush notion | Johann Lombardi |
| 06/03/2013 | v3.0: cleanups | Johann Lombardi |
| 06/05/2013 | v4.0: more minor cleanups | Johann Lombardi |
| 06/25/2013 | v4.1: integrate feedback from Ned Bass | Johann Lombardi |

## Introduction

While standard Lustre recovery aims at dealing with transient failures of Lustre servers (e.g. target failover or just server restart), applications still regularly have to dump their own in-core states on disk to be able to restart from a consistent and meaningful (from the application perspective) dataset after a non-transparent failure (e.g. client crash, Lustre recovery failed to restore the latest state of the filesystem). Such checkpoint/restart mechanisms often involve copying unmodified data and creating a significant number of files for each checkpoint. This could then result in expensive data movement across storage targets as well as a namespace pollution causing the metadata server to become a bottleneck.

Through the concept of epoch, the DAOS API allows applications to define atomic sets of changes and to specify the order in which those change sets should be applied. Persistent distributed states are then generated automatically by the backend filesystem which guarantees to the application that a consistent version of the container is always readable and that, in the worst case, only changes for uncommitted epochs have to be replayed.

The purpose of this document is to describe in details the various epoch recovery scenarios.

## Definitions

- OST: Object Storage Target, traditionally used by Lustre to store file data.
- MDT: MetaData Target where Lustre stores filesystem metadata (i.e. namespace, file layout, …)
- disk commit: local transaction commit on the backend filesystem Should not be confused with epoch commit.
- epoch commit: distributed commit at the DAOS level which results in a consistent state change on all the shards
- HCE: Highest Committed Epoch, which is the last successfully committed epoch. The HCE is the highest epoch accessible to readers.
- HSE: Highest Shard Epoch, which the highest epoch number committed on at least one shard. The HSE is readable if it is equal to the HCE, otherwise the HSE corresponds to a partially committed epoch which is then stuck on commit and is not published to readers.

## Changes from Solution Architecture

As mentioned in the "Lustre Restructuring and Procotol Changes" design document, several changes were made to the DAOS API and solution architecture. Among all those modifications, the ones with an impact on epoch recovery are the following:

- Introduction of the Highest Shard Epoch, namely HSE. This is the highest committed epoch of all shards in a container. When a container is in consistent state (i.e. all shards successfully committed the last epoch), the

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

HSE is equal to the HCE. Otherwise, the last commit attempt was only partially successful (some shards failed to commit while some others successfully created a snapshot) and, in this case, the HSE value is higher than the HCE.

- The epoch scope notion has been removed and merged into the container handle abstraction.
- Container open can now notify the caller if the container is inconsistent (through a "status" parameter), requiring application-driven recovery.
- A partially committed epoch cannot be aborted or rolled back. Deletion of snapshot can fail and make the undo procedure quite complex. Therefore one cannot abort a partially committed epoch and should instead either attempt to re-commit in the case of a transient shard failure, or disable failed shards permanently in a higher epoch and commit.
- Flushed data in HSE+ will always be discarded if the writer closes the container. It is indeed impossible to know whether flushed data is consistent or not because it has not been committed yet. As a result, un-committed data has to be discarded in this case.

# Specification

## i. Interaction with Standard Lustre Recovery

As explained in section iii of the "Lustre Restructuring and Protocol Changes" design document, the Lustre implementation of the DAOS API still relies on the standard Lustre recovery to handle transient network or server failures.

### 1. DAOS Request Resend

Lustre deals with network failures by resending RPCs multiple times. Each request is assigned a timeout and is resent if the client did not get a reply when the timeout expires. Timeouts are adaptive and can be extended by the server by issuing an early reply to the client. If the reply turns out to be lost, then the Lustre server might receive the same request twice. In Lustre current implementation, all OST requests can be safely re-executed (i.e. all OST requests are idempotent) multiple times whereas most MDT requests have to be processed only once (aka "execute once" semantic). To address this, the MDT records in the last_rcvd file the processing result of the last request sent by each client and reconstructs the reply instead of re-executing the request if this latter is received a second time. The drawback is that each Lustre client is limited to one RPC in flight to the MDT since there is only one slot per client in the last_rcvd file to store request information. There is no such limitation on OSTs thanks to the idempotence of request processing.

As far as DAOS is concerned, all container operations (except container query/getattr) processed by the MDS are not idempotent and will thus have to deal with the last_rcvd file limitation. This means that, like regular Lustre metadata requests, there can be only one DAOS container creation or unlink in flight. There is a plan to address this limitation in the Lustre mainline by adding support for multiple slots for each client in the last_rcvd file.

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

That said, all shard, object and epoch operations are idempotent and can then be safely re-executed multiple times.

## 2. MDT Recovery

The MDT maintains in-core states composed of the following elements:

- File handles (including unique cookie, read/write, epoch referenced) and capabilities
- Epoch state: current HCE & HSE, epoch to commit and lowest referenced epoch (can be calculated from the file handles).
- Layout which stores the whole history (based on epoch numbers) of layout modifications

The purpose of this section is to describe how the MDT uses client replay data to rebuild those states after a restart or failover. The MDT-shard recovery (aka container recovery) on the first open is detailed in chapter iii of this document.

### *Container Handle Replay*

On MDT restart, clients have to replay container handle as done today with open file handles. MDT can thus rebuild the list of open container handles out of client information. MDS uses RPC XID and local in-memory structures to recognize the case of resent and reconstruct the reply.

Capabilities are regenerated upon open replay and the client will have to refresh it. That said, no capability checks are done on the MDT during request replay.

In order not to trust clients blindly during recovery, MDT should sign the structure containing the handle and capability list sent to the client with a private key so that it can check during recovery that this structure was produced by itself and has not been modified by a malicious client. This is considered out of the scope of the project.

### *Epoch State Recovery*

Upon restart, the MDT also has to reconstruct the epoch tracking list (what is the lowest version currently read by clients). During recovery and upon recovery completion, the MDT does not remove unreferenced epochs from the shards, instead it's done in lazy manner when the client moves the cursor ahead with slip/wait request – then MDT will find the lowest epoch referenced by the file handles and will command OSTs to remove epochs (snapshots) up to that one.

As for commit request, clients are not supposed to receive an acknowledgement of commit until all the shards report their commit back. Thus, upon MDS reboot, the client will be resending commit request to recover "epoch to commit" state and restart commit process. MDS should not initiate commit on its own unless explicitly requested by the client, so that the client can handle partial commit.

### *Container Layout Reconstruction*

Every new shard added to the layout is tagged with the epoch it was added in. Given this, to commit the epoch, MDT should regenerate a list of active shards for the epoch. The client's responsibility is to replay changes to the layout using regular Lustre recovery. To

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

do so, all RPCs altering layouts are assigned a transno associated with the on-disk update of the layout on the MDT. If the MDT crashes, then clients will replay layout modifications and allow the MDT to rebuild the layout as it was before the crash. VBR (Version Based Recovery) should be used to improve recoverability in the light of possible missing clients.

***Client-MDT vs Container Recovery***

As replay process is going in transno order and commit process requires valid layout, MDT should not start container recovery (i.e. MDT-shards recovery) until client-MDT recovery is completed. This means that replay of open should only create an in-memory structure representing file handle (mfd structure) and schedule container recovery to be done up on recovery completion. The only state that can't be retrieved from clients is HCE. So scheduled container recovery just needs to query the shards and cache current HCE.

Resend and replay for container unlink is done by regular Lustre means.

## 3. OST Recovery

The shard in-core state consists of the handle and capability list. On restart, OSTs have to fetch the capability list from the MDT.

As for the shard persistent state, it is represented by:

- default dataset with intent logs
- snapshots (should be listed in an index in the default dataset)
- highest locally committed epoch

The persistent state is supposed to be idempotent so that any request can be executed arbitrary times. As a result, all DAOS object and shard operations are replayed using Lustre standard mechanism. Besides, similarly to MDT recovery, no capability checks are done on the OST during replay.

## 4. Eviction & Unexpected Application Termination

In general, a container handle is closed by the application that opened it. If the application quits unexpectedly, then it is the responsibility of the DAOS client to close the container on behalf of the application. Similarly, if the client node is globally evicted, then it is up to the MDT to clean up the export associated with the client and to revoke the container handle.

Closing a container handle results in the revocation of the capabilities associated with this handle on the shards though a server collective initiated by the MDT. This capability revocation is vital to protect the storage from any in-flight I/Os that might still be submitted with a container handle that has been closed already, e.g.:

- in-flight I/Os that might still be on its way or delayed indefinitely in a router for example.
- a slave process which is part of a collective open and is not aware yet that the master process has aborted the container and might continue sending I/O operations to shards.

Lustre servers will fail the capability check for such requests and return EPERM to the client.

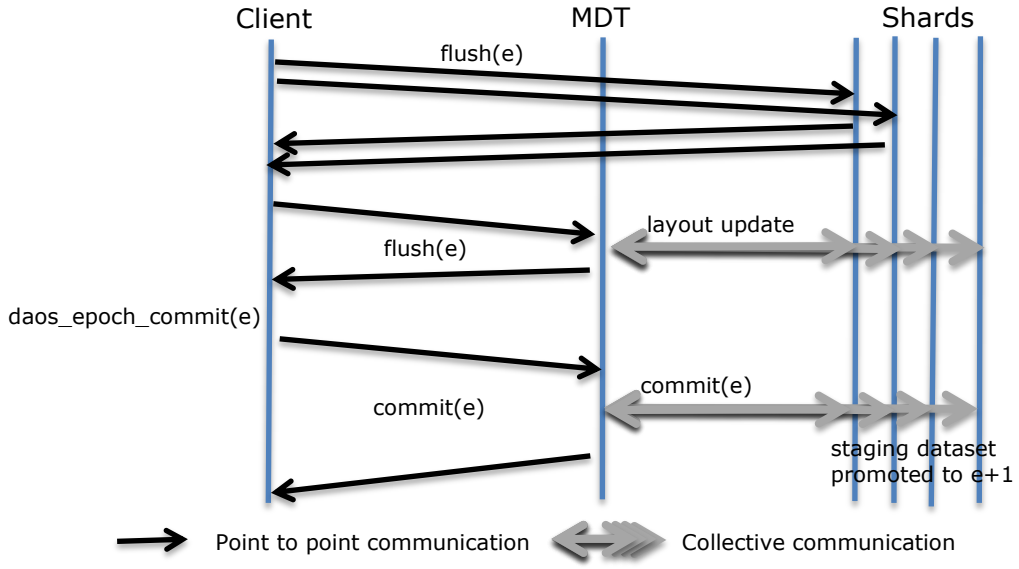
**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

## ii. Commit and Failure Handling

### 1. Commit Framework

To commit an epoch e, the application has to submit and flush all the I/O operations for all epochs smaller or equal to e. The commit request is processed by the MDT which acts as a proxy to the container shards and uses server collectives to trigger a local epoch commit on all the shards. This step involves flattening the intent logs into the staging dataset, deleting the intent logs and taking a snapshot. Once completed, the staging dataset can be promoted to the next epoch (i.e. e + 1) and flattening for this epoch can even eagerly start (as detailed in the next chapter, rollback of the staging dataset will still be possible if the application closes and reopen the container without committing e + 1).

The diagram below represents how a MDT handles a commit request.




Epoch Commit

If all local commits are successful on all the shards, the epoch is considered as globally committed and becomes the new HCE.

### 2. Failure and Partial Commit

There are several types of problems that can prevent a shard from successfully completing a local commit:

- The OST hosting the shard is unresponsive. The failure might be temporary (e.g. OST failover/restart, network issue, …) or persistent (e.g. the OST is dead and can't be recovered). In this case, the commit request does not reach the shard and fails after a timeout.
- The shard received the commit order, but failed to complete the procedure due to errors (e.g. EIO, ENOMEM) on the backend storage. In this case, the shard returns an error to the MDS and leaves the backend storage in a partially committed state:

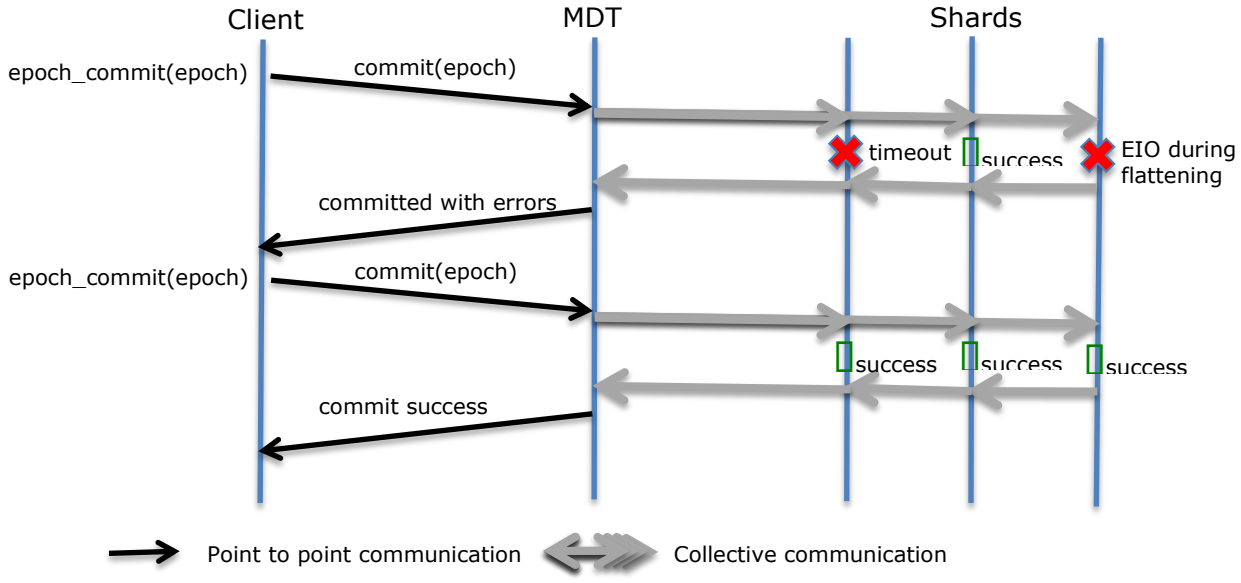
**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

- If the error happened during flattening, then the intent logs won't be deleted and re-committing would re-execute the flattening operation from the beginning (which is safe since all object operations are idempotent).
- If the error happened while taking the snapshot, then flattening was successful and the intent logs were deleted, a re-commit would just try taking the snapshot again.
- In both cases, the staging dataset is not promoted to the next epoch, so that a re-commit attempt on the failed shard is possible.

When some of the active shards failed (because of a timeout or any other errors) to commit an epoch, the epoch is considered as partially committed. In this case, the epoch is stuck on commit and an error with the list of shards that failed to commit is returned to the client. This latter can then decide to either:

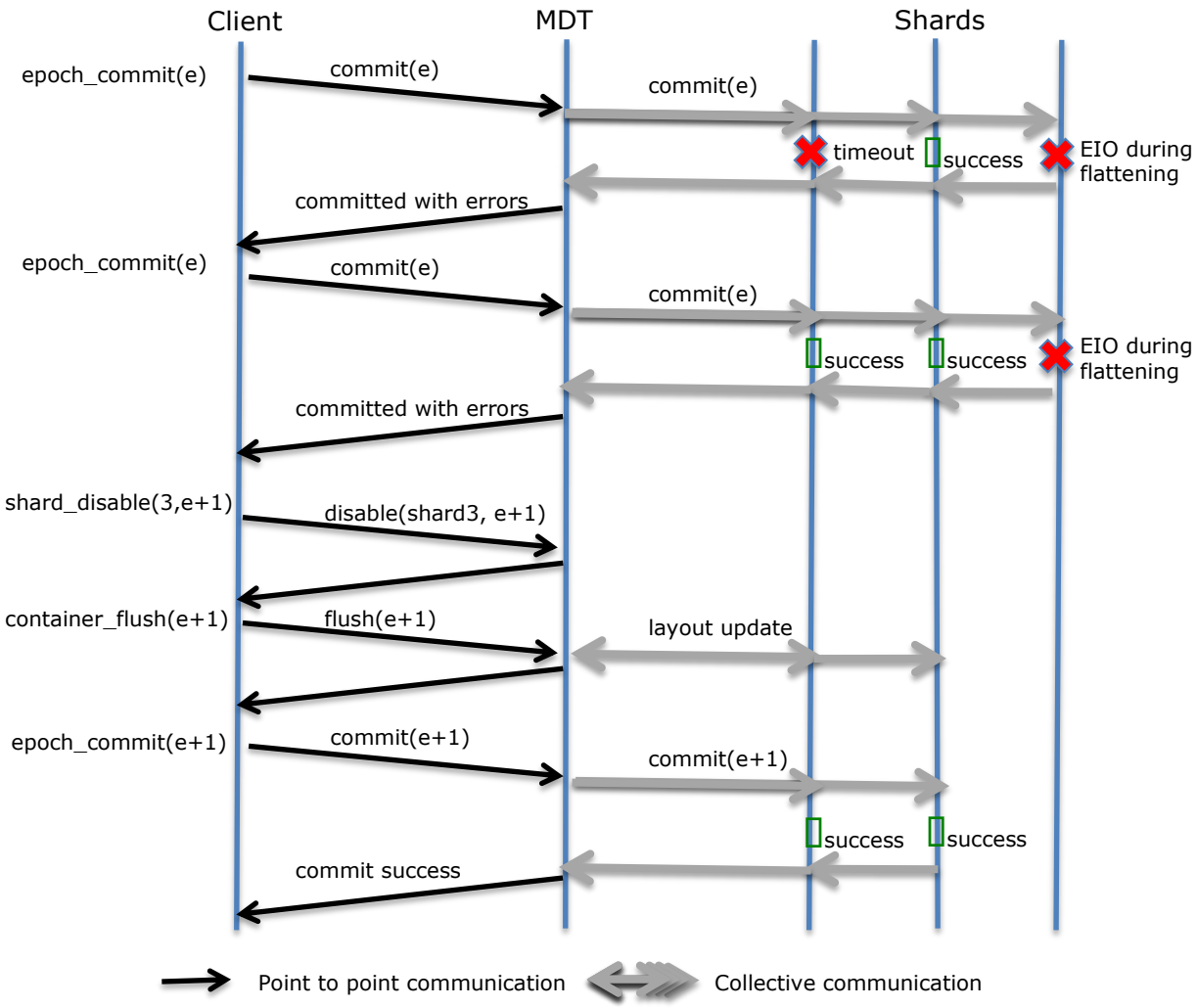
- retry to commit. This will result in the MDS triggering a recommit through a server collective on all the active shards. Shards that have already successfully committed this epoch locally will just return success, whereas the others one will attempt to commit. Recommitting might be successful if the problem was just transient.
- Or the client can decide to disable the shards that failed using an epoch number higher than the one partially committed and then try to commit this epoch. This way, epoch numbers always roll forward and there is no need to rollback a locally committed epoch on a shard and to handle failure of the rollback process.

Those two options are schematized in the figures below:




Successful Re-Commit of a Partial Commit

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**




Disabling Shard after a Re-Commit Attempt

If the partial commit is not resolved by the application which closes the container intentionally or not (e.g. the application terminates unexpectedly or the client gets evicted), then the container is left in this state and it will be up to the next opener to recover the container by either attempting to recommit or by disabling the failing shard(s). One caveat though is that, on a new container open, data in intent logs for epochs that have been neither committed nor partially committed are discarded. Container query has been modified to report not only the HCE, but also the highest shard epoch, namely HSE. The next chapter provides more details on container recovery on open.

Readers can only access globally committed epochs and are thus not notified when an epoch is partially committed. A successful recommit will of course bump the HCE and notify readers that a new globally committed epoch is available.

## 3. Shard Addition and Commit

As explained in the "Lustre Restructuring and Protocol Changes" design document, the MDT is only caching the container layout and the definitive layout is actually the one stored in the shards. On shard addition, the MDT allocates a shard to a container and

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

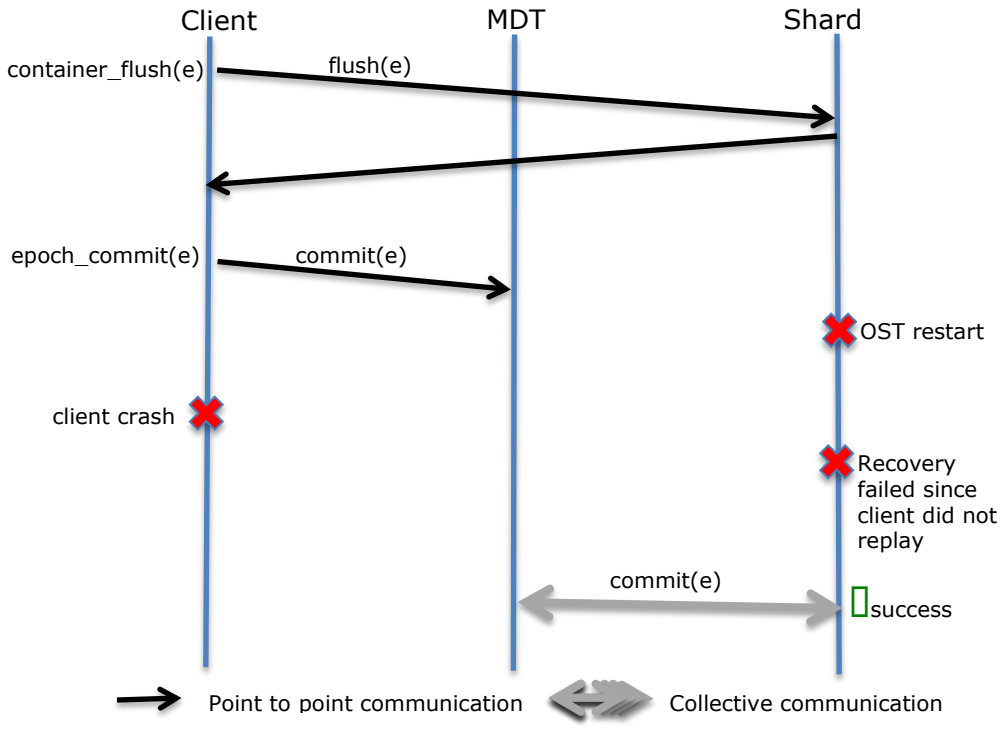
update the layout locally. Layout changes are then propagated to the shards once the epoch in which the shard was added is flushed. This layout update involves a synchronous flush to make sure that all changes are safely committed to disk before starting the commit process. The MDT might also take this opportunity to flush the layout changes to its local disk.

If the MDT crashes before the flush operation is executed, then we rely on the client to replay the shard addition request.

If the client then cannot replay the shard addition, the client is evicted and the layout flush operation (if not sooner) will return an error which will notify the application that unflushed data have been lost. As for the orphaned shard(s), it will be cleaned up by the regular MDT orphan recovery mechanism already in place today for object pre-allocation.

## 4. Flush & Epoch Recovery

On successful flush, the application is guaranteed that all I/O operations previously executed from this client for all epochs lower or equal to the flushed epoch have been successfully written into the intent logs and will not have to be replayed by the standard Lustre recovery in case of OST failure. This requirement is vital to make sure that the MDT does not commit an incomplete epoch like in the scenario described by the figure below.




Shard Restart during Commit

In the diagram above, if the flush operation does not guarantee that all I/O operations have safely been written to disk (in the intent log or directly in the staging dataset), recovery won't be able to restore the state of the shard due to the missing client. As a

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

consequence, the MDT might proceed with a commit on a shard which is missing data, breaking the DAOS transactional semantic.

Therefore a flush operation not only triggers writeback on the Lustre client, but also results in a flush RPC sent to the shard(s). The client could pack in the flush request the highest transno assigned to its RPCs which can be used by the server to determine whether a sync is really required. The server can indeed compare the transno provided by the client in the flush request with the last committed transno and just report success if all updates have already reached the backend storage.

## iii. Container Open & Recovery

A container is considered "clean" when:

- no I/O operations have been submitted for uncommitted epochs
- the last epoch commit was successful on all the shards active in the layout. In other words, the HCE is equal to the HSE.

An application might leave the container in an unclean state at close time in several cases:

- the application decides of its own to call daos_container_close() on an unclean container after encountering an internal error and wants to abort processing.
- the application might terminate unexpectedly after a fault or just be killed by the job scheduler
- the client node where the master process of a collective open got evicted (e.g. after a reboot or a crash). In this case, the MDT cleans up the export associated with the client and closes the container on its behalf.

In all those cases, the container is left in bad shape and needs to be recovered.

The first opportunity to recover a container with partially committed states is on the next open. The opener sends the request to the MDT, which, as a proxy to the shards, is responsible for rebuilding the state of the container based on the shard information. This process is composed of three steps that are detailed in the next sections:

1. Layout and HSE detection
2. Abortion of I/O operations from uncommitted epoch
3. HCE resolution

Once successfully opened, it is then up to the application to recover the container. This final step is addressed in the last section entitled "Application Recovery".

### 1. Layout and HSE Detection

In DAOS, container data stored on the MDT is not authoritative and is actually just a cache of container attributes that are actually distributed across all the shards. As such, the container layout on the MDT disk is not necessarily up-to-date and might need to be refreshed by consulting the layout stored on the shards.

The MDT proceeds as follows to discover any layout changes that might have happened since it last wrote a copy of the container layout to disk:

1. The MDT reads the cached layout from disk

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

2. The MDT sets up a server collective over all the shards that are still marked as active in the current layout to read the latest committed epoch on each shard. The maximum of all the values reported by the shards is the HSE.
3. The MDT sets up another server collective to read the container layout from all the shards which have reported a latest committed epoch equal to the HSE.
4. If the layout is not the same on all the shards, the container is then corrupted and needs manual repair. As a consequence, the open request fails and the returned status to daos_container_open() is CORRUPT. A container repair tool might be developed in the future to repair such containers (see last chapter).
5. Finally, the MDT compares its current layout with the one it got from the collective.
    a. If the layout is different, then we restart the same procedure from step 2 with this new layout since we still have to discover shards.
    b. Otherwise, the MDT should now have discovered all the shards and have the definitive layout. By now, the MDT should also know what is the latest committed epoch on each active (i.e. ones that are not marked as disabled in the layout) shard.

One caveat is if all the shards listed in the cached layout stored on the MDT have all been disabled. This extreme case could be addressed by doing a collective over all the OSTs of the Lustre filesystem to discover shards associated with the container FID. Another option would be to force a sync of the container layout to the MDT disk on client's flush. This corner case might be addressed in the prototype as time permits.

### 2. Uncommitted Epoch Abortion

Once aware of the latest layout and the HSE, the MDT can clean up I/O operations of uncommitted epochs. Those I/Os (if any) were submitted by the previous opener and haven't been committed and maybe not even fully flushed. Since we can't tell to the new opener what I/O operations have been processed and which ones haven't, the MDT notifies all shards to abort all those operations.

In practice, all epochs strictly higher than the HSE have never been committed, even partially. Since at least one shard has successfully committed HSE, it means that all the shards have received and flushed all I/O operations for all epochs up to HSE (that's part of the contract signed with the application: commit should be issued only once all I/O operations have been flushed).

The MDT thus set up a server collective with all the active shards requesting data for epochs higher than HSE to be discarded. This process includes deleting all intent logs associated with those epochs and maybe even resetting the staging dataset.

In addition, the MDT parses its on-disk layout and destroys all shards that have been added in an epoch strictly higher than the HSE.

Besides, it is worth noting that the DAOS API does not provide a function to discard changes from uncommitted epochs. The way to do this is to close the container with uncommitted data and to reopen it.

### 3. HCE Resolution

The purpose of this last step is to determine what the highest consistent epoch to offer to readers. Consistent here means that all shards that are active in the layout associated

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

with this epoch have successfully managed to commit this epoch. In other words, the MDT is looking for the highest non-partially-committed epoch, namely HCE.

If all the active shards have a snapshot for HSE, then the container is considered as cleaned and the container HCE is simply equal to HSE. The open request is successful.

Otherwise, the HSE is reported as the highest partially committed epoch for the container through daos_epoch_query() and the MDT now needs to find out what is the real HCE that can be presented to readers. Let's call HSEi the the highest locally committed epoch on shard i and n the number of shards. HCE is then smaller than or equal to min({HSEi}0≤i≤n.

As a result, the MDT uses the following algorithm to discover HCE:

1. Parse the shard information collected so far by the MDT and find the lowest committed epoch. Let's call this epoch e.
2. Parse the layout and build the list of shards that were active in epoch e.
3. Set up a collective across all those shards and ask them if they have a snapshot for epoch e. If so, they should just return e. Otherwise, they should return the nearest epoch smaller than e for which a snapshot exists on the shard. If none, then 0 should be returned.
4. If all shards return e, then the current container HCE is e. If not, then we should repeat from step 1 with the new epoch numbers that have been collected.

The recovery flag (actually the STUCK status) is returned through daos_container_open() to notify that the container has a partially committed epoch that needs to be fixed by the application itself.

### 4. Application Recovery

When the STUCK status is returned on open, the application has to decide whether to try to recommit the same epoch or to disable failed shards (the list of shards that failed the last commit is now accessible through the DAOS API) in order to restore the container in a clean state.

Attempting a recommit at this point is safe given that at least shard successfully committed, so this means that the MDT asked at some point for this epoch to be committed and this can only happen once all I/O operations (including layout update from the MDT) have been flushed.

If recommit still fails, then the only option is to disable the failing shards. To do so, the DAOS API was extended (see daos_container_query_shard() ) to allow applications to access the list of shards that failed to produce a snapshot during the last commit.

## iv. Orphaned Shard and Broken Container Cleanup

### 1. Shard Scrubbing (optional)

An orphaned shard could be detected by checking the actual container layout, which should have no reference to this shard. As a result, a scrubbing process could parse the shards hosted by a given OST, do reverse lookups to the MDT (similar to online lfsck) and destroy orphaned shards, if any.

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

## 2. Idle Container Cleanup (optional)

In addition to recovering the container on open, it might be desirable to pro-actively repair unclean containers instead of waiting for the next job to resolve the problem. Such a feature would be a requirement for a DAOS-HA library implementation which might definitely want to rebuild the RAID sooner rather than later. That said, fixing unclean containers involves some basic understanding of the data and metadata distribution and replication scheme used by the application (e.g. what DAOS-HA would typically handle internally). It is therefore almost impossible to come up with a generic repair tool that could handle the application recovery. As a result, application-specific plugins would have to be integrated into the scrubbing engine in order to pro-actively fix partially committed container.

## 3. Container Repair Tool (optional)

### *Corrupted Container Recovery*

A container is considered corrupted when at least two shards have successfully committed the same epoch but report a different layout.

This might happen, for instance, if the flush-before-commit rule was not honoured for whatever reasons. Although there is no plan currently to enforce this rule at the DAOS level, one could consider recording on the OST for each epoch the latest transno where something was modified in this epoch. Then at commit time, one could check that this transno is smaller than the last committed transno. If not, it means that a client is trying to commit while some I/O operations using this epoch did not hit the disk yet. Such a protection won't work if the OST restarts and fails Lustre recovery. Another option could be to add in the intent log a "flush" record. Shards could then check that the IL has a final "flush" record before committing.

A repair tool could perform a careful study of the available snapshots on all the shards and rollback the container to the last consistent state. This process might involve destroying snapshots associated with broken epochs.

Besides, there is also a “legitimate” case where a container could end up with two disjoint layouts:

- A container has 4 shards, namely A, B, C and D
- Some OSTs are down causing shards C & D to be unavailable
- An application opens the container, disables shard C & D, commits and closes the containers.
- Shard C & D are back to life and now shards A & B become unavailable.
- An application opens the container, finds C & D as active, disables A & B and commits.

In the scenario above, it is assumed that the MDT was not able to update the layout on disk.

A solution could be to rely on the layout generation to define which set of shards should be kept and which one should be destroyed. To do so, the MDT would have to bump the layout generation number to an always-increasing value (e.g. based on wall clock time) and choose the shards having a layout with the highest generation. The MDT could

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**

actually handle this by itself in the step 4 of the open procedure (see chapter ii, section 1) without requiring an external tool to repair the container.

Another approach to address the problem above is to require a quorum of shards to allow the container to be opened.

***MDT Rebuild***

Given that the actual state of a container is distributed across all its shards, it might be possible to rebuild the MDT namespace by scanning shards on all the OSTs. This would require propagating some container attributes which are only stored on the MDT for now to the shards, that's to say:

- the container UID/GID (needed anyway on shards for quota accounting)
- the name(s) associated with the container in the namespace

The former can easily be done on shard addition since we anyway have to propagate the layout along with the container FID to the newly added shard. The latter would require updating the shards on rename and hard link creation.

## Risks & Unknowns

- Too many sync operations (clients' flushes, layout update through MDT and snapshot creation) might be required to commit an epoch, which could impact performance. We are considering some optimizations to the model to reduce the number of syncs.
- The name "epoch" might be replaced with transaction and version to be consistent across the Fast Forward stack.

**The information on this page is subject to the use and disclosure restrictions provided on the cover page to this document. Copyright 2012, Intel Corporation.**