Date: June 4, 2014	# DAOS Epoch Recovery Design ## 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
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
06/4/2014	v4.2: update after prototype implementation	Johann Lombardi Li Wei Liang Zhen

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.

Specification

i. Transient Server & Network Failure

1. Network Failure & 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 idempotent 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.

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

For operations involving server collectives, DAOS clients will retry for a certain amount of time if corresponding server collectives fail because of network issues before reporting errors back to applications.

2. Transient MDT Failure

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

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

Standard Lustre replay mechanisms are used to reconstruct the above states in case of MDT failure. 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 recovered upon open replay with client's copies.

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/close 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.

Container Layout Reconstruction

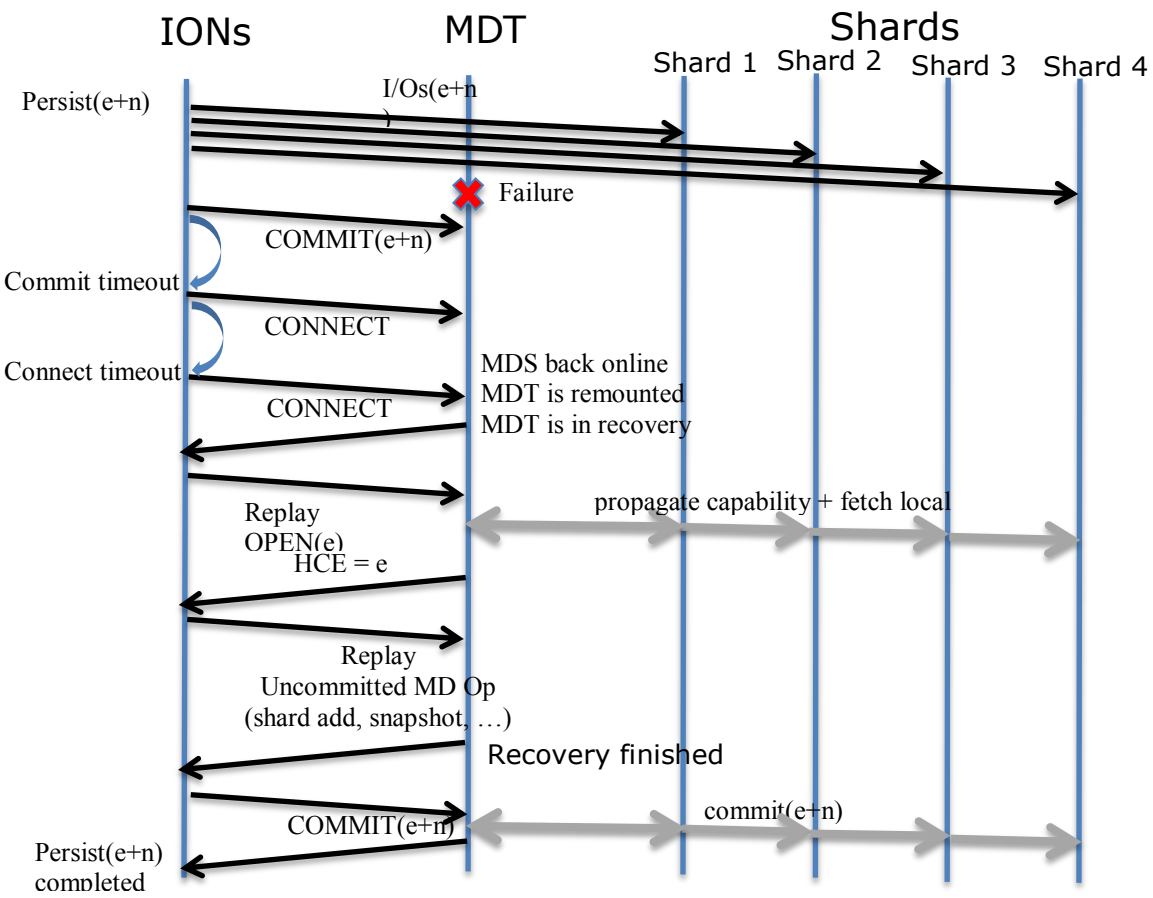
Every new shard added to the layout is tagged with the epoch it was added in. Given this, to commit the epoch, MDT should regenerate a list of active shards for the epoch. The client's responsibility is to replay changes to the layout using regular Lustre recovery. To do so, all RPCs altering layouts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

Container create/unlink

Resend and replay for container unlink is done by regular Lustre means. Shards are destroyed via LOD/OSP which issued SHARD_DESTROY RPCs to OST once the container has been unlinked from the namespace on the MDT.

The figure below schematizes how MDT recovery was demonstrated in milestone 7 of the project.



In this example, the MDT has been restarted while IOD was trying to commit a new epoch. Upon restart, the MDT enters in recovery and asks previously connected client to replay container open handle(s) as well as uncommitted metadata operations. Once recovery is finished, IOD can re-execute the commit request and complete the persist operation.

Note on Eviction & Capability Revocation

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 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 closed the container and might continue sending I/O operations to shards.

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

3. Transient OST Failure

The shard in-core state consists of the handle and capability list. On restart, OSTs have to fetch the capability list from the MDT. This happens after regular Lustre OST recovery is completed, so that it is not racing with any shard creation replays. Non-recovery requests are accepted and processed in parallel, unless capability checking is necessary, in which case the request in question get EINPROGRESS and will be retried by the client until the capability list fetching has completed.

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
- shard "object" in root dataset, including an "filter_fid" EA storing its container FID

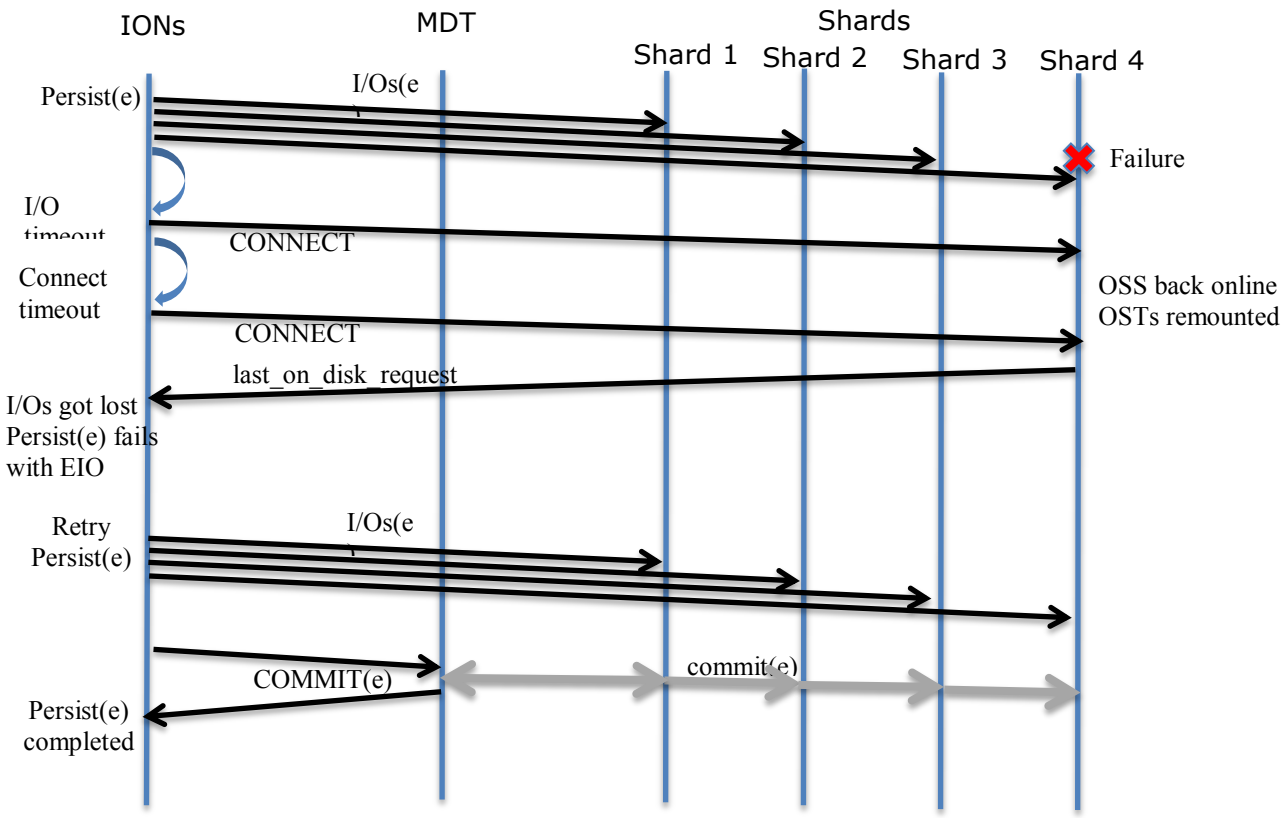
The persistent state is supposed to be idempotent so that any request can be executed arbitrary times. As a result, all DAOS object and shard operations are replayed using Lustre standard mechanism, except for object write and punch.

Unlike regular Lustre object write and truncate, no on-disk transno is assigned for replay to DAOS object write and punch which are thus not automatically replayed during Lustre recovery. When DAOS object I/O requests are lost after an OST request, it is indeed up to the application (IOD in this case) to resubmit all missing I/O operations. The application is notified of failure when calling flush: any error during flush means that all operations since the last successful flush have to be resubmitted. In the prototype, any error during persist() is returned by IOD to the upper layer (i.e. HDF) which retries the whole persist operation.

The following mechanism is used by client to find out whether any I/O requests got lost which should cause flush to fail:

- Each time a client gets a RPC reply for a DAOS write/punch operation, it records the highest transno packed in the reply for each shard
- Upon restart, OSTs report back to clients the last on-disk transno
- Clients then scan the per-OSC shard list and mark any shard with a submitted_transno higher than the OST on-disk transno as failed
- Next time an I/O operation (write, punch or flush) is submitted for a shard marked as failed, EIO is returned
- A flush call clears the failed state of a shard on the client.

The figure below summarizes how transient OST failure has been demonstrated in milestone 7 of the project.

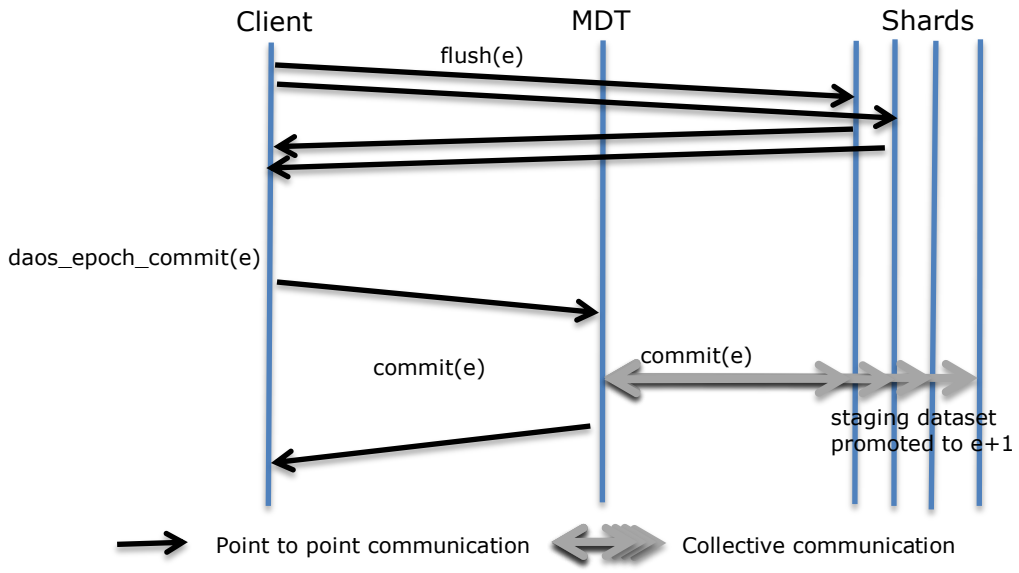


ii. Commit and Failure Handling

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

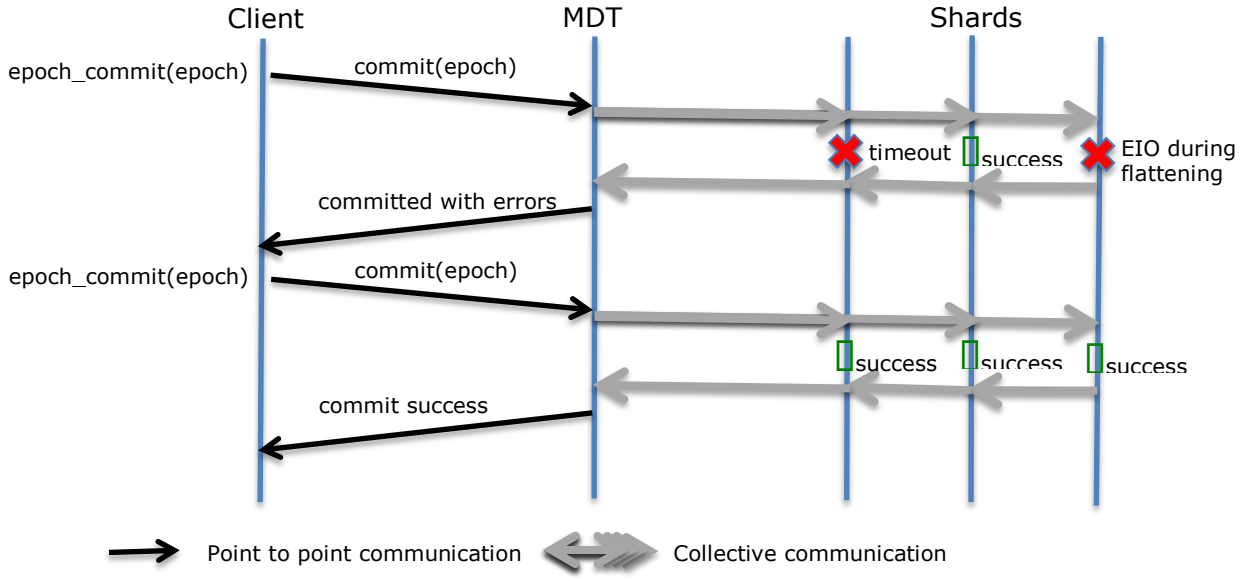
There are several types of problems that can prevent a shard from successfully completing a local commit:

- 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 via gossip (or after a timeout if gossip isn't used).
- 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:
    - 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.

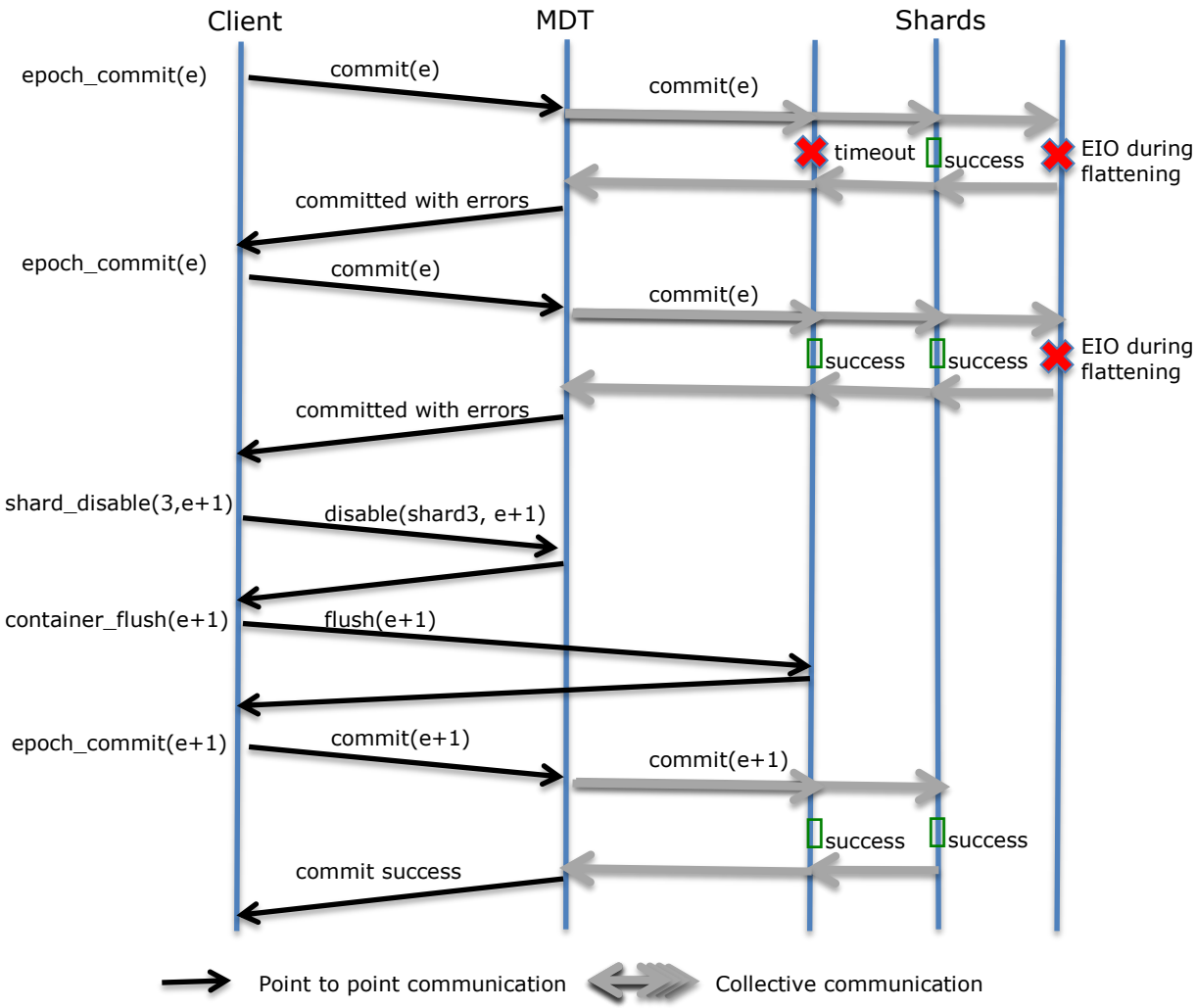
When some of the active shards failed (because of a timeout or any other errors) to commit an epoch, the epoch is considered as partially committed. In this case, the epoch is stuck on commit and an error with the list of shards that failed to commit is returned to the client. This latter can then decide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

- retry to commit. This will result in the client asking the MDS to do 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



Disabling Shard after a Re-Commit Attempt

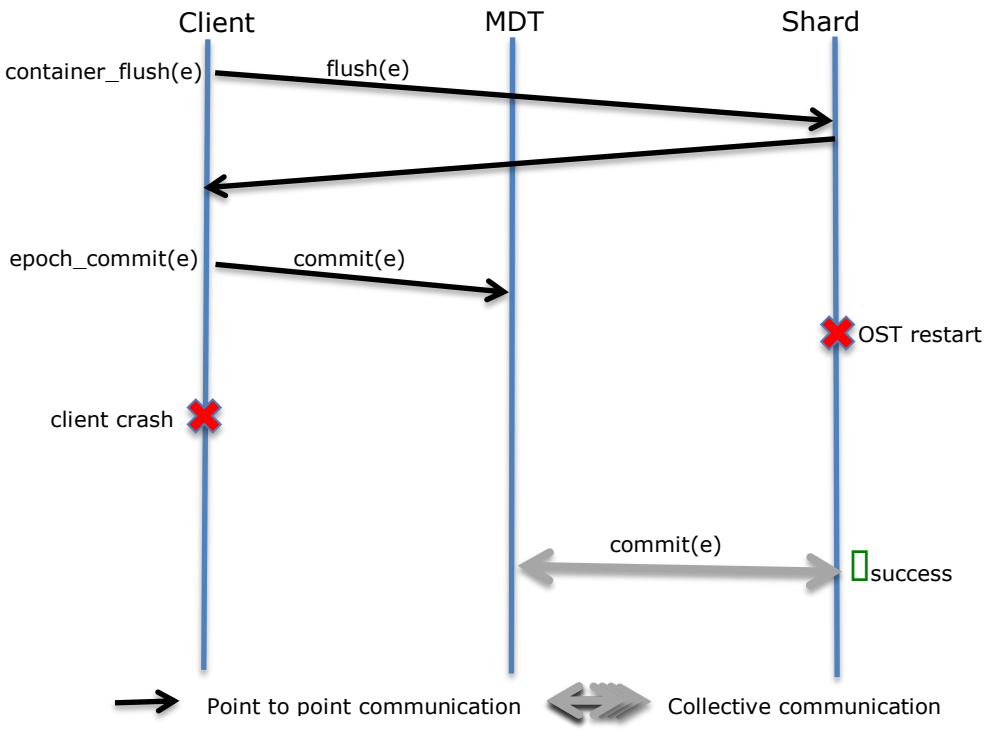
If the partial commit is not resolved by the application which closes the container intentionally or not (e.g. the application terminates unexpectedly or the client gets evicted), then the container is left in this state and it will be up to the next opener to recover the container by either attempting 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. 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 from the burst

buffer. 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), missing I/O operations won't be re-executed from the burst buffer due to the missing IONs. As a 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.

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. Epoch Status Detection

The MDT proceeds as follows to learn about the state of affairs. The MDT reads the layout from disk and sets up a server collective over all the shards that are still marked as active in the current layout. The purpose of this collective is to find out how many different committed epoch numbers are present across all the shards. Reply aggregation is used to return the following information:

a. The lowest committed epoch

b. The highest committed epoch

c. An intermediate committed epoch between the lowest and highest ones, if any.

d. The number of shards with no valid HCE snapshot (i.e. epoch is still 0)

The MDT then processes each case as follows:

- If the collective failed to reach some of the shards, the container status is set to "incomplete" and returned to the application.
- If all shards are reachable, but some returned an error (e.g. cannot open shard because of I/O error), then the "faulty" status is reported to the application.
- If all shards returned the same epoch (lowest = highest = intermediate = HCE), the container status is set to "OK" and the MDT initiates another server collective across all the shards to abort I/O operations logged for epochs >= HCE.
- If there are exactly two different epochs (the lowest epoch is different form highest and intermediate equals either highest or lowest), the container is considered as "stuck". The MDT attempts to recommit the highest epoch number. Upon success, the HCE is set to the highest committed epoch number, the container status is "OK" and another collective is executed to abort epochs >=HCE. If recommit failed, the "stuck" status is returned to the application.
- If there are more than two different epochs returned from the collective (i.e lowest, intermediate and highest epochs are all different numbers), the container is considered as corrupted and the "corrupted" state is directly returned to the application.

2. Application Recovery

As detailed in the previous section, container open will be successful in most of the cases and the application has to interpret the container status flag returned by the MDT:

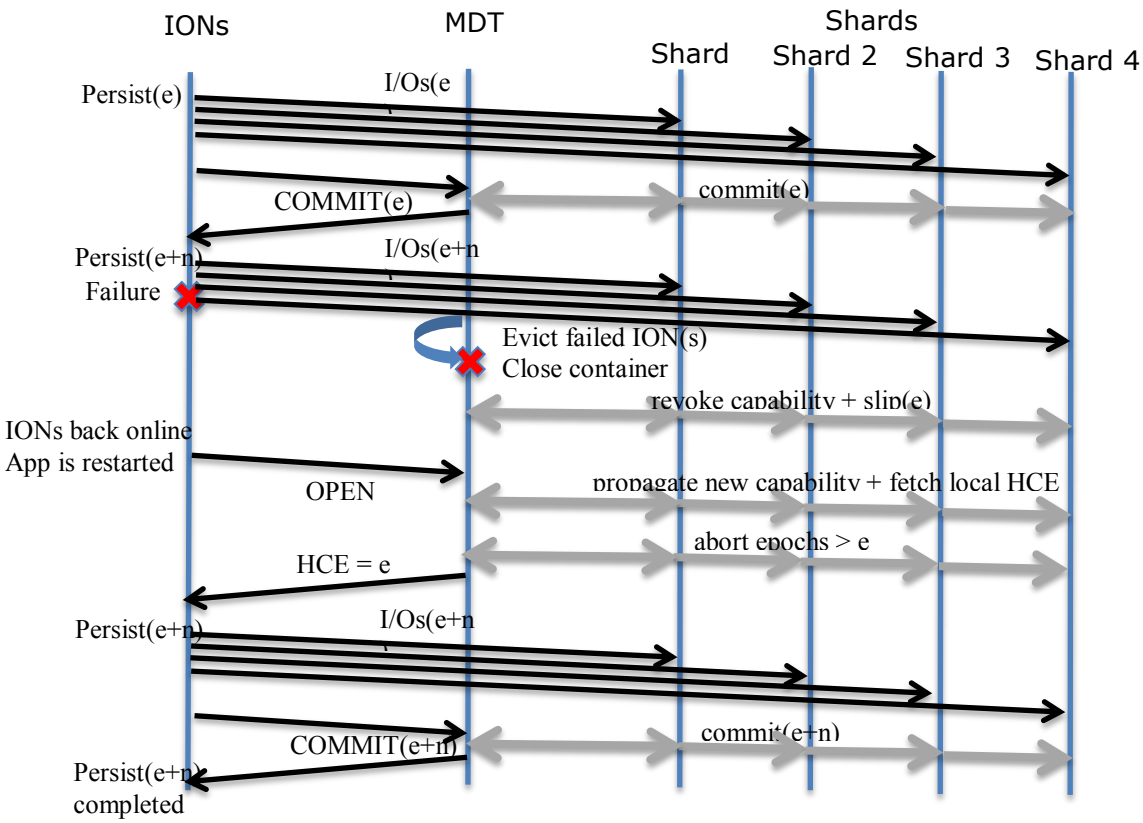
1. When the "stuck" status is returned on open, the application has to decide whether to try to recommit (although the MDT has retried already) the same epoch or to disable failed shards (the list of shards that failed the last commit is now accessible through the DAOS API) in order to restore the container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

   Attempting a recommit at this point is safe given that at least one shard had 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 the 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.
2. If the container status is reported as "incomplete", some shards were not reachable and the application can either disable the inaccessible shards or close & reopen the container, hoping for the missing shards to be back.
3. If the container status is "faulty", the application will have to either disable the failed shards or repair them.
4. If the container is "corrupted", then a careful study of the per-shard HCE is required to fix the container (see next chapter for more information).

When DAOS-HA is eventually supported, disable shards and adding new ones to rebuild redundancy will be very common.

The diagram below schematises how epoch recovery (i.e. ION restart in this case) was demonstrated in quarter 7 of the project.



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.

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 restore the redundancy 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 might be corrupted if, for instance, 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 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.