Date: **May 26th, 2014**	# DAOS Lustre Restructuring and Protocol Changes 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
03/20/2013	v0.1: document outline	Johann Lombardi
03/25/2013	v0.2: first draft	Johann Lombardi
04/10/2013	v0.3: add event and event queue section	Liang Zhen
04/12/2013	v0.4: complete client-side section	Johann Lombardi
04/13/2013	v0.5: add storage topology section	Johann Lombardi
04/23/2013	v0.6: integrate early feedback from Andreas and from LUG discussions	Johann Lombardi & Liang Zhen
04/24/2013	v0.7: add missing sections	Johann Lombardi
04/26/2013	v0.8: add snapshot & clone	Johann Lombardi
05/07/2013	v0.9: integrate feedback from Paul and Liang	Johann Lombardi
05/10/2013	v1.0: integrate initial feedback from Alex	Johann Lombardi
05/14/2013	v2.0: add more details on capability, collective open and layout management and take Eric's initial feedback into account.	Johann Lombardi
05/16/2013	v3.0: integrate more feedback from Eric and Liang.	Johann Lombardi
05/22/2013	v3.1: integrate DAOS API changes, add API as appendix	Liang Zhen
05/23/2013	v3.2: minor cleanups	Johann Lombardi
05/28/2013	v3.3: integrate DAOS API changes	Liang Zhen
05/29/2013	v3.4: more API changes	Liang Zhen
05/29/2013	v3.5: add more details on shard pre-allocation	Johann Lombardi
05/31/2013	v3.6: add notion of layout update flush	Johann Lombardi
06/25/2013	v3.7: integrate feedback from Ned Bass	Johann Lombardi
05/29/2014	v3.8: update document after prototype implementation	Johann Lombardi

Introduction

The DAOS API introduces novel abstractions involving changes all over the Lustre stack. This document aims at describing in more details the Lustre modifications that were briefly enumerated in the solution architecture. This includes not only the Lustre client extensions required to support the non-blocking object-based DAOS API, but also the server-side functionality used to implement containers, shards and distributed transactions over multiple storage targets.

Please note that server collectives are addressed in a dedicated design document also delivered in quarter 4. A separate design document is also provided in the same quarter to address epoch recovery.

Definitions

- RPC: Remote Procedure Call
- bulk RPC: request involving a bulk transfer, typically a RDMA.
- 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, …)
- MGS: ManaGement Server storing the global Lustre configuration (i.e. list of targets, network identifiers, …)
- LU infrastructure: device and object framework introduced in Lustre 2.0 (see lu_device, lu_object, lu_site, …)
- 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
- transno: transaction number associated with a request. The Lustre server allocates transaction number sequentially when processing requests and reports back to clients the last committed transaction number (it is actually packed in all replies). Any request with a transno smaller or equal to the last committed transno can be deleted from memory by clients since all changes associated with those requests have been safely committed to disk and won’t have to be replayed during Lustre recovery.
- ioctl: I/O control call on a file descriptor, see ioctl(2).
- POSIX: an acronym for Portable Operating System Interface

API and Protocol Additions and Changes

i. Storage Topology

The Lustre metadata server handles object pre-allocation and assignment (namely QoS which stands for Quality of Space) more or less efficiently by taking into account OST relative space usage as well as OSTs to OSS mapping. This model is not flexible and doesn’t allow application-driven allocation policies.

In contrast, the DAOS API exports through the system container very detailed information about the backend filesystem topology. Applications can now take into account parameters like bandwidth, latency, storage type and network proximity when allocating shards, allowing more efficient and fine-grained placement policies. As a result, allocation engine like CRUSH can now be developed in user space to manage data distribution and replication efficiently (i.e. trying to optimize performance and space distribution) and safely (i.e. by respecting failure domains).

In practice, Lustre will implement the DAOS system container API by collecting and aggregating data from the Lustre configuration log, OBD_STATFS and LNET.

1. Integration with Lustre Configuration Log

Storage Topology

The MGS configuration logs store the OST list with NID association. This is used to produce the initial node/OST list.

As for the cage/rack/node map, it has to be created with *lctl conf_param* and is thus stored in the client configuration log as well. This map has to be maintained manually in case of server addition or NID change.

Storage Type, Latency & Bandwidth

The storage type cannot be easily determined by Lustre itself and should eventually be specified manually through lctl or mkfs. The target will then pass the storage type to the MGS at registration time, which will record it in the client configuration log. By lack of time, this hasn’t been implemented in the prototype (DAOS_TARGET_TYPE_HDD is always returned), but could be easily added in the future.

For this project, the target latency and bandwidth are measured at the LNET level when messages are exchanged between nodes. An improvement (not implemented in the prototype) would be to measure the bandwidth and latency automatically at mkfs time (and provide a way to refresh it via tunefs.lustre) or at start-up time.

The LNET distance is used to populate the network distance parameter. The only supported values in the prototype are 0 when the target runs on the same node as the client and 1 when the target is remote.

2. Dynamic information and Change Notification

Target Dynamic Information

The OST size and amount of free space for each target will be retrieved by executing OBD_STATFS RPCs.

As for daos_target_info_t::ti_status field, it could be computed as follows:

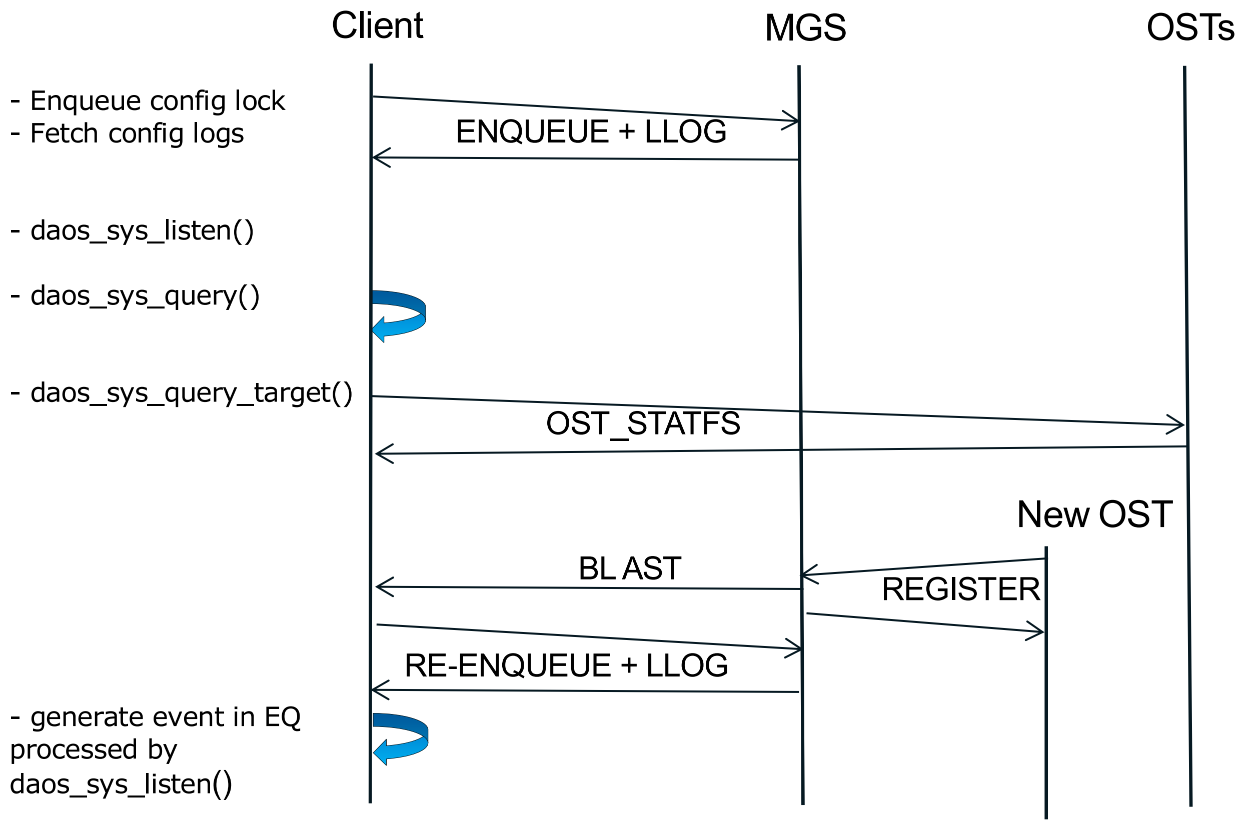
- -1: the target has been administratively disabled in the configuration log

- 0: the import is marked as disconnected or an invalidation is in progress
- the default value is based on the number of reconnects that happened over the last one hundred RECONNECT_DELAY_MAX slots. Let i be the number of reconnects that happened during the last 100 x RECONNECT_DELAY_MAX seconds. The ti_status field will then be set to max(0, 100 – i).

The current prototype always returns 100 for now.

Configuration Change Notification (optional)

DAOS also provides a way for applications to be notified of configuration changes. This is done by registering an event via daos_sys_listen() which will be completed when a configuration change has occurred. As schematized in the figure below, this mechanism will be tightly coupled with the Lustre configuration log management.



When a configuration change (e.g. new OST addition) takes place, the MGS revokes the configuration lock, which is immediately re-enqueued by the client. Once the MGS lock is granted again, the client fetches the configuration changes and reconfigures the client stack accordingly. The event to be generated for daos_sys_listen() will be piggy-backed on these stack reconfiguration notifications.

As for failover notifications, a similar scheme will be implemented with the IR table and lock.

None of those mechanisms have been implemented in the prototype.

ii. Container Abstraction Support

The purpose of this section is to describe how a container is implemented in Lustre.

1. Container Representation on the MDS

A container is an inode managed by the MDS. It is assigned a unique FID (using a sequence owned by the MDT) allocated by the Lustre client that created the container. MDTs support the following set of operations for containers:

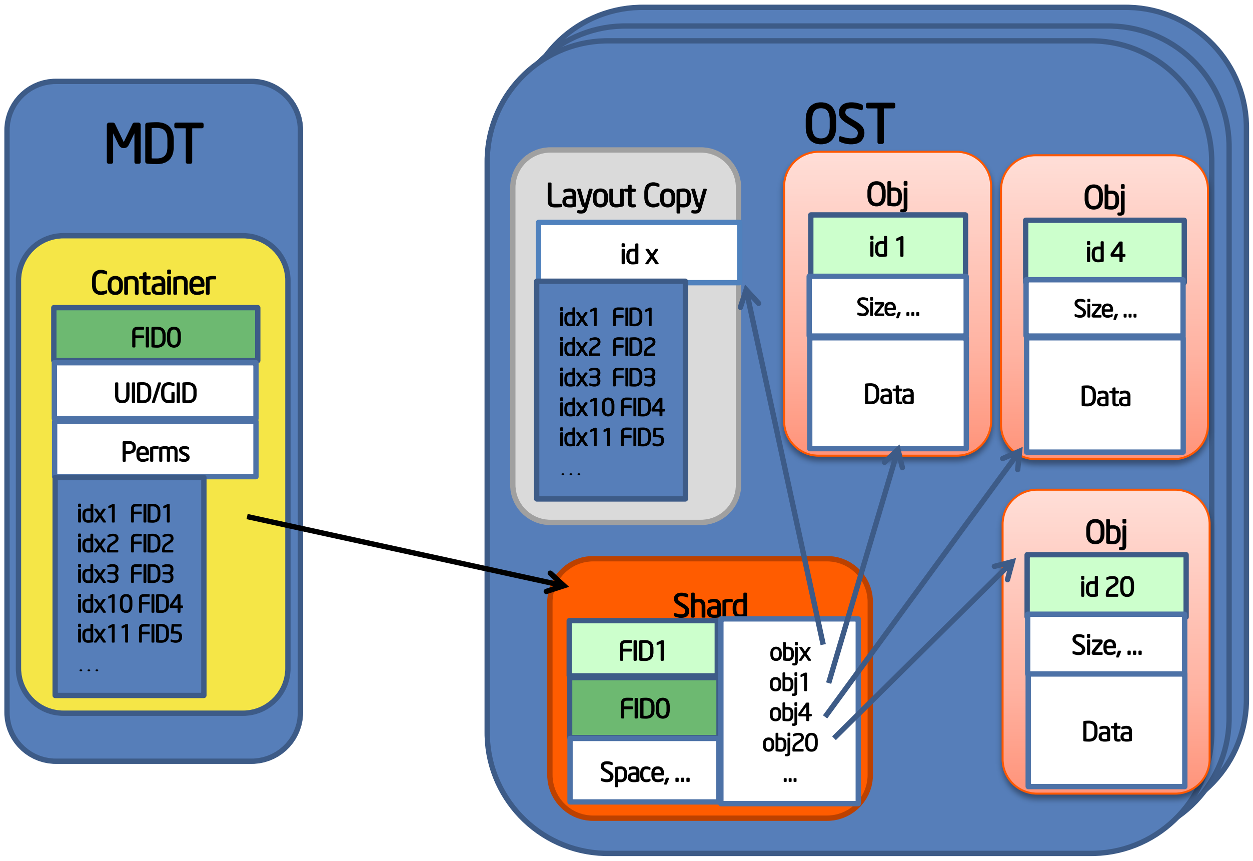
- container creation
  - The MDS allocates the container inode and inserts it into the namespace
- Add shard/disable shard and flush layout update
  - See next section
- container unlink
  - like POSIX files, the MDT handles shard deletions in the background
- container getattr (used by query and stat(2))
  - st_dev/ino/mode/nlink/uid/gid/rdev/blksize work as expected
  - st_blocks returns the total space consumed by the staging dataset
  - st_size isn't meaningful for a container
  - st_[acm]time (optional) returns approximate times (more details in next chapter). This hasn't been implemented in the prototype.
- open/close
  - no POSIX open/close, only calls issued through the DAOS client are supported
- epoch management (commit/wait)
  - MDT tracks committed epochs which are still referenced by container handle and reclaims space on OSTs when the last reference is dropped
  - MDT propagates commit to OSTs when the write opener has committed
- container snapshot
  - Like new container creation, MDT allocates a new inode and inserts it into the namespace
  - MDT clones all the shards (each one get a new FID) and updates the layout (including all the copies) accordingly. More details are provided in the next section.

2. Container Layout

Layout Structure & Storage

A container layout is different from the traditional LOV EA. Unlike regular Lustre files, a container has no striping and can have several shards on the same OST.

As a result, a container layout is composed of a list of shards, each one having an index inside the container. Each shard is assigned a unique FID by the OST it belongs to. The container layout is thus an index of OST FIDs. This layout is replicated on all the shards and is cached on the MDT, allowing a fast reconstruction of the MDT in case of disaster recovery. The figure below summarizes the various data structures involved.



The layout also includes a counter recording the number of shards that are part of the container layout and a generation which is bumped each time the layout is modified.

In addition to a FID and an index, the container layout also stores for each shard:

- the epoch number when the shard was added to the layout
- the epoch number when the shard was disabled, if relevant. If the shard has not been disabled in any epoch, then this field is set to 0.

Besides, the MDS is in charge of caching this container layout (either in the traditional LOV EA or in the container inode itself), which actually includes the whole history of the layout modifications. In the prototype, the layout stored on the MDS is considered as definitive since there is no layout replica on the OST (to be implemented in a follow-on project).

Layout Modification

There are two types of layout modifications that can be performed:

- Shard addition: a new shard is allocated and the shard FID is added at a given index in the layout. The new shard can be used straightaway on the node where it was created, but will only be considered as persistent when the epoch in which this shard was added is successfully committed.

- Shard deactivation: an existing shard is marked as disabled in the layout and is now skipped by the MDT when propagating epoch commit. A disabled shard cannot be enabled again.

In the future (once the layout is replicated), both layout operations require updating the layout copy on all the shards as well as the one cached by the MDT. The MDT is responsible for updating the layout copy on shard during the flush operation. This distributed layout update will be done through a server collective operation. Given that the message that can be sent through a server collective is limited in size (i.e. 1MB), the MDT will set up a separate bulk descriptor and pack it in the collective message. Then each server involved in the collective will be able to fetch the layout through a bulk transfer from the MDT, before acknowledging the server collective.

When a shard is added the MDT allocates the new shard, propagates the writer's capability and current container layout to it and adds an entry for the new shard to a list of pending shard layout changes for the container. When a shard is disabled, the MDT similarly adds an entry to this list of pending layout changes. On layout flush, the MDT distributes these pending layout changes to all the shards. As reminder, a flush is always required before committing (see epoch recovery design document for more information).

As said before, the MDT also updates the layout on the local inode. If this fails due to an I/O error, manual recovery is required. If this fails due to MDT restart or failover, the MDT layout is updated on the next open – most usually when clients replay open on recovery. Recovery details are covered in the epoch recovery design document.

Layout Change Notification

Besides waiting on commit completion of specified epoch, daos_epoch_wait() also uses the layout generation to determine whether the layout was modified in the targeted epoch. This is how readers can find out whether some shards were added or disabled. The updated layout can then be redistributed by rerunning local2global on the master and globa2local on the slaves. Each task can also invoke daos_epoch_wait() which will result in the layout to be refreshed on all the nodes individually. That said, this last option isn't very scalable since all nodes will have to send multiple RPCs to the same MDT which might be overwhelmed.

As for writers, the process initiating the layout change will obviously be aware of the modification and can then fetch the latest layout. With a collective open, local2global and global2local will have to be run on the master and slave after a layout modification in order to refresh the layout. In practice, local2global on the master will fetch the latest layout generation from the MDT (it might not be aware of the latest layout if e.g. a shard was added or disabled from a slave) and refresh the layout from the MDT if needed (i.e. local layout generation is smaller than the one returned by the MDT). The node executing the layout modification updates the layout locally and can always access added shards, even without rerunning global2local.

3. Container Snapshot

A container snapshot results in a new container which has its own FID, inode, layout and entry in the namespace. The layout of the new container is effectively a clone of the original one: it has the same number of shards located on the same OSTs except that shard FIDs are different. The HCE snapshot and the staging dataset of the cloned shards are initialized with the same object content of the original container, except the object storing the layout copy which is set to the new cloned layout. Shards that are marked as disabled in the original container won't be cloned.

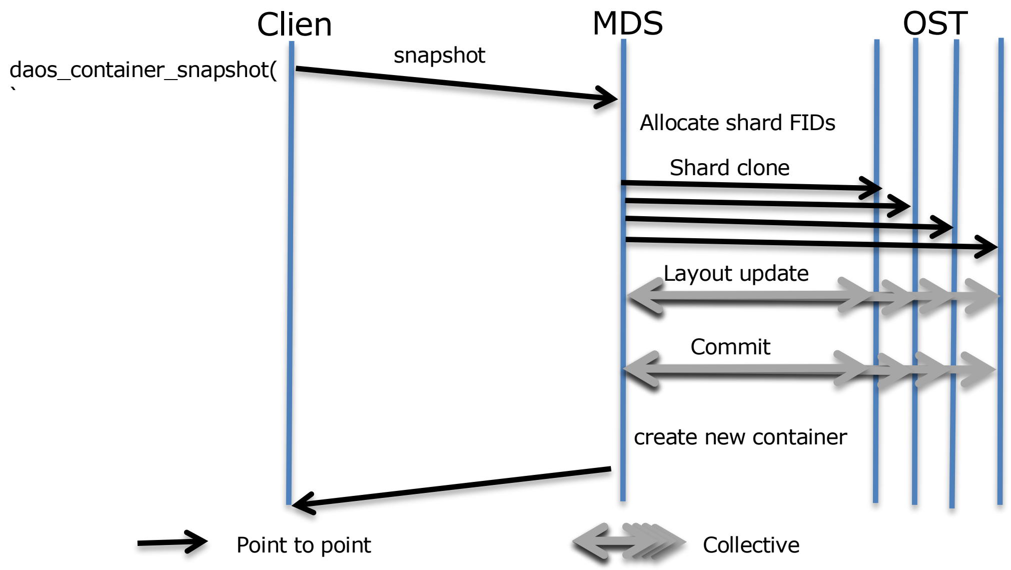
The new snapshot container can be modified independently of the original container. This means that, like any other containers, it can be opened in read & write mode through the usual DAOS API and shards can also be added or disabled.

Besides, containers created through a snapshot operation could have an additional extended attribute recording the FID of the original container and the epoch number used at snapshot time. This extra EA hasn't been implemented in the prototype.

A snapshot request will be processed as follows by the MDT:

- Allocate shard FIDs from the pre-allocated pool and build the new layout.
- Send a shard clone RPC to all the shards of the new layout with the FID of the original shard to be cloned as a parameter. Each cloned shard will initialize its staging dataset with the HCE snapshot of the original shard.
- Set up a server collective(s) (there might be multiple of them if the layout is large) with all the new shards to update and flush the new layout. The layout update isn't executed in the current prototype given that there is no layout replica.
- Commit the epoch where all those changes have been made.
- Allocate an inode for the new container, insert it into the namespace and update the layout copy on the MDT

This processing is schematized in the figure below.



Container Snapshot

4. Capabilities

To prevent illicit access to shards, capabilities are associated with a container handle. The capability list is transferred to all the shards via a server collective operation (see design document dedicated to the spanning tree protocol). In every RPCs sent to servers, clients should pack the container handle, which will be validated against the capability by the OSTs.

On close or eviction, the MDT issues a new server collective to revoke the capabilities (if any) associated with the container handle.

The list of supported capabilities is relatively small for now, but can be extended easily in the future:

- CAPA_SHARD_OBJ_READ
    - Permission to read objects from the shard. This includes listing all the non-empty objects in the shard.
    - Granted to clients who opened the container for read or read-write.
- CAPA_SHARD_OBJ_WRITE
    - Permission to write and punch objects in the shard
    - Granted to clients who opened the container for read-write

Both read and write capabilities are supported in the prototype.

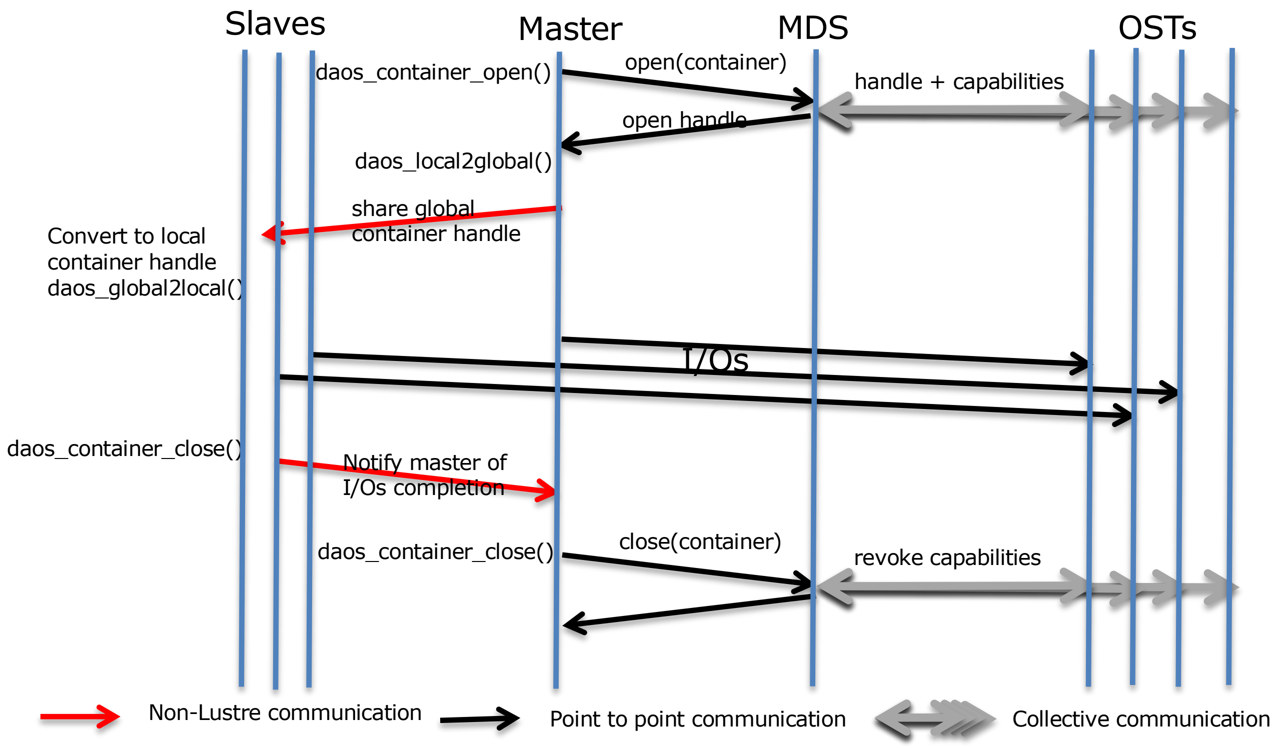
Alternatively, the capability list could be signed with a private key by the MDT and returned to clients directly. This would avoid setting up a server collective each time a container is opened. The client then provides the signed capability list to the OSTs that can check the MDS signature and validate the capabilities. The MDS will still have to distribute his public key to OSTs through a server collective. This alternative approach is considered out of scope and will have to be addressed in a follow-on project if required. In the prototype, the capability list is broadcasted by the MDS to all shards each time a container is opened.

In addition to capabilities, OSTs could also validate that requests are coming from the NID owning the handle. This could be implemented by having the MDT packing the NID of the client when broadcasting the handle and capabilities to OSTs. That said, this might not be compatible with collective open for which the original opener will have to provide a list of NIDs that could use this handle at open time. This mechanism is also considered as out of scope.

5. Collective Open & Open Handle

Collective Open

The DAOS API allows a set of processes running on different client nodes to share the same container handle. This mechanism prevents request storms on the metadata server at application start-up time. A master process opens the container and can then convert the local handle into a global handle (via local2global()) which is distributed to peer processes. The slave processes can then call global2local() to set up the local DAOS client stack without sending any RPC to the metadata server. The figure below summarizes the sequence of RPCs involved in a collective open/close.



Collective Open

It is worth noting that only the master process is allowed to call epoch commit and wait. It is considered invalid to call those functions from a slave process.

Besides, if the MDT evicts the master process, all the handles are revoked and slave processes lose access to the container.

Local & Global Handles

Like file descriptors, a local open handle is just an opaque integer that points at a kernel structure storing data (e.g. reference to container dentry/inode on client) relative to the container handle.

The task of local2global() is to extract all information from the kernel to clone the setup on a different client node without sending any RPC to the metadata server. To do so, the global handle is actually a userspace buffer storing the following information:

- the FID of the container
- the open handle granted by the MDT which has capabilities associated on the shards
- the layout of the container, including all the shard FIDs
- the location (i.e. OST index) of each shard to prevent FLDB requests to the MDS.
- the current HCE
- (optional) the capability list signed by the MDS. Required if capabilities are returned to clients and not communicated directly to shards by the MDT.

All this will be packed into an opaque structure in the userspace buffer which can then be sent by the application to another client node where global2local() will set up the local DAOS client to access the container. The format of the userspace buffer should be versioned (i.e. through a magic number) to allow format changes in the future.

Moreover, as suggested in the previous sections, when local2global is run on the master, this latter verifies whether it has an up-to-date layout and refreshes it if needed. This guarantees that any call to local2global/global2local after a shard addition/deactivation on one of the slave processes always propagates the latest layout.

iii. Lustre DAOS Client

DAOS defines a new object-based API fundamentally different from the legacy POSIX interface. The purpose of this section is to describe all the Lustre client changes required to support this new API.

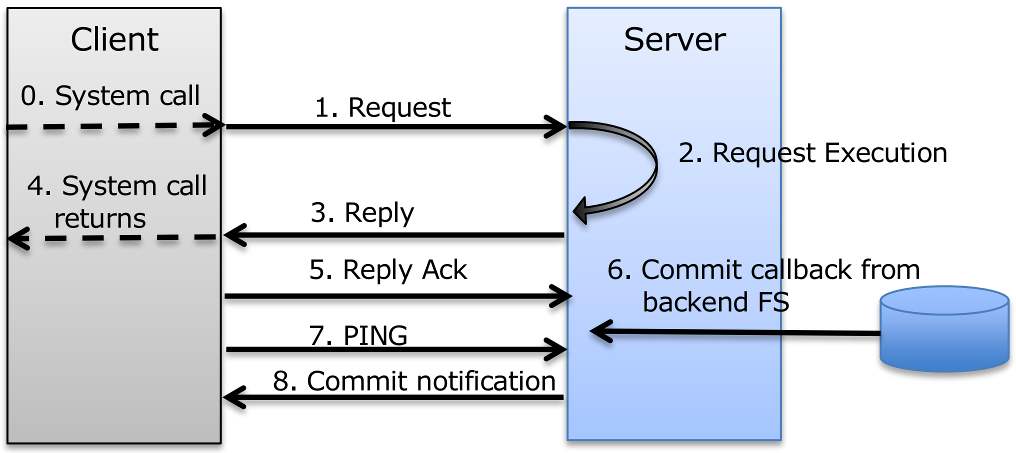
1. Asynchronous I/O Support

Buffered writes aside, most filesystem operations supported by the Lustre client today result in RPCs sent synchronously to Lustre servers. In other words, applications are stuck in system calls until Lustre servers process network requests. Moreover, POSIX imposes strong locking constraints that make truly parallel non-blocking I/O very difficult to support. Even cached writes might not always be fully asynchronous since the system call can still block waiting for requests in-flight to complete (e.g. not enough grant space available on the client or maximum per-OSC dirty page limit reached). An asynchronous I/O interface allows applications to take advantage of computing cycles that are traditionally wasted, waiting for filesystem resources to be available.

POSIX defines a standard for asynchronous I/O (see aio(7)) which is unfortunately only used for read/write/fsync and doesn't cover metadata operations. In addition, the DAOS event API is richer and more generic than the POSIX one.

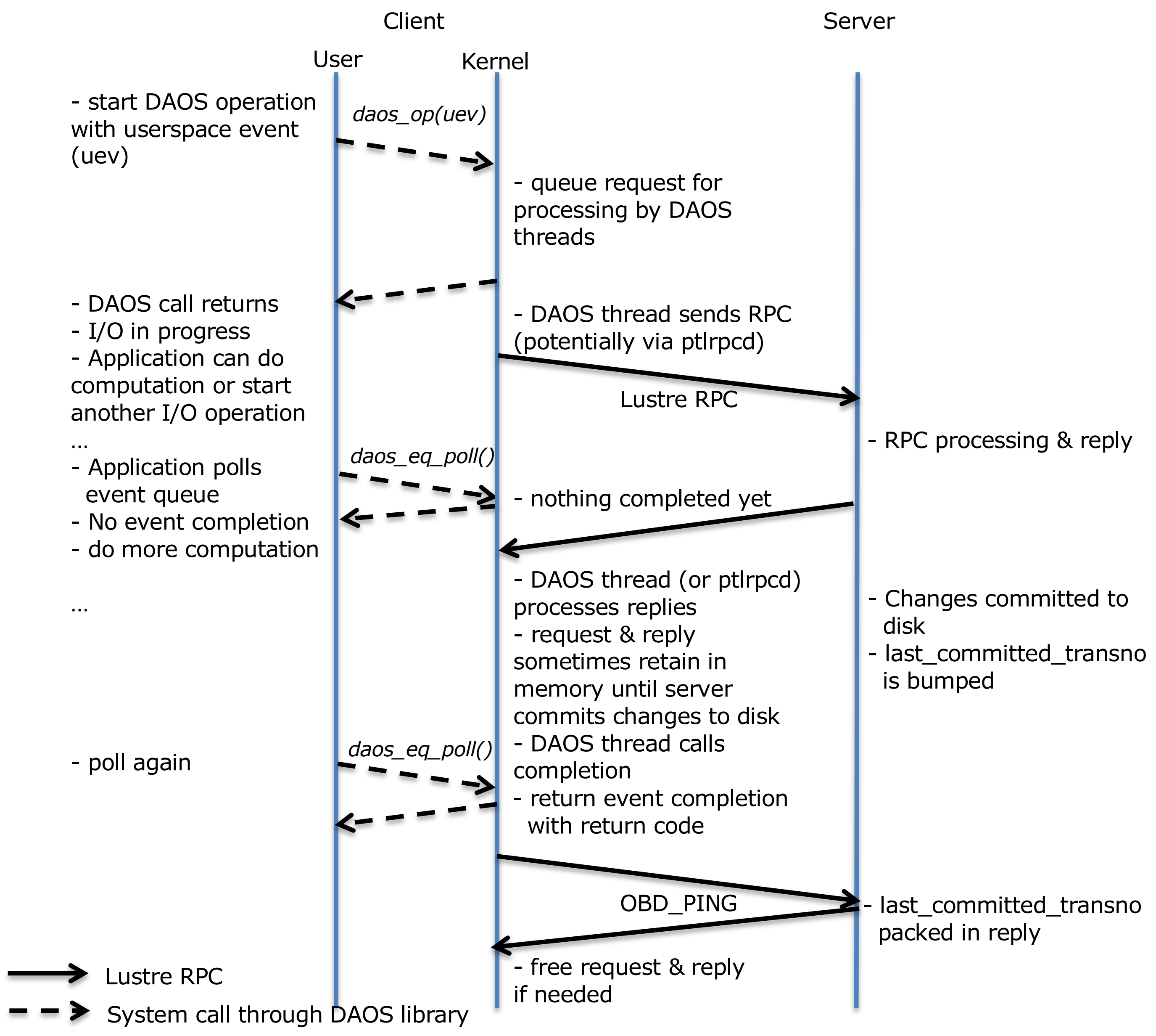
Request Lifecycle & DAOS Operation Flow

The figure below represents the typical request lifecycle in place today in Lustre. The application sends RPCs and waits for server processing. Once RPCs are completed, the system call returns but the Lustre client still keeps the request and reply in memory, ready for replay on server recovery until the server confirms that changes are committed to disk. The last committed transaction number is returned in every RPC reply to the client since disk commit happens asynchronously.



Lustre Request Lifecycle

The figure below represents the generic workflow of a DAOS operation. A pool of DAOS thread takes over the DAOS operation and sends RPCs over the wire (sometimes via ptlrpcd). The DAOS library call then returns immediately and the application can continue with other processing (e.g. computation or more I/O operations submission). Once the client receives the reply, the event completion callback is invoked and marks the event as completed. Next time the application polls the event queue associated with the DAOS operation, it will be notified of the I/O completion. Some DAOS RPCs might be retained in the client memory for recovery. This is the case for metadata requests (container/shard operations) which use regular lustre replay mechanisms, but not for I/O operations which will have to be replayed from the burst buffer.


Client
Server
User
Kernel
- start DAOS operation with userspace event (uev)
daos_op(uev)
- queue request for processing by DAOS threads
- DAOS call returns
- I/O in progress
- Application can do computation or start another I/O operation
…
- Application polls event queue
- No event completion
- do more computation
…
- DAOS thread sends RPC (potentially via ptlrpcd)
Lustre RPC
- RPC processing & reply
daos_eq_poll()
- nothing completed yet
- DAOS thread (or ptlrpcd) processes replies
- request & reply sometimes retain in memory until server commits changes to disk
- DAOS thread calls completion
- return event completion with return code
- Changes committed to disk
- last_committed_transno is bumped
- poll again
daos_eq_poll()
OBD_PING
- last_committed_transno packed in reply
- free request & reply if needed
Lustre RPC
System call through DAOS library

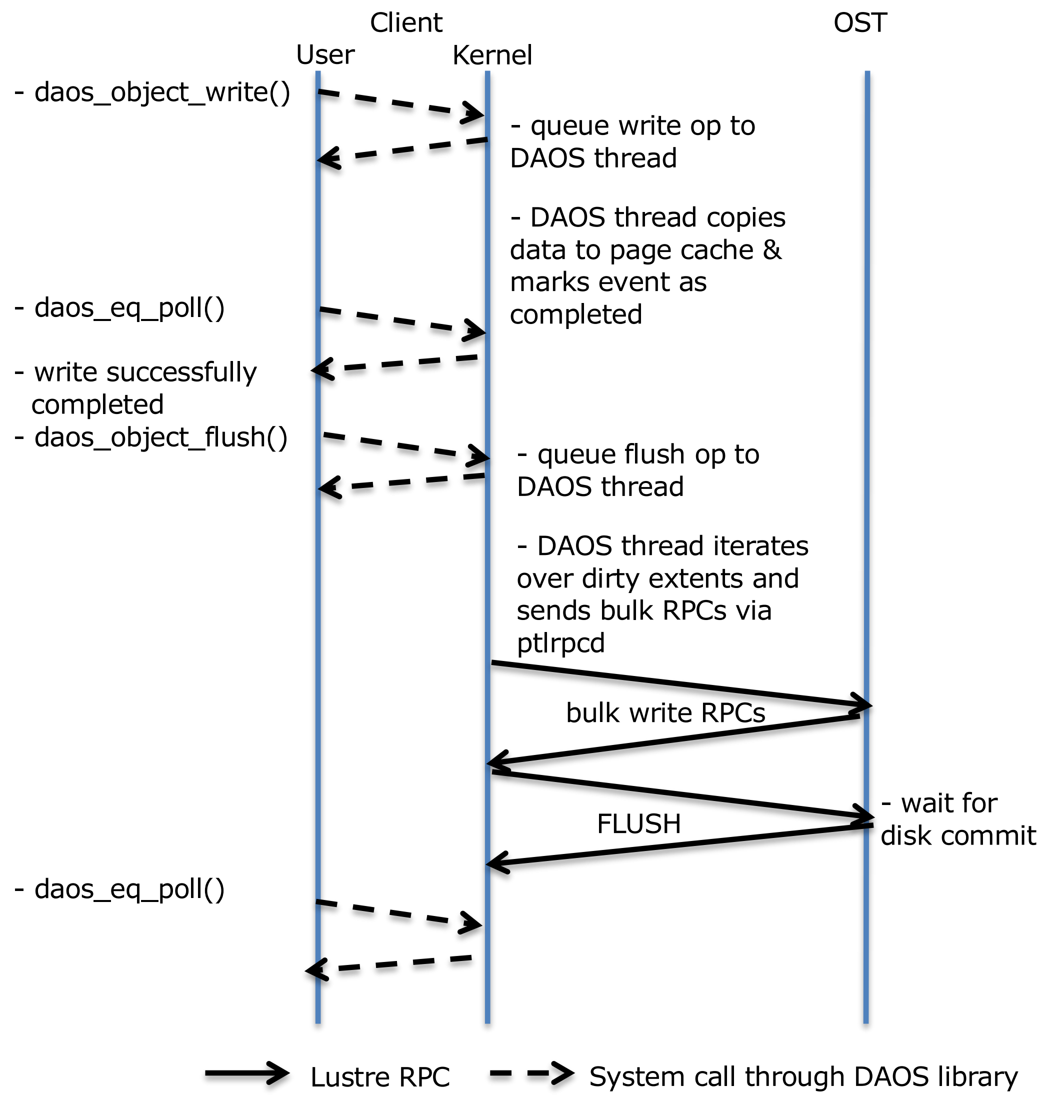
DAOS Operation Flow

As a result, the application gets the event completion once the server has processed (via intent log or directly in staging dataset) the request, but before changes are committed to disk. A different approach is taken in case of server restart depending on the request type. For metadata requests (i.e. shard addition/disable, container creation, …), the usual Lustre recovery mechanisms apply in case of failover or server restart to replay uncommitted disk changes. As for data requests (i.e. object I/Os), RPCs are not retained for replay which means that uncommitted changes can be lost and have to be replayed from the burst buffer by IOD. Epoch-based recovery is addressed in a separate design document.

Object Write Operation

The workflow described in the section above is slightly different for object write. The Lustre client may (depending on open flags) either copy data from userspace to the page cache and send bulk write RPCs asynchronously or map the userspace buffers and send it directly over the network, similar to direct I/O.

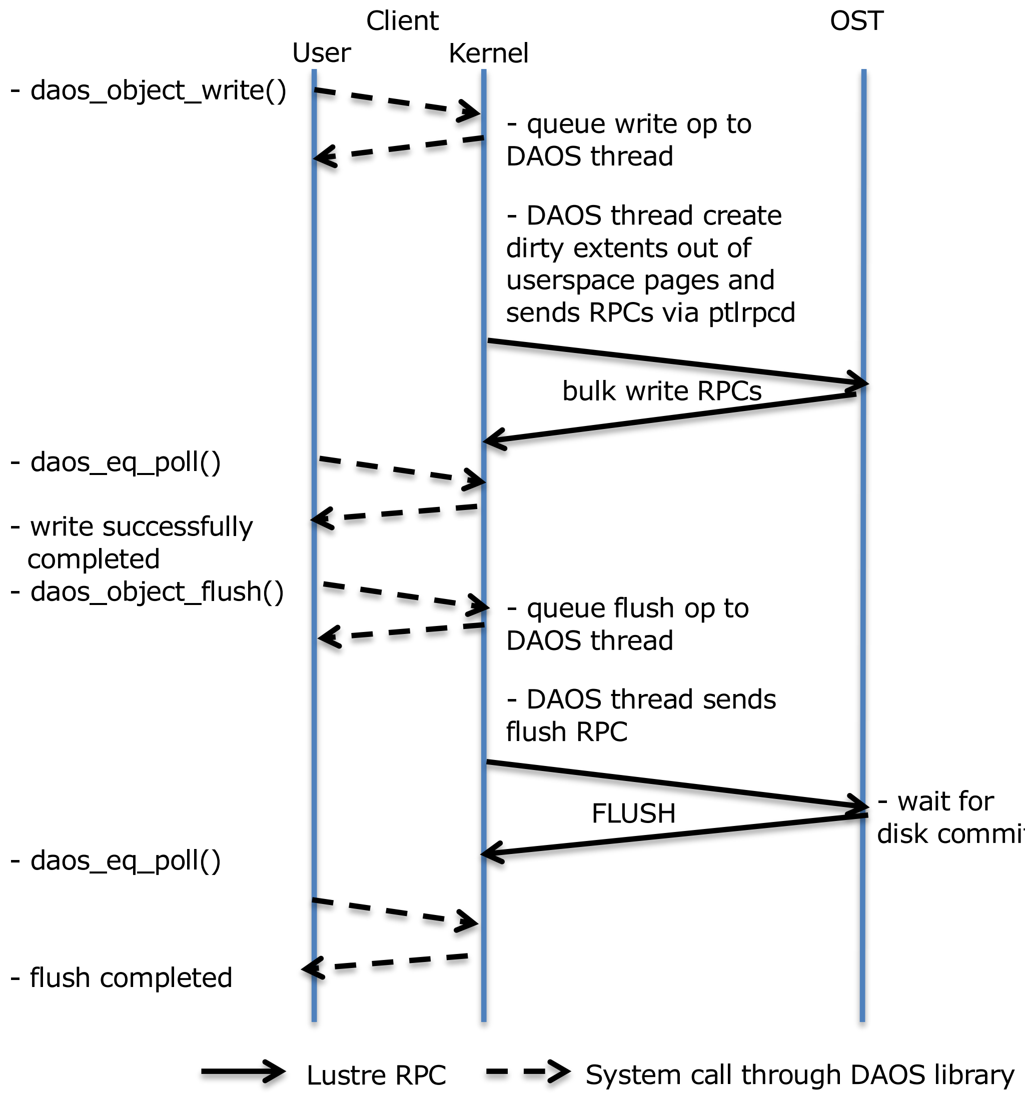
In the former case, the event completion is raised as soon as the copy from userspace is completed. A call to daos_object_flush() is then required to trigger dirty page writeback which completes once all bulk write RPCs have been processed by the OST. The figure below summarizes the event sequence.


Client
User
Kernel
OST
- daos_object_write()
- queue write op to DAOS thread
- DAOS thread copies data to page cache & marks event as completed
- daos_eq_poll()
- write successfully completed
- daos_object_flush()
- queue flush op to DAOS thread
- DAOS thread iterates over dirty extents and sends bulk RPCs via ptlrpcd
bulk write RPCs
FLUSH
- wait for disk commit
- daos_eq_poll()
Lustre RPC
System call through DAOS library

DAOS Cached Write Flow

The flush operation has to wait for (or potentially force) disk commit (i.e. last_committed_transno to be higher or equal to the transno of the last write RPC) in order to deal with failures of both the OST and client just after the commit request is issued by the client (more details are provided in the epoch recovery document).

As for the latter case (i.e. direct write), the completion event is delayed until the OST has processed the bulk write RPCs. A flush call will also be required to guarantee that changes have been committed to disk. The figure below represents the sequence of events for a direct I/O write.


Client
OST
User
Kernel
- daos_object_write()
- queue write op to DAOS thread
- DAOS thread create dirty extents out of userspace pages and sends RPCs via ptlrpcd
bulk write RPCs
- daos_eq_poll()
- write successfully completed
- daos_object_flush()
- queue flush op to DAOS thread
- DAOS thread sends flush RPC
FLUSH
- wait for disk commit
- daos_eq_poll()
- flush completed
Lustre RPC
System call through DAOS library

DAOS Direct Write Flow

As previously mentioned, write and punch RPCs are not automatically replayed by DAOS Lustre. If I/Os turn out to be lost after a server reboot, the next flush operation will fail with EIO and it is up to the application to resubmit I/Os since the last successful flush.

Besides, given that a single bulk RPC is limited to a 1MB transfer (4MB in 2.4 and up to 32MB in later releases once the allocation issues are fixed), object operations (except punch) might result in more than one RPC sent over the wire. The max_rpcs_flight limit as well as the max_dirty_mb threshold might require copying data from userspace and issuing bulk RPCs in multiple waves. This is supported thanks to the DAOS thread pool.

More details on read and write caching are presented in section 4.

2. **Event, Event Queue and Handle**

DAOS userspace entries

DAOS API cannot really map to POSIX filesystem calls, even those routines that have similar functionalities will have totally different parameters (most of them require event), which means ioctl is the main entry of most DAOS APIs.

This project implements two POSIX file types and reuse Lustre directory as DAOS API entries:

- *Lustre directory*

  DAOS container create and unlink can be entered via ioctl of Lustre directory because container can be created under any Lustre directory.

- */dev/daos file descriptor*

  It is a misc-device that plays the role of calling entry of these APIs:

  - All event queue (EQ) and event APIs
    Instead of defining a new file type and implementing poll interface for EQ, it is much easier to just implement EQ and event APIs with ioctl, which can bypass overheads of: a) add new file type to VFS framework for EQ; b) covert POSIX style APIs to DAOS APIs in userspace.

  - Storage system APIs
    Because /dev/daos represents the whole distributed storage system, so it is reasonable to choose ioctl of /dev/daos as entry of storage system APIs.

  - Collective functions
    They are new routines, also, daos_global2local can not map to any other opened entity in namespace, so it is straightforward and simple to just reuse ioctl of /dev/daos as entries.

- *Container file descriptor*

  DAOS container has a new file type and has its own VFS interfaces because a container is a visible entity in POSIX namespace, and it can be queried via POSIX file stat. So instead of putting everything under ioctl of /dev/daos, these APIs can be entered via ioctl of container file descriptor:

  - Container operations
    daos_container_listen() is listening on container handle, daos_container_query() is supposed to query shards and other information of container by open handle, so they should be entered via ioctl of container file descriptor.

  - Epoch functions
    Epoch functions take a container handle as a parameter, all its APIs should thus be built within ioctl of container file descriptor.

  - Shard and object operations
    Shard and object are not visible to POSIX namespace, there is no reason to add file type and open file descriptor for them. At the meanwhile, logical identifiers of

shard and object are managed inside container, so it is natural to choose container file descriptor as entry of shard/object routines.

DAOS handle table

DAOS handle is a 64-bit value, it is identifier of kernel space instance of EQ/container/epoch-scope/object. DAOS handle might not be safe to be inherited by child processes, for example, if a process created a EQ then forked a child process, then child process can get unknown results when try to poll from that EQ.

As previously mentioned, not all DAOS entities have POSIX file descriptors, which means DAOS needs to manage its own handles table. There are two options for managing handle table: a) each process has its own handle table, b) A global handle table. The first option is very scalable, but handle table has to be passed together with handle to all DAOS kernel functions, which is not very convenient. The second way is simpler, but it will rely on a scalable implementation of hash table, fortunately, cfs_hash of Lustre can has two levels hash structures and it is very scalable, so the second way is adopted for now.

A 64-bit DAOS handle contains three parts:

- 3 bits for handle type (EQ, container, object and reserved types for future use)
- 7 bits for PID (process ID) low bits, which is the first level hash key.
- 54 bits to store an incremental value, it is also the key of the second level hash. Even DAOS can consume 16 millions handle (of a specific type) per second, it will take more than a hundred year to wrap cookie value, so it is plenty enough to ensure handle to be unique.

DAOS Event Queue (EQ) and event

As described in DAOS solution architecture, DAOS API supports concurrency by using non-blocking initiation procedures and completion events, which means all operations that cannot complete immediately require user to pass an event pointer into userspace library, this event pointer is the pre-allocated space to store returned completion status from kernel space. Also, there are a few reserved bytes in daos_event_t structure, they are supposed to be used by DAOS userspace library to store information like EQ handle, kernel space event ID etc.

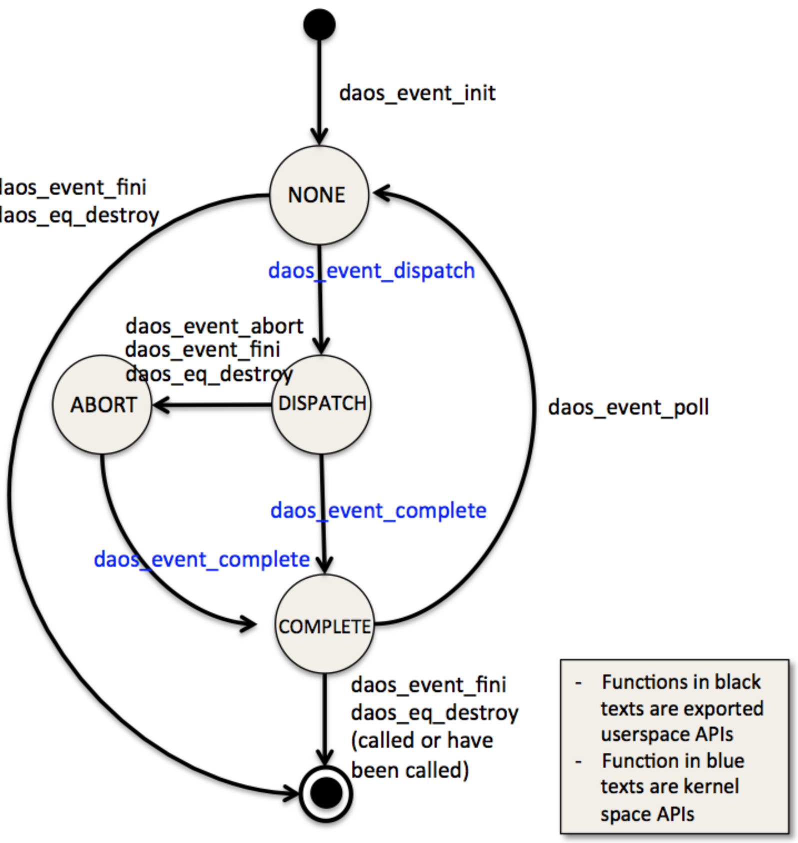
Because kernel cannot directly operate on spaces allocated in userspace (e.g. list operations), so it is meaningless to maintain relationships between EQ and event or event and parent event in userspace, kernel DAOS has to maintain its own EQ/events anyway.

Information stored in userspace event are just IDs, and they should be passed into kernel space via DAOS API ioctl entries, kernel DAOS can find corresponding EQ or event.

Kernel event will be allocated on calling of daos_event_init(), and it will be freed on calling of daos_even_fini(). After called daos_event_init(), event will be given an ID by kernel DAOS, this ID will be copied into reserved bytes of userspace event, each time user called DAOS API with this event, this ID will be copied back into kernel space as searching key of kernel event. Event ID contains two parts: a) EQ handle; b) unique ID in EQ.

There are two reasons that event ID is in domain of EQ, not in global handle table. The first reason is event can be consumed much faster than any other handles, which means 54 bits might not be sufficient to ensure unique for long term. The second reason is, each EQ needs a lock to protect EQ/event status anyway, from the aspect of performance, it is better to allocate/manage event ID under the same lock.

A DAOS event has four different states: NONE, DISPATCH, ABORT, COMPLETE. This graph is state machine of a DAOS event:



DAOS event state machine

As showed in this graph, state transition is triggered by DAOS EQ/event APIs and two kernel(only) APIs:

- *daos_event_init*
  Initialize userspace event, and create a kernel space event (k-event) as shadow of this userspace event (u-event).

- *daos_event_fini*
  Finalize a userspace event and try to destroy the kernel space shadow event.

    - If the kernel event is idle (status is NONE), it is freed immediately
    - If the kernel event is ready to be polled, it will be silently removed from EQ and freed, polling process will not see it.
    - If the kernel space event is still inflight, this function will implicitly abort the inflight operation, in this case, the event will not be

immediately destroyed, instead it will be released by daos_event_complete() (see below descriptions).

- *daos_event_abort*
  Abort inflight operation associating with this event. After underlying operation has been aborted, this event can be returned to user by daos_eq_poll with EINTR as error code of u-event.

- *daos_event_next*
  Because DAOS userspace library will not maintain relationships among EQ/events, so this function has to call into kernel space and locate both parent and current child events with event ID, then return the next event.

- *daos_eq_create*
  Create an empty EQ

- *daos_eq_destroy*
  Destroy the EQ immediately if it is empty, otherwise it will abort all inflight events, and free all idle and completed events.

- *daos_eq_poll/query*
  K-event will save address of u-event on initializing (daos_event_init), so daos_eq_poll/query can return the u-event pointer of a k-event. daos_eq_poll/query will also copy returned value, RPC results of operation to userspace.

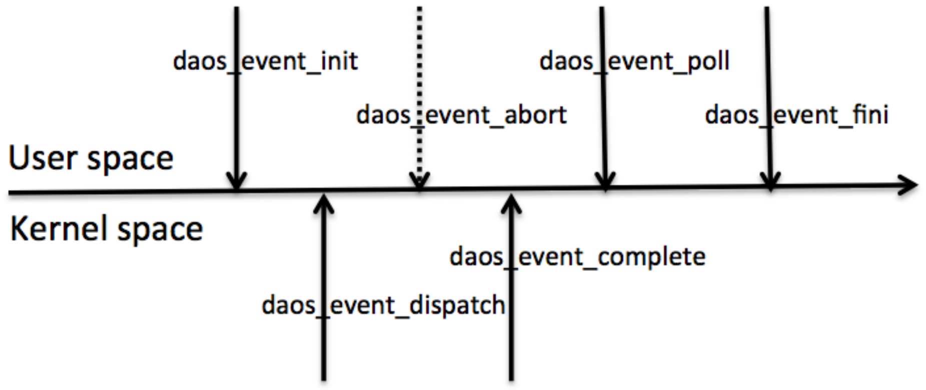
- *struct daos_kevent *daos_event_dispatch(daos_event_id_t eid, void *param, daos_kevent_ops_t ops)*
  This is a kernel (only) API, which is used by other DAOS kernel components, it will be called when a DAOS operation is submitted with this event as a parameter.
  This function will find specified event, change its status to "DISPATCH", and keep a reference of it. This event can be attached on a RPC, the reference can only be released by calling daos_event_complete() on completion of RPC.
  The first parameter of this function is event ID.
  The second parameter is a pointer which will be stored in daos_event::ev_kparam, for example, it can be a pointer to RPC request, this parameter will be passed into callbacks in the third parameter.
  Type of the third parameter is:

```
typedef struct {
        int (*op_abort)(void *kparam, int unlinked);
        int (*op_complete)(void *kparam, void *uparam, int unlinked);
} daos_kevent_ops_t;
```

  daos_kevent_ops_t ::op_abort() is callback for daos_event_abort(), e.g. how to abort a request. daos_kevent_ops_t ::op_complete() is callback for copying replied data to userspace, and release reference on RPC. In normal case, op_complete() is called under thread context of user process because it is the safe context of copying bytes from kernel-space to userspace, which means it is called from daos_eq_poll(). But if this event has been unlinked, e.g. the process is dead and event have been unlinked on closing of file descriptor of /dev/daos, then op_complete() can also be called from daos_event_complete() under kernel thread context (e.g. ptlrpcd), in this case, "unlinked" flag will be passed into this callback so it can skip memory copy to userspace.

- *void daos_event_complete(struct daos_kevent *ev, int error)*
  Release event reference obtained by daos_event_dispatch(), and change event status from "DISPATCH" to "COMPLETE". This function will also wake up polling process. If this event has already been unlinked by user, then this function will call daos_kevent_ops_t ::op_complete to finalize things like release reference of RPC, and free event.

This graph is regular lifetime of a DAOS event, it is another way of showing how user can use DAOS EQ/event APIs.



DAOS event lifetime

3. DAOS Client I/O Stack

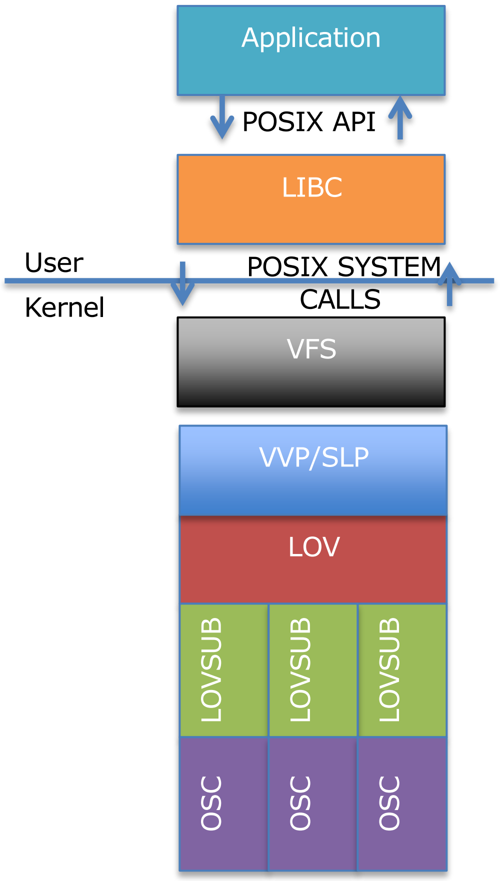
The absence of striping and locking makes the DAOS I/O path much simpler than the Lustre POSIX client. There is indeed no need to split I/Os on stripe boundaries and more generally to use CLIO iterations (i.e. cl_io_loop()). As a result, the DAOS bypasses most of the CLIO complexity by interacting directly with the sub-object (i.e. LOVSUB & OSC layers), similarly to the echo-client.

The DAOS client provides handlers for ioctls executed against /dev/daos (see previous section) and will take care of:

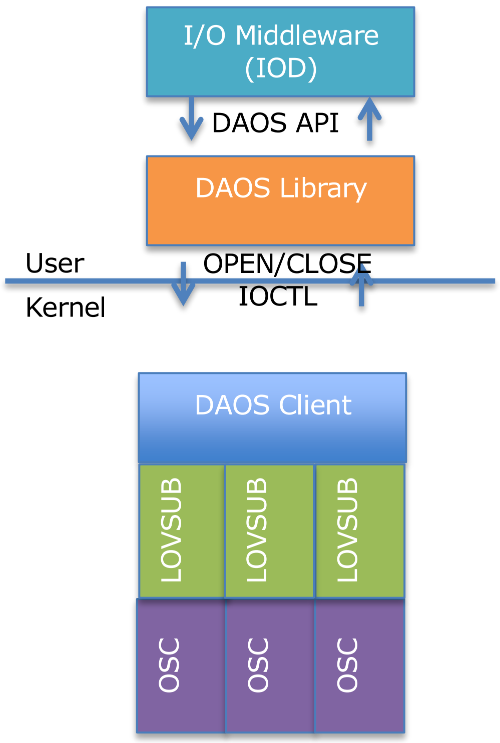
- Event and event queue management
- DAOS handle allocation and management
- Container layout management (including collective open)
- Forward all DAOS metadata operations (e.g. container create/destroy, …) to LMV/MDC
- Forward all DAOS data operations (shard query, object read/write/punch/flush) to OSC.

As for the OSC layer, it will be modified to implement DAOS object (read/write/punch/flush) and shard (information query) operation support. The shard-aware OSC will also disregard grant space reservation since ENOSPC can be returned during DAOS flush operation. The LOV layer is still used for the container layout management.

The figures below compare the DAOS and POSIX client layering.



Lustre POSIX client



Lustre DAOS client

4. Client Caching and Writeback

Although IOD with the burst buffer provides the first level of caching and coalescing, the Lustre DAOS client still has its own readahead (not implemented in the prototype) and RPC generation algorithms to optimize I/O throughput. Those optimizations are driven by the flags passed to daos_object_open() which are DAOS_OBJ_IO_RAND and DAOS_OBJ_IO_SEQ respectively for random and streaming I/Os.

RPC Engine

The Lustre RPC engine will be reused to generate bulk read/write request. That said, the RPC generation algorithm will be simplified as follows:

- no grant reservation is required and thus no need to take block alignment into account for grant calculation
- no need to care about having a single lock covering the whole extent

- no special processing is required for lockless (actually server-side locking) I/O since no extent locks are used in DAOS

Besides, the following new functionalities will be introduced:

- Due to the lack of extent locking and read-modify-write on the client, extents won't necessarily be aligned on page boundaries any more
- The extent tree has to support multiple versions (effectively one per epoch) of the same extent for each DAOS object. The new data structure is actually detailed in the next sections.

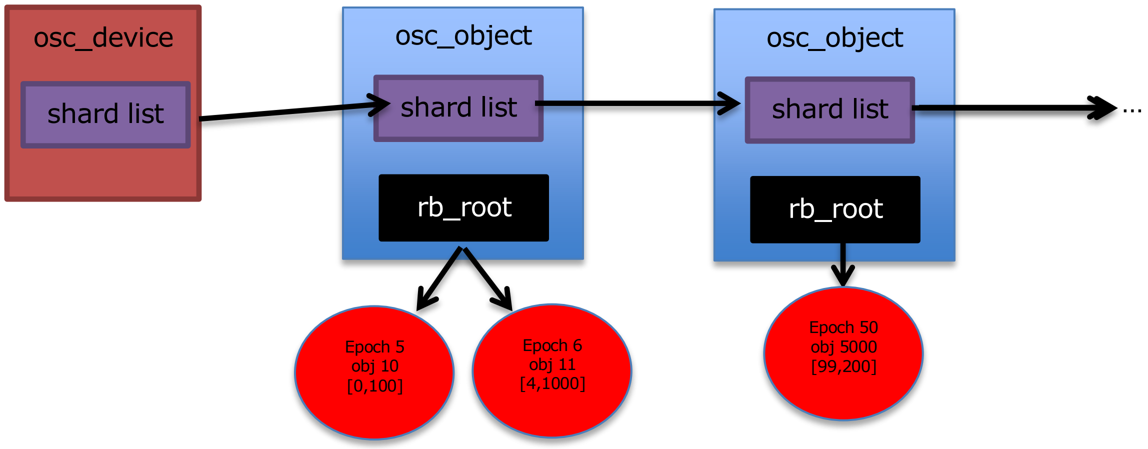
An additional constraint is that a bulk read/write request can only include extents from the same DAOS object, for the same operation type (read or write) and for the same epoch. Support for multi-object bulk read/write in the same epoch is considered as out-of-scope for this prototype, but might be implemented in the future to improve random I/O support.

Writeback

The epoch number in which this extent has been modified

- The DAOS object ID the extent belong to
- The start and end offset of the extent

Moreover, each OSC device maintains a list of OSC shard objects stored on the same OST.



Calls to daos_object_write() generate dirty extents that are inserted at the right index inside the red-black tree. Moreover, ephemeral buffers (i.e. anonymous pages freed on request completion) are allocated in kernel space to copy data from user space. When a new extent is inserted, front and back merge with nearby extents (which are necessarily from the same epoch) will be attempted in order to facilitate RPC generation.

Although rare, extent overlap in the same epoch might still happen. Such conflicts are effectively resolved by shrinking one of the extents and merging them.

Unlike the POSIX client, the RPC engine will have a different behaviour depending on the object open flags:

- DAOS_OBJ_IO_SEQ causes the engine to wait for enough dirty data to fire a 1MB (or 4MB/32MB) request and to take 1MB alignment into account.

- DAOS_OBJ_IO_RAND makes the engine less patient and send bulk write RPCs eagerly.

Support for DAOS_OBJ_IO_RAND hasn't been implemented in the prototype and the default behaviour is the one associated with DAOS_OBJ_IO_SEQ.

Memory pressure on dirty extents is handled by registering a shrinker callback, which triggers RPC generation and release pages upon completion. This hasn't been implemented in the DAOS prototype as well.

As for the shard flush operation, the RPC engine traverses the red-black tree and issues bulk write RPCs until it reaches extents for a higher epoch.

Read Cache and read-ahead (not implemented in the prototype)

Likewise to writes, the client read cache has to become epoch-aware. A client can always safely keep read data in cache and return it to daos_epoch_read() as long as all reads are done in the same epoch. On epoch wait, if the object happens not to be modified in the new epoch, the cache can actually be used again to serve reads for this epoch. The following infrastructure will be provided by OSTs to allow clients to quickly determine whether cached data for a DAOS object can be carried over to a later epoch:

- Similar to VBR, OST stores on-disk in an inode attribute (namely last_committed_epoch) the last epoch that modified the object
- Each bulk read request packs in the reply the last_committed_epoch attribute

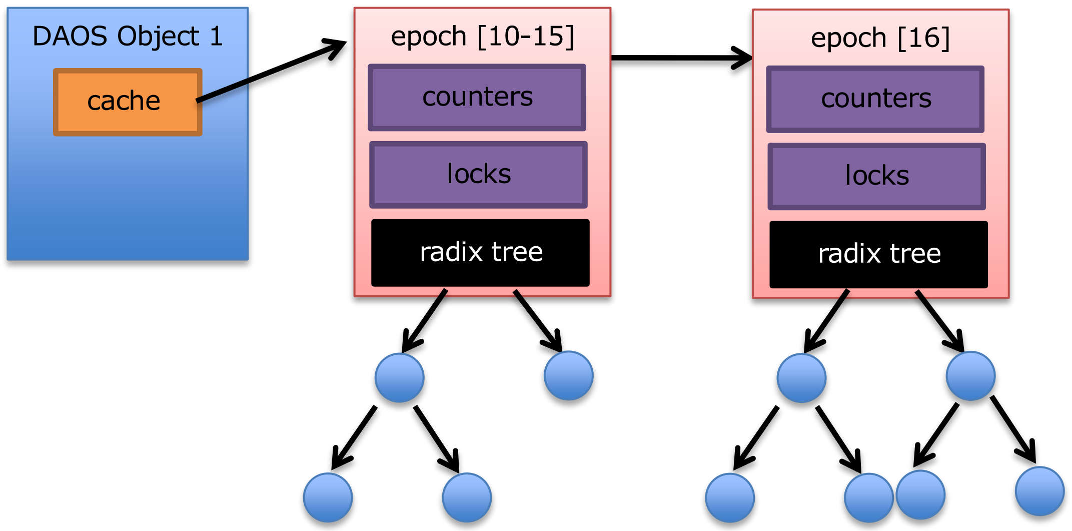
Then, on epoch wait, a client can issue a read request (or alternatively just a getattr RPC) and find out from the reply whether the cache should be refreshed by comparing the last_committed_epoch with the epoch of the cached data. As a consequence, cache invalidation is managed for the whole object, and not on a per-extent basis.

The data structures required to manage the read cache is quite different from the one used for the writeback cache. Each object still maintains a list of "per-epoch" (it is actually an epoch range) data consisting of:

- A radix tree storing cached pages that have been read
- A list of extents to be read. This list is used by the RPC engine to generate bulk read RPCs.
- Counters of pages referenced by the radix tree
- Locks to manage concurrent access to the radix tree and counters
- An epoch interval over which this cache is known to be valid (object hasn't been modified)

The figure below provides an example of an object caching two different versions of the same DAOS object:

- the first version was read in epoch 10 and has been carried over to epoch 15 since the object hasn't been modified in between
- the object was then modified in epoch 16 and a radix tree for this epoch was created and populated by bulk read RPCs



The read cache for epoch 16 might then be carried over to a later epoch if the object wasn't modified. This will be done after a call to epoch wait, on the first DAOS read request which will return the last_committed_epoch attribute.

As for the cache for epoch [10-15], it will be released eventually when all local container handles have slipped past epoch 15 or if pages haven't been used for a while or if the shrinker is called. Several optional optimizations could then be considered:

- an inode mapping could be created for each radix tree, which would allow to use the regular page cache LRU to manage memory pressure.
- another optimization to save memory would be to allow the radix trees of epoch [10-15] and epoch [16] to share pages that haven't been modified, effectively implementing deduplication in memory by comparing page checksums.
- one could also pack the "last_slipped_epoch" in the reply as well. This way, clients participating in a collective open can be proactively notified that there is no need any more to cache data for epoch range [10-15] in the example above.

The Lustre DAOS client could also support read-ahead inside the same epoch if the DAOS object was opened with the DAOS_OBJ_IO_SEQ flag. In this case, the RPC engine will prefetch data and store it in the epoch radix tree so that the application can directly read data from the cache. Obviously, no read-ahead will be executed for random I/Os, i.e. DAOS_OBJ_IO_RAND.

Until the read cache is implemented, all reads will be non-cached, similar to direct I/O.

5. Container Representation in POSIX Namespace

On the client, a container is represented by an inode of type DT_REG with no file operation (it can't be opened via open(2)) and a very limited amount of inode operations:

- ->getattr for stat(2) support which returns the following attributes cached on the MDS:
    - st_dev, st_ino, st_mode, st_nlink, st_uid, st_gid, st_rdev, st_blksize work as expected.
    - st_blocks returns the total space on last successful commit.
    - st_size returns st_blocks * blocksize

- st_atime (not supported in the prototype) returns an approximate time of last access to any container shard.
- st_mtime (not supported in the prototype) returns the most recent time any shard of the container was modified or a new shard was added.
- st_ctime (not supported in the prototype) returns the most recent status change of the container including modifications or inode information.

- ->unlink for unlink(2) support which behaves like daos_container_unlink(). The MDS will take care of destroying the shards. Open-unlinked container works in the same way as POSIX files.
- ->rename for rename(2) support

Those operations will result in a RPC sent to the MDS and won't require setting up the container stack on the client.

iv. Protocol Changes

The purpose of this section is to describe the new RPC types that need to be added to the Lustre protocol to support the DAOS API.

1. Container Operations

Container open, close and unlink will rely on the usual MDS_REINT and MDS_CLOSE RPCs. No significant changes over the wire are expected since, like regular files, a container is uniquely identified by its FID. That said, the container layout is never returned on open and is now transferred through a separate RPC type using a (or even multiple) bulk transfer(s) like readdir. This new RPC uses an OBD_IDX_READ RPC.

A new RPC type will also be added for container snapshot, which will result in clone RPCs to be sent to all the shards from the MDT.

Capability will also be propagated and revoked through a new set of RPCs (i.e. OST_SHARD_OPEN/SLIP) which will be based on the new server collective API.

2. Shard Operations

A new RPC type handled by the MDT will be introduced for shard addition (i.e. MDS_SHARDS_ADD) and deactivation (i.e. MDS_SHARDS_DISABLE). The epoch number will be specified as an attribute of the shard body request buffer. This new RPC actually takes a list of shards to be created/disabled which might be transferred through a bulk.

New RPC types will also be added for shard precreation (i.e. OST_SHARD_CREATE) and destruction (i.e. OST_SHARD_DESTROY). A shard is identified over the wire thanks to its unique FID. Shard clone will also be implemented through a new RPC type relying on server collectives (i.e. OST_SHARD_CLONE).

A new shard flush RPC will also be introduced (i.e. OST_SHARD_FLUSH). Both MDTs (i.e. for layout updated) and OSTs (i.e. for object operation) will have to define handlers for those new request types.

3. Object Operations

The operations that can be performed on a DAOS object are very limited for now, only read, write and punch. That said, we also would like to support different access methods in the future like key/value store.

Blob Objects

OST_READ, OST_WRITE and OST_PUNCH RPCs are reused for shard object operations. The FID packed in the request will be set to the shard FID and the epoch number as well as the DAOS object ID will be passed via the ost_body structure.

Moreover, a new RPC type will be added for shard flush operation (i.e. OST_SHARD_FLUSH).

Key/Value Stores (optional)

DAOS index objects would have to support bulk insertion, bulk deletion and index parsing/lookup (similar to OBD_IDX_READ and readdir). Those three RPC types involve a bulk transfer.

Although considered out of scope for this project, remote index listing will still have to be implemented for exporting the shard object index (required for daos_shard_list_obj() support). The DAOS object ID packed in the request will be the ID reserved for the shard object index export (see next chapter). Similarly, layout update might be implemented through a bulk insert over the server collective API.

4. Epoch Protocol Support

New RPC types will be added for epoch management. This includes epoch abort (i.e. MDS_EPOCH_ABORT) and commit (i.e. MDS_EPOCH_COMMIT). Epoch slip is implemented via a special handle close RPC. Besides, new OST operations will also be added for commit, abort and space reclaim to be used on top of the server collective API.

v. Shard Management

The VOSD design document provided in the third quarter already describes how shards and DAOS objects will be managed on the storage targets. The purpose of this section is to define the interactions between all Lustre components (i.e. client, MDT and OST) involved in shard management.

1. Shard Creation & Destruction

Like regular OST objects, shard could be pre-created (not implemented in the prototype) and destroyed by the LOD/OSP layers on the MDT. As explained in the previous chapter, a new RPC type will be introduced to create/delete shards in batch. For the time being, no shard pre-allocation is implemented and shard are created synchronously when the shard addition RPC is processed.

A shard creation involves creating the staging dataset as well as registering the shard in the OST shard index. When layout replication is supported in the future, the layout copies will then be refreshed by the MDS to record the existence of the new shard(s).

As for shard destruction, it implies removing not only the staging dataset, but also all the snapshots.

Shard cloning will involve creating a new shard with a staging dataset created from a snapshot of the original container.

2. Capability Management

OST gets the capability list from the MDT each time a container is opened.

The capabilities/handle association is stored in memory on the OST. This latter validates the open handle each time it receives a request for a given shard. Upon OST restart, the MDT has to resend the capability list to the OSTs that rebooted. Similarly, a newly created shard needs to fetch the list of handle and associated capabilities.

Appendix: Updated DAOS API

```
typedef uint64_t daos_off_t;
typedef uint64_t daos_size_t;

/**
 * generic handle, which can refer to any local data structure
(container,
 * object, eq, epoch scope...)
 */
typedef struct {
      uint64_t    cookie;
} daos_handle_t;

typedef struct {
      /** epoch sequence number */
      uint64_t    seq;
} daos_epoch_t;

#define DAOS_EPOCH_HIGHEST   (-1)

/** Event type */
typedef enum {
      DAOS_EV_NONE,
      /** parent event */
      DAOS_EV_COMPOUND,
      DAOS_EV_EQ_DESTROY,
      DAOS_EV_SYS_OPEN,
      DAOS_EV_SYS_CLOSE,
      DAOS_EV_SYS_QUERY,
      DAOS_EV_SYS_QUERY_TGT,
      DAOS_EV_CO_OPEN,
      DAOS_EV_CO_CLOSE,
      DAOS_EV_CO_UNLINK,
      DAOS_EV_CO_SNAPSHOT,
      DAOS_EV_CO_FLUSH,
      DAOS_EV_CO_QUERY,
      DAOS_EV_CO_LS_SHARD,
      DAOS_EV_G2L,
      DAOS_EV_SHARD_ADD,
      DAOS_EV_SHARD_DISABLE,
      DAOS_EV_SHARD_QUERY,
      DAOS_EV_SHARD_FLUSH,
      DAOS_EV_SHARD_LS_OBJ,
      DAOS_EV_OBJ_OPEN,
      DAOS_EV_OBJ_CLOSE,
      DAOS_EV_OBJ_READ,
      DAOS_EV_OBJ_WRITE,
      DAOS_EV_OBJ_PUNCH,
      DAOS_EV_EP_SLIP,
      DAOS_EV_EP_WAIT,
      DAOS_EV_EP_QUERY,
      DAOS_EV_EP_COMMIT,
      DAOS_EV_EP_ABORT,
} daos_ev_type_t;

/**
 * Userspace event structure
 */
struct daos_event {
      /** event type */
        daos_ev_type_t       ev_type;
      /**
       * returned result of asynchronous operation, 0 or error code.
       */
      int              ev_error;
      /** reserved space for DAOS usage */
      struct {
            uint64_t   space[7];
      }                ev_private;
};

/** wait for completion event forever */
#define DAOS_EQ_WAIT        -1
/** always return immediately */
#define DAOS_EQ_NOWAIT      0

typedef enum {
      /** query outstanding completed event */
      DAOS_EVQ_COMPLETED     = (1),
      /** query # inflight event */
      DAOS_EVQ_INFLIGHT= (1 << 1),
      /** query # inflight + completed events in EQ */
      DAOS_EVQ_ALL           = (DAOS_EVQ_COMPLETED |
DAOS_EVQ_INFLIGHT),
} daos_ev_query_t;

typedef enum {
      /** all shards are in consistent status */
      DAOS_CONTAINER_ST_OK,
      /** some shards are stuck to commit */
      DAOS_CONTAINER_ST_STUCK,
      /** some shards are inaccessible */
      DAOS_CONTAINER_ST_FAULTY,
      /** container is corrupt */
      DAOS_CONTAINER_ST_CORRUPT,
} daos_container_status_t;

enum {
      /** target is unknown */
      DAOS_TARGET_ST_UNKNOWN       = -2,
      /** target is disabled */
      DAOS_TARGET_ST_DISABLED            = -1,
      /** 0 - 100 are health levels */
};

typedef enum {
      DAOS_TARGET_TYPE_HDD,
      DAOS_TARGET_TYPE_SSD,
      /** TODO: add more types */
} daos_target_type_t;

struct daos_target_event {
      /** target ID */
      unsigned int           te_target;
      /** target health status, see DAOS_TARGET_ST_* for details */
      int              te_status;
};

/** event for adding/disabling shard */
struct daos_shard_event {
      /** shard ID */
      unsigned int           se_shard;
      /** target ID of this shard */
      unsigned int           se_target;
      /** target health status, see DAOS_TARGET_ST_* for details */
      int              se_status;
      /** epoch of shard operation */
      daos_epoch_t           se_epoch;
};

/**
 * DAOS storage tree structure has four layers: cage, rack, node and
target
 */
typedef enum {
      DAOS_LOC_TYP_UNKNOWN   = 0,
      DAOS_LOC_TYP_CAGE,
      DAOS_LOC_TYP_RACK,
      DAOS_LOC_TYP_NODE,
      DAOS_LOC_TYP_TARGET,
} daos_loc_type_t;

/**
 * target placement information
 */
#define DAOS_LOC_UNKNOWN     -1

/**
 * location ID of cage/rack/node
 */
struct daos_loc_key {
      /** location type of this ID: DAOS_LOC_CAGE/RACK/NODE/TARGET
*/
      daos_loc_type_t        lk_type;
      /** logic ID of cage/rack/node/target */
      unsigned int           lk_id;
};

/**
 * placement information of DAOS storage tree
 */
struct daos_location {
      /** cage number */
      int              lc_cage;
      /** rack number */
      int              lc_rack;
      /** node number */
      int              lc_node;
};

/**
 * performance metrics for target
 */
struct daos_target_perf {
      unsigned int           tp_rdbw; /* MB */
      unsigned int           tp_rdbw_bs;
      unsigned int           tp_wrbw; /* MB */
      unsigned int           tp_wrbw_bs;
      unsigned int           tp_iops;
      unsigned int           tp_iops_bs;
};

struct daos_target_aff {
      /** reserved for target CPU affinity */
      int              ta_cpu_aff;
      /**
       * Network distance, it's ZERO if target is attached on local
node,
       * otherwise it's lustre network hops, e.g, 1 for target on
local
       * lustre network, 2 for target on remote network that can be
reached
       * via one routing hop.
       */
      int              ta_net_dist;
      /** latency from caller to target (micro-second) */
      uint64_t         ta_latency;
};

struct daos_target_space {
      /** capacity of the target */
      daos_size_t      ts_size;
      /** free space of target */
      daos_size_t      ts_free;
	/** number of shards in this target */
	unsigned int		ts_nshard;
};

#define DAOS_TARGET_MAX_FAILOVER	4

struct daos_target_loc {
	/** location offset of current node in \a ti_failovers */
	unsigned int		tl_current;
	/** number of failover nodes of this target */
	unsigned int		tl_nlocs;
	/** location of failover nodes */
	struct daos_location	tl_failovers[DAOS_TARGET_MAX_FAILOVER];
};

/**
 * detail information of a target
 */
struct daos_target_info {
	/** target health status, see DAOS_TARGET_ST_* for details */
	int			ti_status;
	/** storage type */
	daos_target_type_t		ti_type;
	/** target location */
	struct daos_target_loc		ti_loc;
	/** bandwidth information of target */
	struct daos_target_perf			ti_perf;
	struct daos_target_space	ti_space;
	struct daos_target_aff		ti_affinity;
};

/** container open modes */
/** read-only */
#define	DAOS_COO_RO		(1)
/** read-write */
#define DAOS_COO_RW			(1 << 1)
/** create container if it's not existed */
#define DAOS_COO_CREATE				(1 << 2)
/** tell DAOS to not retain request and data for replay */
#define DAOS_COO_OWN_RECOV		(1 << 3)

/**
 * container information
 */
struct daos_container_info {
	/** user-id of owner */
	uid_t			ci_uid;
	/** group-id of owner */
	gid_t			ci_gid;
	/** number of shards */
	unsigned int		ci_nshard;
	/** number of shards are stuck to commit */
	unsigned int		ci_nshard_stuck;
```

```
      /** number of disabled shards */
      unsigned int           ci_nshard_disabled;
      /** TODO: add members */
};

typedef enum {
      /** query all shards */
      DAOS_SHARD_LS_ALL,
      /** query all shards stuck to commit */
      DAOS_SHARD_LS_STUCK,
      /** query all disabled shards */
      DAOS_SHARD_LS_DISABLED,
} daos_shard_list_t;

struct daos_shard_space {
      /** number of non-empty object */
      daos_size_t      sai_nobjs;
      /** space used */
      daos_size_t      sai_used;
} ;

struct daos_shard_info {
      unsigned int           sai_target;
      /** storage type */
      daos_target_type_t     sai_type;
      /** target health status, see DAOS_TARGET_ST_* for details */
      int              sai_status;
      /** shard location */
      struct daos_target_loc sai_location;
      /** space infomation of target */
      struct daos_shard_space      sai_space;
      /** TODO: add members */
};

#define DAOS_LIST_END        0xffffffffffffffffULL

/** object ID */
typedef struct {
      /** baseline DAOS API, it's shard ID (20 bits for DAOS
targets) */
      uint64_t   o_id_hi;
      /** baseline DAOS API, it's object ID within shard */
      uint64_t   o_id_lo;
} daos_obj_id_t;

enum {
      /* access mode: either DAOS_OBJ_RO or DAOS_OBJ_RW must be
specified */
      /** read-only mode, no read cache/no read-ahead by default */
      DAOS_OBJ_RO      = (1 << 1),
      /** read & write mode, no read/write caching by default */
      DAOS_OBJ_RW      = (1 << 2),
      /** buffered write, writeback caching is enabled */
      DAOS_OBJ_IO_BUFFERED   = (1 << 3),
      /** random I/O */
      DAOS_OBJ_IO_RAND = (1 << 4),
      /** sequential I/O */
      DAOS_OBJ_IO_SEQ        = (1 << 5),
};

/* Deprecated, keep for compability purpose */
#define DAOS_OBJ_EXCL (DAOS_OBJ_IO_BUFFERED | DAOS_OBJ_RW)

/**
 * DAOS memroy buffer fragment
 */
struct daos_mm_frag {
      void             *mf_addr;
      daos_size_t      mf_nob;
};

/**
 * DAOS memory descriptor, it's an array of daos_iovec_t and it's
source
 * of write or target of read
 */
struct daos_mmd {
      unsigned long          mmd_nfrag;
      struct daos_mm_frag    mmd_frag[0];
};

/**
 * IO fragment of a DAOS object
 */
struct daos_io_frag {
      daos_off_t       if_offset;
      daos_size_t      if_nob;
};

/**
 * IO desriptor of a DAOS object, it's an array of daos_io_frag_t
and
 * it's target of write or source of read
 */
struct daos_iod {
      unsigned long          iod_nfrag;
      struct daos_io_frag    iod_frag[0];
};

struct daos_epoch_info {
      /**
       * Highest Committed Epoch (HCE).
       * Highest consistently committed epoch of all shards (all
shards
       * in container have successfullly committed this epoch), this
       * is the latest readable epoch of a container
       */
      daos_epoch_t           epi_hce;
      /** Oldest readable epoch */
      daos_epoch_t           epi_hce_oldest;
      /** Slipped epoch of current open */
      daos_epoch_t           epi_hce_slipped;
      /**
       * Highest Shard Epoch (HSE)
       * Highest committed epoch of all shards in a container, when
       * container is in consistent status (all shards successfully
       * committed the last epoch) HSE should equal to HCE,
otherwise
       * it could be different if container only partially committed
       * the last epoch (some shards failed to commit), in this
case,
       * HSE should be a value higher than HCE.
       */
      daos_epoch_t           epi_hse;
      /**
       * Layout changed epoch
       * If this structure is used by daos_epoch_query(), the last
layout
       * changed epoch before and including \a epi_hse is returned.
       */
      daos_epoch_t           epi_layout_changed;
};

/*******************************************************************
****
 * Event-Queue (EQ) and Event
 *
 * EQ is a queue that contains events inside.
 * All DAOS APIs are asynchronous, events occur on completion of
DAOS APIs.
 * While calling DAOS API, user should pre-alloate event and pass it
 * into function, function will return immediately but doesn't mean
it
 * has completed, i.e: I/O might still be in-flight, the only way
that
 * user can know completion of operation is getting the event back
by
 * calling daos_eq_poll().
 *
 * NB: if NULL is passed into DAOS API as event, function will be
 *     synchronous.

********************************************************************
***/

/**
 * create an Event Queue
 *
 * \param eq [OUT]     returned EQ handle
 *
 * \return       zero on success, negative value if error
 */
int
daos_eq_create(daos_handle_t *eqh);

/**
 * Destroy an Event Queue, it wait -EBUSY if EQ is not empty.
 *
 * \param eqh [IN]     EQ to finalize
 * \param ev [IN]pointer to completion event
 *
 * \return       zero on success, EBUSY if there's any inflight
event
 */
int
daos_eq_destroy(daos_handle_t eqh);

/**
 * Retrieve completion events from an EQ
 *
 * \param eqh [IN]     EQ handle
 * \param wait_inf [IN]      wait only if there's inflight event
 * \param timeout [IN] how long is caller going to wait (micro-
second)
 *               if \a timeout > 0,
 *               it can also be DAOS_EQ_NOWAIT, DAOS_EQ_WAIT
 * \param eventn [IN]  size of \a events array, returned number of
events
 *               should always be less than or equal to \a eventn
 * \param events [OUT] pointer to returned events array
 *
 * \return       >= 0  returned number of events
 *               < 0   negative value if error
 */
int
daos_eq_poll(daos_handle_t eqh, int wait_inf,
           int64_t timeout, int eventn, struct daos_event **events);

/**
 * Query how many outstanding events in EQ, if \a events is not
NULL,
 * these events will be stored into it.
 * Events returned by query are still owned by DAOS, it's not
allowed to
 * finalize or free events returned by this function, but it's
allowed
 * to call daos_event_abort() to abort inflight operation.
 * Also, status of returned event could be still in changing, for
example,
 * returned "inflight" event can be turned to "completed" before
acessing.
```

```
 * It's user's responsibility to guarantee that returned events
would be
 * freed by polling process.
 *
 * \param eqh [IN]     EQ handle
 * \param mode [IN]    query mode
 * \param eventn [IN]  size of \a events array
 * \param events [OUT] pointer to returned events array
 * \return       >= 0  returned number of events
 *                < 0  negative value if error
 */
int
daos_eq_query(daos_handle_t eqh, daos_ev_query_t query,
            unsigned int eventn, struct daos_event **events);

/**
 * Initialize a new event for \a eq
 *
 * \param ev [IN]event to initialize
 * \param eqh [IN]     where the event to be queued on, it's ignored
if
 *               \a parent is specified
 * \param parent [IN]  "parent" event, it can be NULL if no parent
event.
 *               If it's not NULL, caller will never see completion
 *               of this event, instead he will only see completion
 *               of \a parent when all children of \a parent are
 *               completed.
 *
 * \return       zero on success, negative value if error
 */
int
daos_event_init(struct daos_event *ev, daos_handle_t eqh,
           struct daos_event *parent);

/**
 * Finalize an event. If event has been passed into any DAOS API, it
can only
 * be finalized when it's been polled out from EQ, even it's aborted
by
 * calling daos_event_abort().
 * Event will be removed from child-list of parent event if it's
initialized
 * with parent. If \a ev itself is a parent event, then this
function will
 * finalize all child events and \a ev.
 *
 * \param ev [IN]event to finialize
 *
 * \return       zero on success, negative value if error
 */
int
daos_event_fini(struct daos_event *ev);
```

```
/**
 * Get the next child event of \a ev, it will return the first child
event
 * if \a child is NULL.
 *
 * \param parent [IN]  parent event
 * \param child [IN]   current child event.
 *
 * \return       the next child event after \a child, or NULL if
it's
 *               the last one.
 */
struct daos_event *
daos_event_next(struct daos_event *parent, struct daos_event
*child);

/**
 * Try to abort operations associated with this event.
 * If \a ev is a parent event, this call will abort all child
operations.
 *
 * \param ev [IN]event (operation) to abort
 *
 * \return       zero on success, negative value if error
 */
int
daos_event_abort(struct daos_event *ev);

/*****************************************************************
****
 * Query DAOS storage layout and target information

******************************************************************
***/

/**
 * Open system container which contains storage layout and detail
 * information of each target.
 * This system container is invisible to namespace, and it can't be
 * modified by DAOS API.
 * daos_sys_open will get reference of highest committed epoch of
the
 * system container, which means all queries will only get
information
 * within this epoch.
 *
 * \param daos_path [IN]     path to mount of filesystem
 * \param handle [OUT]       returned handle of context
 * \param ev [IN]      pointer to completion event
 *
 * \return             zero on success, negative value if error
 */
```

```
int
daos_sys_open(const char *daos_path,
            daos_handle_t *handle, struct daos_event *ev);

/**
 * Close system container and release refcount of the epoch
 *
 * \param handle [IN]       handle of DAOS context
 * \param ev [IN]      pointer to completion event
 *
 * \return             zero on success, negative value if error
 */
int
daos_sys_close(daos_handle_t handle, struct daos_event *ev);

/**
 * Listen on change of target status
 *
 * \param handle [IN]       handle of DAOS system container
 * \param tevn [IN/OUT]     number of target events
 * \param tevs [OUT]        array of target events
 * \param ev [IN]      pointer to completion event
 *
 * \return             zero on success, negative value if error
 */
int
daos_sys_listen(daos_handle_t handle, unsigned int *tevn,
            struct daos_target_event *tevs, struct daos_event *ev);

/**
 * Query storage tree topology of DAOS
 *
 * \param handle [IN]       handle of DAOS sys-container
 * \param loc [IN]          input location of cage/rack/node/target
 * \param depth [IN]        subtree depth of this query, if depth
is 0
 *                     then querying the whole subtree.
 * \param lkn [IN/OUT]      if \a lks is NULL, number of subtree
traversal
 *                     footprints is returned (traversal depth is
 *                     limited by \a depth), otherwise it's input
 *                     parameter which is array size of \a lks
 * \param lks [OUT]         array to store footprint of preorder
subtree
 *                     traversal
 *                     a) loc::lc_cage is DAOS_LOC_UNKNOWN, returned
 *                        footprints for different depth:
 *                        0/4+: whole tree traversal
 *                        1: cages traversal
 *                        2: cages and racks traversal
 *                        3: cages, racks and node traversal
 *
 *                     b) loc::lc_cage is specified, loc::lc_rack is
 *                          DAOS_LOC_UNKNOWN, returned footprints for
 *                          different depth:
 *                          0/3+: subtree traversal of this cage
 *                          1: racks traversal under this cage
 *                          2: racks and node traversal
 *
 *                       c) loc::lc_cage and loc::lc_rack are
specified,
 *                          loc::lc_node is DAOS_LOC_UNKNOWN, returned
 *                          footprints for different depth:
 *                          0/2+: subtree traversal of this rack
 *                          1: nodes traversal under this rack
 *
 *                       c) loc::lc_cage, loc::lc_rack loc::node are
 *                          specified, returned footprints:
 *                          any: targets traversal of this node
 *
 * \param ev [IN]pointer to completion event
 *
 * \return       zero on success, negative value if error
 */
int
daos_sys_query(daos_handle_t handle, struct daos_location *loc,
             unsigned int depth, unsigned int *lkn,
             struct daos_loc_key *lks, struct daos_event *ev);

/*
 * Query detail information of a storage target
 *
 * \param handle [IN]        handle of DAOS sys-container
 * \param target [IN]        location of a target
 * \param info [OUT]         detail information of the target
 * \param failover [OUT]     it can be NULL, if it's not NULL,
failover
 *                       nodes location of given target is returned
 * \param ev [IN]        pointer to completion event
 *
 * \return               zero on success, negative value if error
 */
int
daos_sys_query_target(daos_handle_t handle, unsigned int target,
                  struct daos_target_info *info, struct daos_event
*ev);

/*******************************************************************
****
 * Container data structures and functions
 *
 * DAOS container is a special file which exists in POSIX namespace
 * But user can only change/access content of a container via DAOS
APIs.
 * A container can contain any number of shards (shard is kind of
virtual
```

```
 * storage target), and can contain infinite number of DAOS objects.

*******************************************************************
***/

/**
 * Open a DAOS container
 *
 * Collective open & close:
 * ------------------------
 * If there're thousands or more processes want to open a same
container
 * for read/write, server might suffer from open storm, also if all
these
 * processes want to close container at the same time after they
have
 * done their job, server will suffer from close storm as well.
That's
 * the reason DAOS needs to support collective open/close.
 *
 * Collective open means one process can open a container for all
his
 * sibling processes, this process only needs to send one request to
 * server and tell server it's a collective open, after server
confirmed
 * this open, he can broadcast open representation to all his
siblings,
 * all siblings can then access the container w/o sending open
request
 * to server.
 *
 * After all sibling processes done their job, they need to call
close to
 * release local handle, only the close called by opener will do the
real
 * close.
 *
 * \param path [IN]    path to container in POSIX namespace
 * \param mode [IN]    open mode, see above comment
 * \param nprocess [IN]      it's a collective open if nprocess > 1
 *               it's the number of processes will share this open
 * \param status [OUT] status of container, see \a
daos_container_status_t
 *               for details.
 * \param coh [OUT]    returned container handle
 * \param event [IN]   pointer to completion event
 *
 * \return       zero on success, negative value if error
 */
int
daos_container_open(const char *path, unsigned int mode, unsigned
int nprocess,
               daos_container_status_t *status, daos_handle_t *coh,
                struct daos_event *event);

/**
 * close a DAOS container and release open handle.
 * This is real regular close if \a coh is not a handle from
collective
 * open. If \a coh a collectively opened handle, and it's called by
opener,
 * then it will do the real close for container, otherwise it only
release
 * local open handle.
 * If user failed to commit to a container then closed it, all
writes
 * in epochs higher than HSE will be discarded.
 *
 * Highest Shard Epoch(HSE) is the highest committed epoch of all
shards.
 * See \a daos_epoch_info for details.
 *
 * \param coh [IN]     container handle
 * \param event  [IN]  pointer to completion event
 *
 * \return        zero on success, negative value if error
 */
int
daos_container_close(daos_handle_t coh, struct daos_event *event);

/**
 * destroy a DAOS container and all shards
 *
 * \param path [IN]    POSIX name path to container
 * \param event  [IN]  pointer to completion event
 *
 * \return        zero on success, negative value if error
 */
int
daos_container_unlink(const char *path, struct daos_event *event);

/**
 * create snapshot for a container based on its last durable epoch
 *
 * \param coh [IN]     handle of DAOS container to be snapshot
 * \param epoch [IN]   epoch of original container to create
 *                the snapshot
 * \param snapshot [IN]     path of snapshot
 * \param event [IN]   pointer to completion event
 *
 * \return        zero on success, negative value if error
 */
int
daos_container_snapshot(daos_handle_t coh, daos_epoch_t epoch,
                 const char *snapshot, struct daos_event *event);
```

```
/**
 * return shards information, uid/gid and other metadata of
container
 *
 * \param coh [IN]     handle of DAOS container
 * \param epoch [IN/OUT]
 *                If it is a valid committed epoch, then it is an
input
 *                parameter and it is the epoch to query.
 *
 *                If it is DAOS_EPOCH_HIGHEST or an invalid epoch, it
is
 *                an parameter for output, HSE will be stored in it
on
 *                completion. See \a daos_epoch_info for details.
 *
 * \param info [OUT]   returned container information
 * \param event [IN]   pointer to completion event
 *
 * \return        zero on success, negative value if error
 */
int
daos_container_query(daos_handle_t coh, daos_epoch_t *epoch,
                struct daos_container_info *info,
                struct daos_event *event);

/**
 * Query shards of a container, it can return all shards, all
 * disabled shards, or all uncommitted (stuck to commit) shards
 * in specified epoch.
 * If \a epoch is an invalid epoch, then shards of HSE will be
returned.
 *
 * \param coh [IN]     handle of DAOS container
 * \param epoch [IN]   epoch to query
 * \param opc [IN]     operation code of list (all/stuck/disabled)
 *                see \a daos_shard_list_t for details
 * \param anchor [IN/OUT]
 *                anchor for the next shard ID
 *                it will be set to -1 if no more shard can be listed
 * \param shardn [IN]  it is array size of \a shards.
 * \param shards [OUT] buffer to store returned shards, it's filled
with -1
 *                if number of returned shards is less than \a
shardn.
 * \param event [IN]   pointer to completion event
 *
 * \return        zero on success, negative value if error
 */
int
daos_container_list_shard(daos_handle_t coh, daos_epoch_t epoch,
                   daos_shard_list_t opc, daos_off_t *anchor,
                   unsigned int shardn, unsigned int *shards,
                   struct daos_event *event);

/**
 * Listen on layout change of container
 *
 * \param handle [IN]       handle of DAOS container
 * \param sevn [IN/OUT]           number of shard events
 * \param sevs [OUT]        array of shard events
 * \param ev [IN]     pointer to completion event
 *
 * \return             zero on success, negative value if error
 */
int
daos_container_listen(daos_handle_t coh, unsigned int *sevn,
                 struct daos_shard_event *sevs, struct daos_event
*event);

/**
 * Flush all (cached) changes up to give epoch to a container.
 * This needs to be called by all processes that share a collective
 * open before commit.
 *
 * \param coh [IN]          container handle
 * \param epoch [IN]        epoch of this container
 * \param shard [IN]        shard ID
 * \param event [IN]        completion event
 *
 * \return             zero on success, negative value if error
 */
int
daos_container_flush(daos_handle_t coh, daos_epoch_t epoch,
                struct daos_event *event);

/*********************************************************************
****
 * collective operation APIs

*********************************************************************
***/

/**
 * Convert a local container handle to global representation data
which
 * can be shared with peer processes. This function can only be
called
 * by the process did collective open.
 *
 * \param handle [IN]  container handle
 * \param global[OUT]  buffer to store container information
 * \param size[IN/OUT] buffer size to store glolal representation
data,
 *                if \a global is NULL, required buffer size is
 *                returned, otherwise it's the size of \a global.
 *
 * \return       zero on success, negative value if error
 */
int
daos_local2global(daos_handle_t handle,
             void *global, unsigned int *size);

/**
 * Create/Refresh a local container handle for global representation
data.
 * see details in \a daos_container_open and \a daos_local2global
 * If a valid container handle is passed, the container information
like
 * the shard list will be refreshed from the global handle.
 * Otherwise, a new container handle will be allocated.
 *
 * \param handle [IN/OUT] returned handle
 * \param global[IN]   global (shared) representation of a
collectively
 *               opened container
 * \param size[IN]     bytes number of \a global
 *
 * \return       zero on success, negative value if error
 *
 * Example:
 * process-A:
 *     daos_container_open(..., DAOS_CMODE_RD, 2, &coh, ...);
 *     daos_local2global(coh, NULL, &size);
 *     gdata = malloc(size);
 *     daos_local2global(coh, gdata, &size);
 *     <send gdata to process-B>
 *     <start to access container via coh>
 *
 * process-B:
 *     <receive gdata from process-A>
 *     daos_global2local(gdata, size, &coh, ...);
 *     <start to access container via coh>
 */
int
daos_global2local(void *global, unsigned int size,
             daos_handle_t *handle, struct daos_event *ev);

/*******************************************************************
****
 * Shard API
 *
 * Container is application namespace, and shard is virtual storage
target
 * of container, user can add any number of shard into a container,
or
 * disable shard for a container so shard is invisible to that
container.
```

```
 * user need to specify a shard while creating object in a
container.

*****************************************************************
***/

/**
 * Add new shards to a container.
 *
 * \param coh [IN]          container owns this shard
 * \param epoch [IN]        writable epoch of this container
 * \param targetn [IN]      array size of \a targets and \a shards
 * \param targets [IN]      array of targets
 * \param shards [IN/OUT]   specified shard ID array
 *                  Indexes are reset to -1 if the shard
 *                  creation failed.
 * \param event [IN]        completion event
 *
 * \return              zero if all shards have been created.
 *                  negative value if at least one shard failed
 *                  to be added (others might have been
successfully
 *                  created). Shard ID array should be checked.
 */
int
daos_shard_add(daos_handle_t coh, daos_epoch_t epoch,
             unsigned int targetn, unsigned int *targets,
             int *shards, struct daos_event *event);

/**
 * disable a shard for a container
 *
 * \param coh [IN]          container owns this shard
 * \param epoch [IN]        writable epoch of this container
 * \param shardn [IN]       array size of \a shards
 * \param shards [IN/OUT]   shard IDs to disable
 *                  set back to -1 if the shard failed to be
 *                  deactivated
 * \param event [IN]        completion event
 *
 * \return              zero if all shards have been deactivated,
 *                  negative value if at least one shard failed
to
 *                  be disable.
 */
int
daos_shard_disable(daos_handle_t coh, daos_epoch_t epoch, unsigned
int shardn,
               int *shards, struct daos_event *event);

/**
 * query a shard, i.e: placement information, number of objects etc.
 *
 * \param coh [IN]           container handle
 * \param epoch [IN/OUT]     If it is a valid committed epoch, then
it's an
 *                     input parameter and it's the epoch to query.
 *                     If it is DAOS_EPOCH_HIGHEST or an invalid
 *                     epoch, it is an parameter for output, the
last
 *                     committed epoch of this shard will be stored
 *                     in it on completion.
 * \param shard [IN]         shard ID
 * \param sinfo [OUT]        returned shard information
 * \param event [IN]         completion event
 *
 * \return             zero on success, negative value if error
 */
int
daos_shard_query(daos_handle_t coh, daos_epoch_t *epoch, unsigned
int shard,
            struct daos_shard_info *info, struct daos_event *event);

/**
 * Flush all (cached) changes up to give epoch to a shard
 *
 * \param coh [IN]           container handle
 * \param epoch [IN]         epoch of this container
 * \param shard [IN]         shard ID
 * \param event [IN]         completion event
 *
 * \return             zero on success, negative value if error
 */
int
daos_shard_flush(daos_handle_t coh, daos_epoch_t epoch,
            unsigned int shard, struct daos_event *event);

/**
 * enumerate non-empty object IDs in a shard
 *
 * \param coh [IN]           container handle
 * \param epoch [IN]         epoch of this container
 * \param shard [IN]         shard ID
 * \param anchor [IN/OUT]    anchor for the next object ID
 *                     it will be set to -1 if no more object
 *                     can be listed
 * \param oidn [IN]          size of \a objids array
 * \param oids [OUT]         returned object IDs. It's filled with -
1
 *                     if number of returned object IDs is less than
 *                     \a oidn.
 * \param event [IN]         completion event
 *
 * \return             zero on success, negative value if error
 */
int
```

```
daos_shard_list_obj(daos_handle_t coh, daos_epoch_t epoch,
                unsigned int shard, daos_off_t *anchor,
                daos_size_t oidn, daos_obj_id_t *oids,
                struct daos_event *event);

/****************************************************************
 * Object API
 ****************************************************************/
/**
 * open a DAOS object for I/O
 * DAOS always assume all objects are existed (filesystem actually
 * needs to CROW, CReate On Write), which means user doesn't need to
 * explictly create/destroy object, also, size of object is infinite
 * large, read an empty object will just get all-zero buffer.
 *
 * \param coh [IN]     container handle
 * \param oid [IN]     object to open
 * \param mode [IN]    open mode: DAOS_OMODE_RO/WR/RW
 * \param oh [OUT]     returned object handle
 * \param event [IN]   pointer to completion event
 *
 * \return       zero on success, negative value if error
 */
int
daos_object_open(daos_handle_t coh,
             daos_obj_id_t oid, unsigned int mode,
             daos_handle_t *oh, struct daos_event *event);

/**
 * close a DAOS object for I/O, object handle is invalid after this.
 *
 * \param oh [IN]open handle of object
 *
 * \return       zero on success, negative value if error
 */
int
daos_object_close(daos_handle_t oh, struct daos_event *event);

/**
 * read data from DAOS object, read from non-existed data will
 * just return zeros.
 *
 * \param oh [IN]object handle
 * \param epoch [IN]   epoch to read
 * \param mmd [IN]     memory buffers for read, it's an arry of
buffer + size
 * \param iod [IN]     source of DAOS object read, it's an array of
 *             offset + size
 * \param event [IN]   completion event
 *
 * \return       zero on success, negative value if error
 */
int
```

```
daos_object_read(daos_handle_t oh, daos_epoch_t epoch,
             struct daos_mmd *mmd, struct daos_iod *iod,
             struct daos_event *event);

/**
 * write data in \a mmd into DAOS object
 * User should always give an epoch value for write, epoch can be
 * any value larger than the HCE, write to epoch number smaller than
HCE
 * will get error.
 *
 * \param oh [IN]object handle
 * \param epoch [IN]   epoch to write
 * \param mmd [IN]     memory buffers for write, it's an array of
buffer + size
 * \param iod [IN]     destination of DAOS object write, it's an
array of
 *               offset + size
 * \param event [IN]   completion event
 *
 * \return       zero on success, negative value if error
 */
int
daos_object_write(daos_handle_t oh, daos_epoch_t epoch,
              struct daos_mmd *mmd, struct daos_iod *iod,
              struct daos_event *event);

/**
 * discard data between \a begin and \a end of an object, all data
will
 * be discarded if begin is 0 and end is -1.
 *
 * This will remove backend FS inode and space if punch it to zero
 *
 * \param coh [IN]     container handle
 * \param epoch [IN]   writable epoch of this container
 * \param oid [IN]     object ID
 * \param begin [IN]   start offset, 0 means begin of the object
 * \param end [IN]     end offset, -1 means end of the object
 * \param event [IN]   completion event
 *
 * \return       zero on success, negative value if error
 */
int
daos_object_punch(daos_handle_t coh, daos_epoch_t epoch,
              daos_obj_id_t oid, daos_off_t begin, daos_off_t end,
              struct daos_event *event);

/**
 * Move the reading cursor to the specified epoch, which means all
epochs
 * earlier than specified epoch will be recycled unless there is
another
```

```
 * reader opened this container.
 * If \a epoch is DAOS_EPOCH_HIGHEST or higher than HCE, then slip
to HCE.
 *
 * \param coh [IN]     container handle
 * \param epoch [IN]   epoch to slip to.
 * \param ev [IN]pointer to completion event
 *
 * \return       zero on success, negative value if error.
 */
int
daos_epoch_slip(daos_handle_t coh, daos_epoch_t epoch, struct
daos_event *ev);

/**
 * Wait for a given epoch to be committed, if the required epoch is
already
 * committed, it will return right away, if the required epoch is
not
 * committed yet, then user will be notified by completion event
when the
 * specified or later epoch has been successfully committed.
 * If \a epoch_layout is not NULL, it will also return the last
layout
 * changed epoch between slipped epoch and the returned epoch.
 *
 * \param coh [IN]     container handle
 * \param epoch [IN/OUT]
 *               epoch to wait on, if it is DAOS_EPOCH_HIGHEST or
 *               any future epoch, then the next committed will be
 *               returned on completion.
 * \param epoch_layout [OUT] returned last epoch when layout was
 *                           modified
 * \param ev [IN]pointer to completion event
 *
 * \return       zero on success, negative value if error.
 */
int
daos_epoch_wait(daos_handle_t coh, daos_epoch_t *epoch,
           daos_epoch_t *epoch_layout, struct daos_event *ev);

/**
 * It is the first epoch function should be called after container
open,
 * it returns HCE and other epoch informations about this container,
 * see daos_epoch_info for details.
 *
 * \param coh [IN]     container handle
 * \param info [OUT]   returned epoch information
 * \param ev [IN]pointer to completion event
 *
 * \return       zero on success, negative value if error.
 */
```

```
int
daos_epoch_query(daos_handle_t coh, struct daos_epoch_info *info,
            struct daos_event *ev);

/**
 * Signals that all writes associated with this container up to and
 * including \a epoch have completed. Completes when this epoch
becomes
 * durable.
 * If commit is failed, it is just like the commit has not happened
yet, HCE
 * remains it was but HSE is updated to the committed epoch.
 * User can either re-commit HSE later, or find out failed shards
and disable
 * them, and commit to HSE+1.
 *
 * \param coh [IN]           container handle
 * \param epoch[OUT]         epoch to complete
 * \param sync[IN]           DAOS will take a snapshot for this
commit
 *                     (no aggregation)
 * \param shard_failed [OUT] failed shared ID when there is error on
commit
 * \param ev [IN]      pointer to completion event
 */
int
daos_epoch_commit(daos_handle_t coh, daos_epoch_t epoch, bool sync,
             unsigned int *shard_failed, struct daos_event *ev);

/**
 * Abort uncommitted epoch, all writes/punches in \a epoch or later
epochs
 * will be abandoned.
 *
 * If \a epoch has already been committed, this function will fail
and return
 * -EPERM.
 * If user submitted changes to aborted epochs after completion of
 * daos_epoch_abort, all changes will be preserved by DAOS as
normal.
 * If there were data left in client cache while calling abort, or
user
 * submitted changes before completion of abort, these data/changes
may or may
 * not be preserved by DAOS, it is unpredictable.
 *
 * \param coh [IN]           container handle
 * \param epoch [IN]         epoch to abort
 * \param shard_failed [OUT] failed shared ID when there is error on
abort
 * \param ev [IN]      pointer to completion event
 */
int
```

```
daos_epoch_abort(daos_handle_t coh, daos_epoch_t epoch,
                 unsigned int *shard_failed, struct daos_event *ev);
```