Date: December 13, 2012	# DAOS API and DAOS POSIX DESIGN DOCUMENT 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

I. Table of Contents

Revision History

Date	Revision	Author
2012-12-12	1.0 Draft for Review	Liang Zhen, Intel Corporation

II. Introduction

DAOS/Lustre depends on development of multiple components still under discussion and in SA phase, at the meanwhile, there are multiple layers depend on implementation of DAOS API. To parallelize developing of whole stack, this project will build DAOS API on top of POSIX, which needs much less efforts and relatively short developing cycle. It should provide sufficient functionality to act as an interim test target until a full implementation based on Lustre is available.

III. Requirements

DAOS API definition

- Provide definition and description of DAOS APIs.

A reduced implementation of DAOS API based on POSIX

- Userspace implementation of EQ and event.
- Implement container, shard and object APIs, which should be all asynchronous.
- This implementation can't support collective operations (open/close of container), although it can simulate collective APIs.
- This implementation can't support transaction.
- This implementation can't guarantee to support consistent read of container because without transaction.

An utility to manage DAOS/POSIX directory tree as simulation of storage tree

- DAOS/POSIX can run over any POSIX filesystem (local or distributed), it will use directory tree in POSIX namespace to simulate DAOS storage tree and support APIs accessing storage tree.
- This utility (daosx_ctl) can generate/manage the POSIX directory tree which has similar tree topology as DAOS storage tree.
- Directory tree (simulation of DAOS storage tree) should have 4 layers: site, rack, node and target. Please check "Logic specification" for details.
- If it's running in Lustre namespace, it should be able to specify MDT for "node" directory, and specify OST for "target" directory (all objects under target should live in the same OST)

IV. Definitions

Event Queue (EQ) and event

- A queue that contains events inside.
- Events occur in any asynchronous DAOS API.
- Most DAOS APIs are asynchronous (except API to create EQ, initialize event…).

- Event Queue (EQ) and events are used for tracking completion event of DAOS functions.
    - DAOS function should return immediately but doesn't mean it has completed, the only way to know completion of operation is get completion event (by poll)

DAOS container

- DAOS container is a special file which exists in POSIX namespace
- Container has special file type, user can only change/access content of a container via DAOS APIs.
- Container is application namespace.
- A container can contain any number of shards (shard is kind of virtual storage target), a shard can contain infinite number of DAOS objects.
- DAOS/POSIX container will be implemented by directory, all its contents are visible to POSIX namespace.

DAOS Container Shard

- Shard is virtual storage target of container.
- User can add any number of shard into a container, or disable shard for a container so all operations to it fail immediately.
- User needs to specify a shard while create object in a container.
- DAOS/POSIX shard is simulated by directory in this project.

DAOS object

- Object is just an array of bytes.
- Object can't be sharded.
- Object is simulated by POSIX file in this project.

Epochs

- Epochs are transaction identifiers and are passed in all DAOS I/O operation.
- We are not going to support transaction in this project.
- Epoch numbers will be maintained.

V. Changes from Solution Architecture

N/A.

VI. Functional specification

Please refer to DAOS API in section XIII.

VII. Use Cases

Create DAOS/POSIX storage tree via daosx_ctl

DAOS/POSIX storage tree will be implemented by a directory tree, daosx_ctl is the utility to manipulate this directory tree

- Add a "site" directory
  daosx_ctl add_site SITE, SITE is just a number
- Add a "rack" to specified site
  daosx_ctl add_rack SITE.RACK
- SITE and RACK are just numbers, SITE should have already been created
- Add a "node" to specified rack
  daosx_ctl add_node [-m MDT] SITE.RACK.NODE
- SITE, RACK and NODE are just numbers, SITE and RACK should have already been created, user can specify MDT by -m
- Add a "target" to specified target
  daosx_ctl add_target [-o OST] SITE.RACK.NODE.TARGET
- SITE, RACK, NODE and target are just numbers, SITE, RACK and NODE should have already been created, user can specify OST by -o

Control target status via daosx_ctl

- daosx_ctl set_target SITE.RACK.NODE.TARGET -h VALUE
  VALUE can be any number between -2 and 100, -2 means "unknown", -1 means "disabled", "0, 1, ..., 100" are health level.
- daosx_ctl might allow user to set other attribute of target, please check define of "daos_target_info_t" in DAOS APIs.

VIII. Logic specification

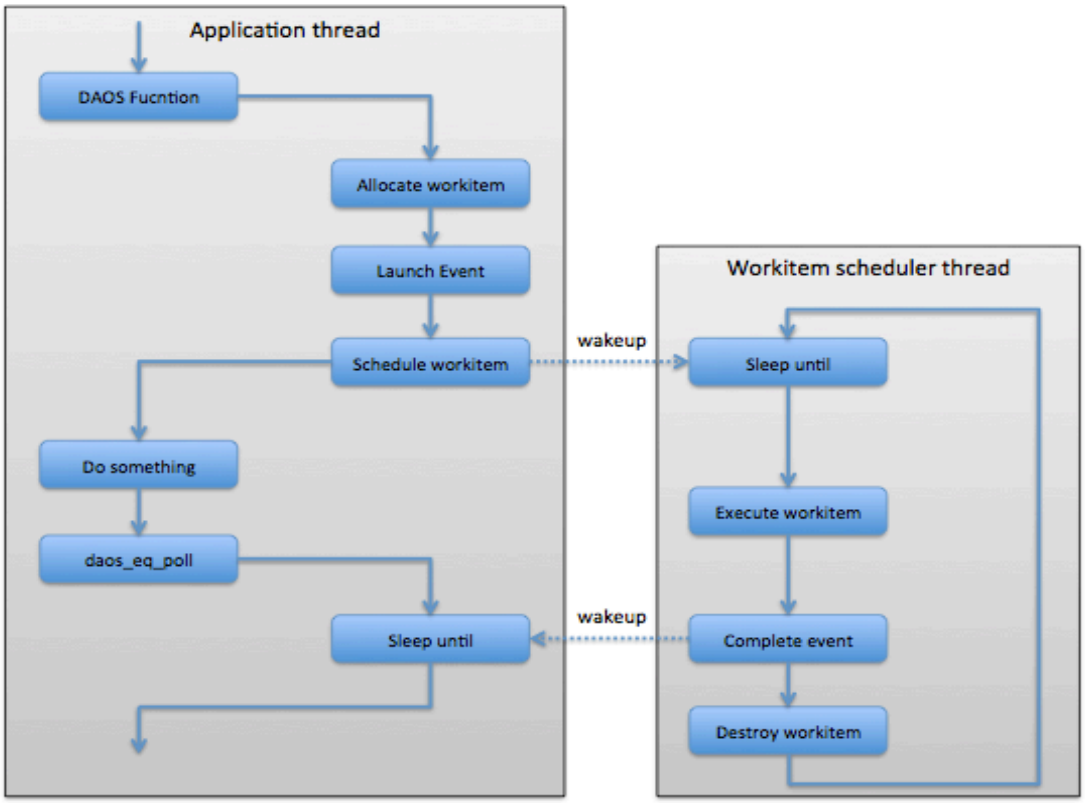
DAOS handle Table

- DAOS will never directly refer to memory address of anything, all objects (EQ, DAOS container, DAOS object, Epoch) are referred by DAOS handle
- Implement a common library to manage all DAOS handles.

Asynchronous operations

- DAOS/POSIX needs a userspace scheduler which contains a thread-pool
- Asynchronous operation should be executed under thread context of scheduler
- Asynchronous DAOS APIs need to create a running unit (workitem) which contains parameters and customized function, then wakeup a schedule thread to execute the workitem, and return immediately.
  - Creation/destroy of workitem should be transparent to API user
  - daos_event_t needs to keep reference to workitem
  - cfs_workitem can be reused in this project, cfs_workitem is a mini-scheduler library in Lustre/libcfs
  - workitem can be aborted before it's been schedule, it can be used to implement daos_event_abort()
- Workitem scheduler needs to enqueue completion event and wakeup polling thread on completion

- If it's a synchronous call (event is NULL), all POSIX operations can be executed under thread context of caller.
- We might need two extra APIs to create/destroy thread-pool for scheduler: daos_posix_init/finalize.



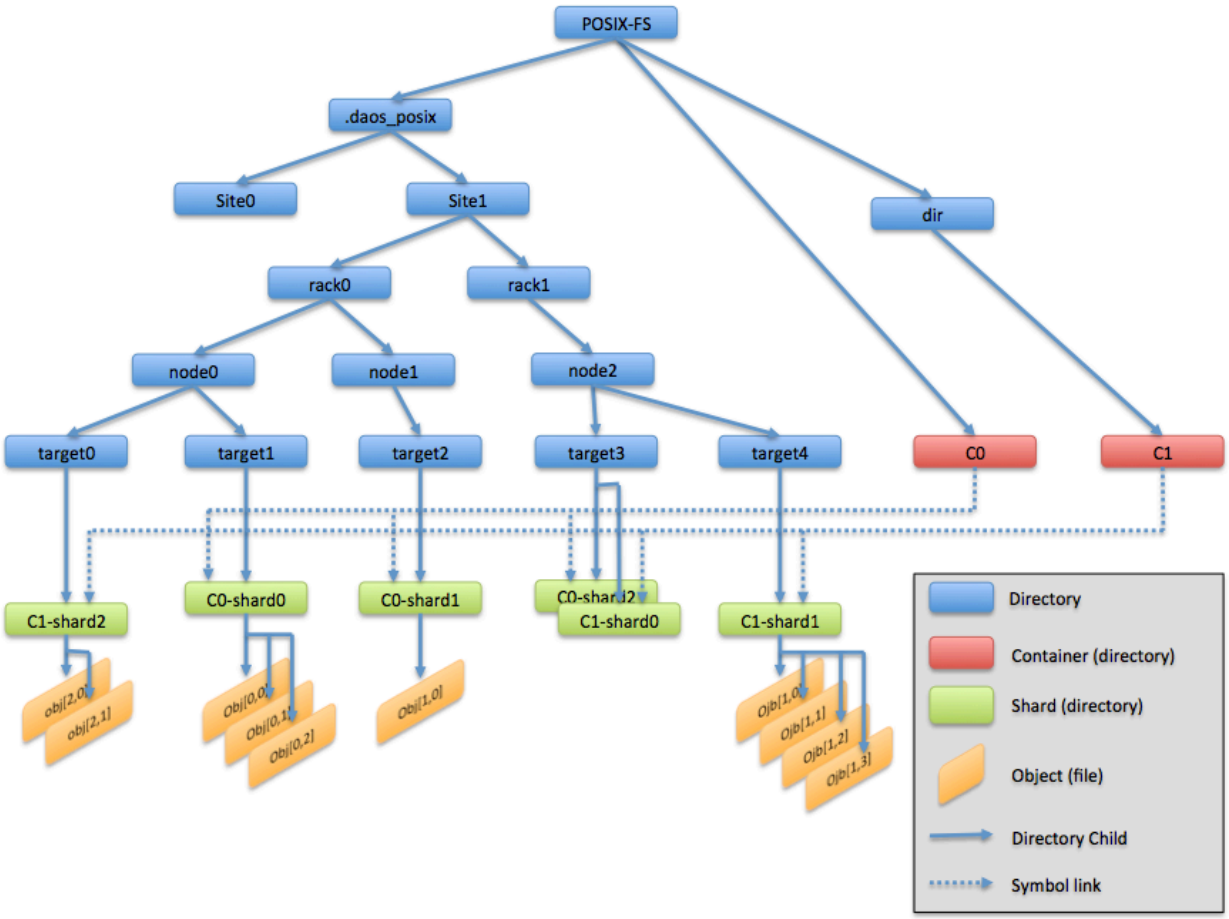
Event Queue (EQ) and Event

- Event queue should contain:
  - inflight event list
  - completion event list
  - pthread_cond_t where polling thread can wait
  - pthread_mutex_t which can serliaze operations
- At the entry of asynchronous function, event will be launched as inflight event, and put on "inflight list"
- Event will be moved from "inflight list" to "completion list" by workitem on completion of operation.

Simulate storage tree by directory

- daosx_ctl can generate directory ".daos_posix" under specified directory
  - If it's running in Lustre namespace, it might be able to automatically probe and build directory tree based on Lustre storage system, but it's not a guaranteed feature.
- This directory tree should have same topology as storage system: site/rack/node/target
- daos_sys_query() can return tree traversal footprint of .daos_posix

- "node" directories can be hashed into different MDTs if DAOS/POSIX is built in Lustre DNE namespace (Lustre 2.4)
- There is a "status" file under each "target" directory, it contains status information of each target
    - It should at least include "OST ID" if it's in Lustre namespace
    - It can also have any status information of future DAOS target, these are just fake information, and can be changed by daosx_ctl or text editor.



Directory based Container and container shard

- DAOS/POSIX container is a directory which can be put at anywhere in the target filesystem.
- DAOS/POSIX container shard is directory as well, it resides in FS_ROOT/.daos_posix/site/rack/node/target/contamer_name_shard#
    - Shard might contain a status file to describe enable/disable status etc.
- Container contains symbolic links to all shards of this container
    - Use symbolic link in POSIX namespace to record relationship between container and shard
- Can't support collective operations for DAOS/POSIX container:
    - local2global really just pack container name and open mode

    - global2local is a real open call
- Can't guarantee to provide container snapshot
    - It will be implemented only if there's time.

File based Object

- DAOS/POSIX Object resides in shard (directory).
    - If it's running in namespace of Lustre, all objects (files) under a shard(directory) should live in a same OST.
    - If it's running in namespace of Lustre, stripecount of file should be one.
- Object (file) is created on the first open for write, this is different with DAOS/Lustre which is CROW (CReat On Write).
- Object (file) is removed on punch
- Use direct I/O for shared writes
- fsync to implement object flush, sync to implement shard/container flush
- Scatter/gather I/O can be simulated by multiple reads/writes in context of workitem.
- DAOS/POSIX objects will pollute POSIX namespace because they are actual files, which means there will be heavy metadata workload while change objects
    - As mentioned in section "simulate storage tree by directory", "node" can be distributed to different MDSs/MDTs if it's Lustre 2.4.
    - By this way, metadata workload of DAOS/POSIX object operations can be distributed to different MDS.

Epochs

- Can't simulate transaction with POSIX
- DAOS/POSIX should maintain epoch numbers for container:
    - Record HCE (Highest Consistent Epoch) of container
    - Don't allow writer to write with epoch equal or lower than HCE
    - Track reference on epoch in-write, epoch can only be ended if all writers have ended that epoch.
    - daso_epoch_slip should be blocked if epoch is still open for write
    - Please refer (Solution Architecture for DAOS) for details
    - Use flock to synchronize epoch changes from distributed processes.
- Can't guarantee to keep consistent version of container.
    - Reader might see the latest change even reading HCE
    - Here is proposal to keep consistent version (this is not a guaranteed feature, it will be implemented only if there's time):
        - Object file only presents object at current HCE
        - Writes for later epochs are logged in each shard and each client has it's own log file for each epoch [object, offset, len, date]
        - When someone ends the epoch, ending request has to wait until there are no readers of the HCE or all readers are slipping, then replay (i.e. copy the data in the logs into the objects) the logs in each shard in epoch order up to the ended epoch
        - Record the new HCE and allow epoch end to complete
        - Allow any in-progress slip to complete

- This might require a dedicated daemon process to wait, replay logs and commit, another daemon process to wait and slip
- Because daemon process needs to rewrite all data to object (file), it's helpful to have multiple daemons running on multiple nodes to parallel logs replay and commit.

IX. Configurable parameters

- User needs to export environment DAOS_POSIX=PATH before running with DAOS/POSIX, $PATH is path to directory that contains DAOS/POSIX directory tree. For example: "export DAOS_POSIX=/mnt/lustre", or "export DAOS_POSIX=/tmp".

X. API and Protocol Changes

- Need to add daos_posix_init/finalize, they are only for DAOS/POSIX library, and will be removed for DAOS/LUSTRE.

XI. Open issues

N/A.

XII. Risks and Unknowns

- This project can't simulate any kind of transaction, can't rollback to consistent status, so it's almost impossible for layers on top DAOS/POSIX to handle errors and rollback to a clean version of container.
- DAOS/POSIX should be able to run over any version of Lustre, but it will be better to have DNE for scaling tests.
- If there's a chance to implement consistent read, there will be degradation of performance because everything will be written for twice.

XIII. DAOS API

DAOS API draft

```
/*****************************************************************
*
  * public definitions

*****************************************************************
/
 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       ep_seq;
 } daos_epoch_id_t;


 typedef struct {
       /** epoch scope of current I/O request */
       daos_handle_t  ep_scope;
       /** epoch ID of current I/O request */
       daos_epoch_id_t ep_eid;
 } daos_epoch_t;


 /** object ID */
 typedef struct {
       /** baseline DAOS API, it's shard ID */
       uint64_t       o_id_hi;
       /** baseline DAOS API, it's object ID within shard */
       uint64_t       o_id_lo;
 } daos_obj_id_t;



/*****************************************************************
*
  * 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.

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


 /** Event type */
 typedef enum {
        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_G2L,
        DAOS_EV_CO_QUERY,
        DAOS_EV_SHARD_ADD,
        DAOS_EV_SHARD_DISABLE,
        DAOS_EV_SHARD_QUERY,
        DAOS_EV_SHARD_LS_OBJ,
        DAOS_EV_OBJ_START,
        DAOS_EV_OBJ_STOP,
        DAOS_EV_OBJ_READ,
        DAOS_EV_OBJ_WRITE,
        DAOS_EV_OBJ_PUNCH,
        DAOS_EV_EPC_OPEN,
        DAOS_EV_EPC_CLOSE,
        DAOS_EV_EP_SLIP,
        DAOS_EV_EP_CATCHUP,
        DAOS_EV_EP_END,
 } daos_ev_type_t;


 enum {
        DAOS_EVS_FINI,
        DAOS_EVS_INIT,
        DAOS_EVS_INFLIGHT,
        DAOS_EVS_COMPLETED,
 };


 /**
  * Event structure
  */
 typedef struct {
        /** event type */
         daos_ev_type_t                ev_type;
        /**
```

```
         * event status, it can be DAOS_EVS_*, or negative error code.
         */
        int                     ev_status;
        /** reserved space for DAOS usage */
        struct {
                uint64_t        space[15];
        }                       ev_private;
} daos_event_t;


/**
 * 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, if \a ev is NULL, it will wait until EQ is
empty,
  * otherwise it will launch DAOS_EV_EQ_DESTROY as inflight event, then
  * return immediately. DAOS_EV_EQ_DESTROY is guaranteed to be the last
  * event, EQ will be destroyed after DAOS_EV_EQ_DESTROY is polled out.
  * Except daos_eq_query and daos_eq_poll, all attempts of using a
destroyed
  * EQ will fail immediately.
  *
  * \param eqh [IN]    EQ to finalize
  * \param ev [IN]     pointer to completion event
  *
  * \return            zero on success, negative value if error
  */
int
daos_eq_destroy(daos_handle_t eqh, daos_event_t *ev);
/** wait for completion event forever */
#define DAOS_EQ_WAIT         -1
/** always return immediately */
#define DAOS_EQ_NOWAIT               0


/**
  * Retrieve completion events from an EQ
  *
  * \param eqh [IN]    EQ handle
  * \param wait_if [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 n_events [IN]       size of \a events array, returned number
of events
  *                    should always be less than or equal to \a
n_events
```

```
  * \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_if,
              int64_t timeout, int n_events, daos_event_t **events);


 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;


 /**
  * 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 n_events [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 n_events, daos_event_t **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(daos_event_t *ev,
                 daos_handle_t eqh, daos_event_t *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
  */
 void
 daos_event_fini(daos_event_t *ev);


 /**
  * Get the next child event of \a ev, it will return the first child
event
  * if \a child is NULL.
  *
  * \param event [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.
  */
 daos_event_t *
 daos_event_next(daos_event_t *event, daos_event_t *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(daos_event_t *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, daos_event_t *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, daos_event_t *ev);


 /**
  * DAOS storage tree structure has four layers: site, rack, node and
target
  */
 typedef enum {
        DAOS_LOC_TYP_SITE,
        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 site/rack/node/target
 */
typedef struct {
       /** type of this ID: DAOS_LOC_SITE/RACK/NODE/TARGET */
       daos_loc_type_t        lk_type;
       /** logic ID of site/rack/node/target */
       unsigned int           lk_id;
} daos_loc_key_t;


/**
 * placement information of DAOS storage tree
 */
typedef struct {
       /** site number */
       int                    lc_site;
       /** rack number */
       int                    lc_rack;
       /** node number */
       int                    lc_node;
       /** target number */
       int                    lc_target;
} daos_location_t;


/**
 * Query storage tree topology of DAOS
 *
 * \param handle [IN] handle of DAOS context
 * \param loc [IN/OUT]       location of site/rack/node/target:
 *                     a) loc::lc_site is DAOS_LOC_UNKNOWN
 *                        total number of sites will be stored in
 *                        loc::lc_site
 *                        total number of racks will be stored in
 *                        loc::lc_rack
 *                        total number of nodes will be stored in
 *                        loc::lc_node
 *                        total number of targets will be stored in
 *                        loc::lc_node
 *
 *                     b) loc::lc_site is valid but loc::lc_rack is
 *                        DAOS_LOC_UNKNOWN
 *
 *                        total number of racks in loc::lc_site will be
 *                        stored in loc::lc_rack
 *                        total number of nodes in loc::lc_site will be
 *                        stored in loc::lc_node
 *                        total number of targets in loc::lc_site will
be
 *                        stored in loc::lc_target
```

```
  *
  *                    c) both loc::lc_site and loc::lc_rack are valid,
but
  *                       loc::lc_node is DAOS_LOC_UNKNOWN
  *
  *                       total number of nodes in loc::lc_rack will be
  *                       stored in loc::lc_node
  *                       total number of targets in loc::lc_rack will
be
  *                       stored in loc::lc_target
  *
  *                    d) loc::lc_site, loc::lc_rack and loc::lc_node
are
  *                       all valid, total number of targets in
  *                       loc::lc_node will be stored in loc::lc_target
  *
  * \param nlks [IN]   size of array \a lks, it's ignored if \a lks is
NULL
  * \param lks [OUT]   array to store footprint of preorder traversal of
  *                    storage (sub)tree:
  *                    a) loc::lc_site is DAOS_LOC_UNKNOWN, footprints
of
  *                       whole tree traversal will be stored in this
array
  *
  *                    b) loc::lc_site is valid but loc::lc_rack is
  *                       DAOS_LOC_UNKNOWN, footprints of tree traversal
  *                       of loc::lc_site will be stored in this array
  *
  *                    c) both loc::lc_site and loc::lc_rack are valid,
but
  *                       loc::lc_node is DAOS_LOC_UNKNOWN, footprints
of
  *                       tree traversal of loc::lc_rack will be stored
  *                       in this array
  *
  *                    d) loc::lc_site, loc::lc_rack and loc::lc_node
are
  *                       valid, footprints of tree traversal of
  *                       loc::lc_node will be stored in this array.
  *
  * \param ev [IN]     pointer to completion event
  *
  * \return            >= 0 number of lks returned into \a lks
  *                    negative value if error
  *
  * Example: Query all targets in DAOS filesystem
  *   loc.lc_site = DAOS_LOC_UNKNOWN;
  *
  *   daos_sys_query(handle, &loc, 0, NULL, NULL);
  *   nlks = loc.lc_site + loc.lc_rack + loc.lc_node + loc.lc_target;
  *   lks = malloc(nlks * sizeof(*lks));
  *
  *   loc.lc_site = DAOS_LOC_UNKNOWN;
  *   daso_sys_query(handle, &lock, nlks, lks, NULL);
  */
 int
```

```
daos_sys_query(daos_handle_t handle, daos_location_t *loc,
               int nlks, daos_loc_key_t *lks, daos_event_t *ev);

/**
 * bandwidth (MB) to target
 */
typedef struct {
       /** bandwidth between caller site and target site */
       unsigned int           tb_site;
       /** bandwidth between caller rack and target rack */
       unsigned int           tb_rack;
       /** bandwidth between caller and target node */
       unsigned int           tb_node;
       /** bandwidth between caller and target */
       unsigned int           tb_target;
} daos_target_bw_t;


/**
 * detail information of a target
 */
typedef struct {
       /**
        * health status of target, i.e:
        * -2   : unknown
        * -1   : disabled
        *  0 - 100 : health levels
        */
       int                    ti_status;
       /** storage type of the target, i.e: SSD... */
       unsigned int           ti_type;
       /** capacity of the target */
       daos_size_t            ti_size;
       /** free space of target */
       daos_size_t            ti_free;
       /** number of failover nodes of this target */
       unsigned int           ti_nfailover;
       /** network hops to target */
       unsigned int           ti_dist;
       /** reserved for target CPU affinity */
       int                    ti_aff_id;
       /** latency from caller to target (micro-second) */
       uint64_t               ti_latency;
       /** bandwidth information of target */
       daos_target_bw_t       ti_bw;
} daos_target_info_t;


/*
 * Query detail information of a storage target
 *
 * \param handle [IN]        handle of DAOS context
 * \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, daos_location_t *target,
                       daos_target_info_t *info, daos_location_t
*failover,
                       daos_event_t *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 modes */
 /** read-only */
 #define       DAOS_CONT_RO                    (1)
 /** read-write */
 #define DAOS_CONT_RW                  (1 << 1)
 /** create container if it's not existed */
 #define DAOS_CONT_CREATE              (1 << 2)


 /**
  * 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 name [IN]   POSIX name path to container
  * \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 coh [IN/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 *name, unsigned int mode,
                     unsigned int nprocess, daos_handle_t *coh,
                     daos_event_t *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.
  *
  * \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, daos_event_t *event);


 /**
  * destroy a DAOS container and all shards
  *
  * \param name [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 *name, daos_event_t *event);


 /**
  * create snapshot for a container based on its last durable epoch
  *
  * \param name [IN]   POSIX name path to container
```

```
 * \param snapshot [IN]      name of snapshot
 *
 * \return            zero on success, negative value if error
 */
int
daos_container_snapshot(const char *name, const char *snapshot,
                        daos_event_t *event);


/**
 * container information
 */
typedef struct {
       /** the highest committed epoch */
       daos_epoch_id_t coi_epoch_id;
       /** number of shards */
       unsigned int    coi_nshard;
       /** user-id of owner */
       uid_t           coi_uid;
       /** group-id of owner */
       gid_t           coi_gid;
       /** TODO: add more members */
} daos_container_info_t;


/**
 * return shards information, the highest committed epoch etc
 *
 * \param name [IN]   POSIX name path to container
 * \param info [OUT]
 * \param event [IN]  pointer to completion event
 *
 * \return            zero on success, negative value if error
 */
int
daos_container_query(const char *name,
                     daos_container_info_t *info, daos_event_t *event);



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

**************************************************************************
/
 /**
  * Convert a local handle to global representation data which can be
  * shared with peer processes, handle has to be container handle or
  * epoch scope handle, otherwise error will be returned.
  * This function can only be called by the process did collective
open.
  *
  * \param handle [IN] container or epoch scope 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, daos_size_t *size);


/**
 * Create a local handle for global representation data.
 * see details in \a daos_container_open and \a daos_local2global
 *
 * \param coh [OUT]   returned handle
 * \param global[IN]  global (shared) representation of a collectively
 *                    opened container/epoch scope
 * \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, daos_size_t size,
                  daos_handle_t *handle, daos_event_t *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 create object in a container.

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


 /**
```

```
 * Add a new shard to a container
 *
 * \param coh [IN]           container owns this shard
 * \param epoch [IN]         writable epoch of this container
 * \param loc [IN]           placement information of this shard
 * \param shard [OUT]        returned shard ID
 * \param event [IN]         completion event
 *
 * \return                   zero on success, negative value if error
 */
int
daos_shard_add(daos_handle_t coh, daos_epoch_t epoch, daos_location_t
*loc,
              uint32_t *shard, daos_event_t *event);


/**
 * disable a shard for a container
 *
 * \param coh [IN]           container owns this shard
 * \param epoch [IN]         writable epoch of this container
 * \param shard [IN]         shard to disable
 * \param event [IN]         completion event
 *
 * \return                   zero on success, negative value if error
 */
int
daos_shard_disable(daos_handle_t coh, daos_epoch_t epoch,
                   uint32_t shard, daos_event_t *event);


typedef struct {
       /**
        * status of shard, health-level/unhealthy/disabled which is
identical
        * to target status
        */
       int                   sai_status;
       /** number of non-empty object */
       daos_size_t           sai_nobjs;
       /** space used */
       daos_size_t           sai_size;
       /** shard location */
       daos_location_t       sai_loc;
       /* TODO: add members */
} daos_shard_info_t;


/**
 * query a shard, i.e: placement information, number of objects etc.
 *
 * \param coh [IN]           container handle
 * \param epoch [IN]         epoch of this container
 * \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,
                 uint32_t shard, daos_shard_info_t *info,
                 daos_event_t *event);


/**
 * Flush all (changed) 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,
                 uint32_t shard, daos_event_t *event);


/**
 * enumerate non-empty object IDs in a shard
 *
 * \param coh [IN]           container handle
 * \param epoch [IN]         epoch of this container
 * \param shard [IN]  =09shard ID
 * \param obj_off [IN]               offset of object ID to list
 * \param count [IN]         size of \a objids array
 * \param objids [OUT]               returned object IDs.
 * \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,
                    uint32_t shard, daos_off_t obj_off,
                    daos_size_t count, daos_obj_id_t *objids,
                    daos_event_t *event);


/**********************************************************
 * Object API
 **********************************************************/
enum {
       DAOS_OBJ_RO           = (1 << 1),     /** shared read */
       DAOS_OBJ_RO_EXCL      = (1 << 2),     /** exclusive read */
       DAOS_OBJ_RW           = (1 << 3),     /** shared write */
       DAOS_OBJ_RW_EXCL      = (1 << 4),     /** exclusive write */
       DAOS_OBJ_IO_RAND      = (1 << 5),     /** random I/O */
       DAOS_OBJ_IO_SEQ       = (1 << 6),     /** sequential I/O */
};


 /**
```

```
 * start 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_start(daos_handle_t coh,
                  daos_obj_id_t oid, unsigned int mode,
                  daos_handle_t *oh, daos_event_t *event);


/**
 * stop 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_stop(daos_handle_t oh, daos_event_t *event);


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


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


/**
 * IO fragment of a DAOS object
 */
typedef struct {
       daos_off_t     if_offset;
       daos_size_t    if_nob;
```

```
} daos_io_frag_t;


/**
 * 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
 */
typedef struct {
      unsigned long  iod_nfrag;
      daos_io_frag_t iod_frag[0];
} daos_iod_t;


/**
 * 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,
                 daos_mmd_t *mmd, daos_iod_t *iod, daos_event_t *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,
                  daos_mmd_t *mmd, daos_iod_t *iod,
                  daos_event_t *event);


/**
```

```
  * flush all (cached) writes up to the give epoch to a object
  *
  * \param oh [IN]     object handle
  * \param epoch [IN]  epoch to flush
  * \param event [IN]  completion event
  */
 int
 daos_object_flush(daos_handle_t oh,
                   daos_epoch_t epoch, daos_event_t *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,
                   daos_event_t *event);


 /*************************************************************
  * Epoch & Epoch functions
  *
  * Version numbers, called epochs, serve as transaction identifiers
and are
  * passed in all DAOS I/O operations.
  *
  * Epochs are totally ordered.  An epoch becomes consistent only after
all
  * prior epochs are consistent and all writes in the epoch itself have
  * completed. It is therefore an error to attempt to write in a
consistent
  * epoch since all valid writes in such epochs have completed already.
  * Writes belonging to epochs that can never become consistent (e.g.
due to
  * some failure) are discarded. Readers may query the current highest
  * committed epoch (HCE) number and use it on reads to ensure they see
  * consistent data. DAOS effectively retains a snapshots of the HCE
given
  * to any reader while such readers remain to provide read consistency
and
  * allow concurrent writers to make progress. When all readers for an
old
```

```
  * snapshot have departed the snapshot becomes inaccessible and space
is
  * reclaimed.
  *
  * Epochs are used within an epoch scope. Each epoch scope covers a
unique
  * set of filesystem entities that may be affected by transactions
using
  * the scope. Epoch scopes may not overlap - i.e. cover the same
filesystem
  * entities. Currently there is a 1:1 mapping between epoch scopes and
DAOS
  * containers. A single epoch scope covers a single DAOS container and
exists
  * for the lifetime of the container therefore transactions may only
span
  * a single container and all transactions within a container are
executed
  * in the same epoch scope and exist in the same total order. Note
that the
  * lifetime and coverage of an epoch scope may be made more flexible
in the
  * future.
  ************************************************************/
 #define DAOS_EPOCH_HCE       {-1, -1}


 /**
  * Open an epoch scope on the specified container(s). Epoch returned
is the
  * highest committed epoch (HCE) of container and it is guaranteed not
to
  * disappear until it is slipped or the epoch scope is closed.
  *
  * \param coh [IN]    container handle for this epoch
  * \param eps_h[OUT]  handle of epoch sequence
  * \param hce[OUT]    returned HCE
  * \param event [IN]  pointer to completion event
  */
 int
 daos_epoch_scope_open(daos_handle_t coh, daos_epoch_id_t *hce,
                       daos_handle_t *eps_h, daos_event_t *ev);


 /**
  * Closes the epoch scope.
  * If 'error' is set, all writes in epochs that are not yet marked
  * consistent are rolled back. Behaviour is undefined if 'error' is
zero
  * but writes in epochs that have yet to be marked consistent are
  * outstanding since it's impossible to determine whether all writes
  * in all epochs have been marked consistent until the epoch scope is
  * closed by all processes holding it open.
  *
  * \param eps_h [IN]  epoch sequence to close
  * \param error       [IN]    error code
  * \param event [IN]  pointer to completion event
```

```
  */
 int
 daos_epoch_scope_close(daos_handle_t eps_h, int error, daos_event_t
*ev);


 /**
  * Complete when \a epoch is durable. If *epoch is DAOS_EPOCH_HCE, it
  * sets \a epoch to the actual highest committed epoch and completes
  * immediately. If \a ev is NULL, it completes immediately but returns
  * failure if epoch is not currently durable. If is completes
successfully,
  * the reference on the epoch sequence's previous durable epoch is
moved
  * to the epoch returned.
  *
  * \param eps_h [IN]  epoch sequence handle
  * \param epoch[IN/OUT] epoch to slip to, if it's DAOS_EPOCH_HCE or
the
  *                     given epoch is garbage collected, then epoch
number
  *                     of HCE is returned.
  * \param ev [IN]     pointer to completion event
  */
 int
 daos_epoch_slip(daos_handle_t eps_h,
                 daos_epoch_id_t *epoch, daos_event_t *ev);


 /**
  * Returns an epoch number that will "catch up" with epoch number
usage
  * by other processes sharing the same epoch scope.
  * This ensures that processes executing a series of long running
  * transactions do not delay short running transactions executed in
the
  * same epoch scope.
  *
  * \param eps_h [IN]  epoch sequence handle
  * \param epoch[OUT]    returned "catch up" epoch which is best for
write
  * \param ev [IN]     pointer to completion event
  */
 int
 daos_epoch_catchup(daos_handle_t eps_h,
                    daos_epoch_id_t *epoch, daos_event_t *ev);

 /**
  * Signals that all writes associated with this epoch sequence up to
and
  * including 'epoch' have completed. Completes when this epoch becomes
  * durable.
  * If commit is failed, it's just like the commit hasn't happened yet,
HCE
  * remains it was, so user can find out failed shards and disable
them,
  * and commit again.
```

```
 * If it's a fatal error, user should call daos_epoch_scope_close with
 * setting abort flag to discard changes.
 *
 * \param eps_h [IN]  epoch sequence handle
 * \param epoch[OUT]  epoch to complete
 * \param ev [IN]     pointer to completion event
 */
int
daos_epoch_commit(daos_handle_t eps_h,
                  daos_epoch_id_t epoch, daos_event_t *ev);
```