Date: December 13, 2012	# SOLUTION ARCHITECTURE - DAOS # 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
2012-12-13	1.0 Draft for Review	Eric Barton, Intel Corporation
2012-12-21	1.1 Updated with indication of mandatory requirements	Eric Barton, Intel Corporation

Introduction

The ***Distributed Application Object Storage*** (***DAOS***) API is an object-based I/O API designed for scalability and resilience to create a foundation for building future I/O stacks. It replaces the POSIX file abstraction with the DAOS container that provides an object address space partitioned over storage targets. Each partition, called a ***container shard***, is the unit both of concurrency and of fault-tolerance. Higher levels in the I/O stack may therefore distribute I/O over objects in different container shards to achieve both horizontal scalability and resilience. DAOS provides mechanisms that greatly simplify the implementation of ACID transactions on multiple objects by cooperating processes. This ensures that application and higher-level I/O stack data and metadata stored in DAOS containers remains self-consistent after all possible failures. Finally, the DAOS API supports concurrency by using non-blocking initiation procedures and completion events.

Solution Requirements

Requirements listed below are mandatory unless explicitly labeled desirable.

1. Storage Targets

- A DAOS system consists of a hierarchy of storage targets representing fault domains and resource contention.
  - A DAOS ***storage target*** is the basic unit of storage subsystem in DAOS.
    - A storage target typically comprising a single exported backend storage volume.
    - Storage targets may have widely different properties including capacity, bandwidth, IOPs, resilience and availability.
    - Failures of any type in any part of a DAOS system may cause affected storage targets to fail temporarily or permanently. All operations on a failed storage target complete in finite time with error.
  - A DAOS ***node*** consists of a set of storage targets located on the same server.
    - Failure of a node will affect all storage targets currently exported by that node.
    - Storage targets exported by a single node may remain accessible on node failure if they are capable of failover, otherwise they become inaccessible.
    - Storage targets on the same node contend for node resources such as network interface bandwidth and CPU.
  - A DAOS ***rack*** consists of a set of nodes that share critical resources such as networking and power.
    - Failure of a resource critical to the operation of the rack may cause some or all of its nodes to fail.
    - Nodes in the same rack contend for rack resources such as network bandwidth and power.
  - A DAOS ***site*** consists of a set of racks co-located in a single physical location.

        - Failure of a resource critical to the operation of the site may cause some or all of its racks to fail
        - Racks in the same site contend for site resources such as external network bandwidth.
- DAOS numbers items in the storage hierarchy as follows…
    - storage targets are numbered consecutively within the same site – i.e. site.target uniquely specifies a specific storage target (e.g. one capable of failover) irrespective of its current node or rack.
    - Nodes are numbered consecutively within their rack.
    - Racks are numbered consecutively within their site.
    - Sites are numbered consecutively globally.
- DAOS provides queries to enumerate the storage hierarchy and to determine the caller's position relative to it.
    - This enables higher levels of the I/O stack to implement different resilience and scaling schemes.
    - Distributed applications may use this to determine how to load balance and maximize performance.

2. Containers

- A DAOS container is a new type of file in the POSIX address space.
    - A DAOS container can be regarded as a virtualized set of storage targets that enables multiple users safely to share a distributed storage system while retaining the scalability and resilience benefits that direct access to the system's underlying storage targets provides.
- A DAOS container consists of a set of ***container shards***.
    - Each container shard is placed on a single storage target.
        - Higher levels of the I/O stack determine the storage target to use for a particular shard based on scalability and resilience requirements.
        - Distributed applications may query the storage target served by a given container shard to determine the "distance" at site, rack and node level from any process to any container shard to support load balancing and maximize performance.
    - Container shards are numbered contiguously from 0.
    - On creation, a DAOS container has 0 shards.
    - New shards may be added at any time.
    - Shards cannot be removed, only marked 'inactive' to advise callers that all operations affecting the shard will fail.
- A DOAS container must be opened before I/O can be performed on it.
    - Open returns a handle which may be used in subsequent DAOS I/O calls.
    - Flags passed to open a container may specify that it should be created if it does not already exist.
    - Normal POSIX access permissions are applied on open.
    - DAOS supports collective open across peer processes running in the same cluster.
        - A process with an open container handle may convert the handle to a global handle which may then be passed to peer processes running in the same cluster.

        - The global handle is an opaque structure that includes all information required to provide access and enforce security on peer processes (e.g. listing of the container's shards and capabilities to enforce security).
        - Peer processes may then then convert the global handle back to a local handle which can be used for I/O.
        - Peer processes are not guaranteed continued access to the container if the opening process closes its handle or exits.
        - All processes participating in a collective open are granted the same access rights.
    - A DAOS container may be opened for read by multiple processes.
        - These processes may be entirely independent.
    - A DAOS container may only be opened for write by a single process.
        - Cooperating processes must use collective open for write access.
- The original opener of a container can initiate an operation that completes when the container layout changes.
    - This is used to notify readers that a process or parallel job that has the container open for write has added or disabled shards.
    - The original opener should then notify its peers of this change and pass them a new global handle since peers using the previous handle will not be able to access added shards until they have received the new handle and converted it to a local handle.
- POSIX stat(2) and fstat(2) work on a file of type DAOS container.
    - st_dev, st_ino, st_mode, st_nlink, st_uid, st_gid, st_rdev, st_blksize work as expected.
    - st_size returns 0.
    - st_blocks returns the total space.
    - st_atime returns an approximate time of last access to any container shard.
    - st_mtime returns the most recent time any shard of the container was modified or a new shard was added.
    - st_ctime returns the most recent status change of the container including modifications or inode information.
- DAOS calls must be used for I/O within a DAOS container.
    - POSIX I/O calls on a DAOS container (e.g. read(2), write(2)) fail, returning EINVAL.

3. Objects

- DAOS objects inhabit a partitioned address space.
    - High order bits address a single container shard.
    - Low order bits address a single object contained entirely within the single container shard addressed by the high order bits.
- DAOS objects are not inherently resilient.
    - I/O errors on the storage target exporting an object's container shard are passed back to the DAOS API.
        - Higher levels of the I/O stack may exploit redundancy across container shards (e.g. replication, RAID, erasure codes) to handle such errors transparently.
    - DAOS may not detect "silent" data corruption.

        - Higher levels in the I/O stack may implement end-to-end integrity checks e.g by checksumming data and storing the checksum metadata in additional DAOS objects.
- A DAOS object is a mutable byte array.
    - The contents of a DAOS object are addressed by offset.
    - Arbitrary bytes in the array may be written or overwritten.
    - The performance of sequential or random I/O by any process to a given object is determined by the properties of the relevant storage target and the process's "distance" to that storage target.
- DAOS objects are neither created nor destroyed.
    - Objects initially consume no space and all reads return 0s.
    - Writes cause space allocation on the underlying storage target.
    - "***Punch***" is a special type of write that resets the extent written to contain 0s and may free space in the extent punched previously allocated by writes.
- DAOS objects have no metadata.
    - Traditional metadata (e.g. size, mtime etc) is not maintained by DAOS to avoid penalizing usage that does not require it.
    - Higher levels in the I/O stack should maintain whatever metadata they require – e.g. by using additional DAOS objects to store it.
- DAOS objects must be opened before they are read or written.
    - Open is simply used to inform DAOS of the type of access that will be required e.g whether the object will be written, if I/O will be streaming or random and whether access will be shared or exclusive to the calling process.
    - This is used to help determine whether and how the object can be cached locally.
- DAOS supports scatter/gather I/O on individual objects.
    - Caller provides an array of object offsets and sizes.
    - Caller provides an array of memory buffers and sizes.
    - The number of object and memory fragments need not match.
    - The total size of all object and memory fragments must match.

4. Transactions

- DAOS uses explicit versioning to enable higher levels of the I/O stack to implement I/O transactions with the following properties…
    - **No implicit serialization** – throughput is maintained by making callers solely responsible for resolving conflicting transactions.
    - **Multiple threads** – any number of threads and/or processes may participate in the same transaction.
    - **Multiple objects** – any number of DAOS objects may be written in the same transaction.
    - **Atomic writes** – either all writes in a transaction are applied or none of them are.
    - **Commutative writes** – concurrent writes are effectively applied in version order, not time order.
    - **Consistent reads** – all reads in a transaction may "see" the same version data even in the presence of concurrent writers.

- ***Epochs*** serve as transaction identifiers. An epoch has a ***scope*** that covers a unique set of DAOS objects that may be read or written by the transaction and a ***number*** that specifies a version to use in the transaction.
    - All DAOS I/O operations include an epoch parameter.
        - Read specifies an epoch to ensure multiple reads "see" a consistent version of the data.
        - Write (including punch) specifies an epoch to ensure multiple writes are applied atomically.
    - Epochs sharing the same scope are numbered in a total order.
        - Transactions using the same epoch (i.e. identical scope and epoch number) are effectively aggregated.
        - A failed transaction causes all concurrent transactions with epochs in the same scope but a higher epoch number to fail.
        - Processes executing transactions in the same scope must collaborate to minimize cascading aborts.
        - Processes executing transactions in the same scope must collaborate to recover from storage target failure
    - Different epoch scopes may not overlap - i.e. they may not cover the same DAOS objects.
    - An epoch scope is persistently represented on all storage targets containing objects covered by the scope.
        - Each ***epoch scope shard*** maintains local information on the state of transactions executed within the scope.
    - In the initial DAOS development there will be a 1:1 correspondence between epoch scopes and DAOS containers.
        - The epoch scope is determined from an open DAOS container handle.
        - Write transactions can only be completed in the epoch scope if the DAOS container is open for write.
        - Transactions may only span a single container.
        - All transactions within a container are executed in the same epoch scope and apply in the same total order.
        - The lifetime and coverage of an epoch scope may be made more flexible in the future – e.g. to change the defined lifetime of an epoch scope and to allow multiple epoch scopes in a single container and/or multiple containers in a single epoch scope.
- An epoch becomes consistent and its writes are guaranteed visible only after all writes in the epoch scope with equal or lower epoch numbers have completed and any writes cached locally have been ***flushed***.
    - DAOS requires explicit notification of consistency – i.e. it is the responsibility of higher levels in the I/O stack to detect when all writes in an epoch (and all prior epochs) have completed. Callers may then ***commit*** the epoch to establish a new ***highest committed epoch (HCE)***.
        - Higher levels in the I/O stack may commit an epoch even though some writes failed – e.g. a library using replication to implement resilient storage may fail to write one of the mirrors but still allow the transaction as a whole to succeed.

    - Commit completes when DAOS has attempted to persist the new HCE on all shards in the epoch scope. If any shard in the epoch scope fails to commit, the epoch becomes ***stuck*** and the commit completes with failure.
        - HCE remains the same.
        - Callers may enumerate the shards in the scope to discover which stuck so they can be disabled.
        - Callers may retry the commit.
    - It is an error to attempt to write in an epoch that is already committed since all valid writes in that epoch are considered to have completed already.
    - Writes in epochs that can never become committed (e.g. due to an abort) are discarded.
- Processes must open an epoch to perform I/O on objects covered by the epoch's scope.
    - The scope is specified by passing an open DAOS container handle.
    - The epoch returned on completion contains the handle of the epoch scope and the scope's HCE.
        - The handle is read-only.
        - The epoch number may be altered to specify different versions within the same scope.
    - A snapshot of all objects covered by the scope at the HCE is retained until the epoch is closed or ***slipped*** (see description below).
        - The snapshot includes all transactions up to and including the HCE – i.e. all data in the snapshot is from complete transactions.
        - When the scope is closed or slipped the HCE snapshot may become inaccessible and storage may be reclaimed.
    - The epoch may be ***slipped*** to obtain a more recent HCE.
        - Slipping notifies DAOS that the current HCE snapshot may be released and a snapshot at the new HCE should be retained.
        - Slipping completes when the HCE in the specified epoch's scope equals *or exceeds* the specified epoch's number.
        - This is useful to receive notification when another process or job has completed new write transactions in the same epoch scope - e.g. a simulation has added more timestep data to its output file and an analysis job wants to be notified of the presence of new consistent data.
        - Any epoch number less than or equal to the current HCE (e.g. 0) may be specified to retrieve the current HCE. Completion is immediate.
        - The current HCE+1 may be specified for notification of any change.
    - The epoch may be passed to *any* collaborating process to enable it to perform I/O within the epoch scope.
        - This does not circumvent normal security checks – peers must open containers to transact with objects in the scope.
        - Similarly to collective open on containers, the epoch must first be converted to an opaque global representation.

        - Peer processes may then convert this global representation back to normal for use in local I/O.
        - An epoch passed in this way may only be used for I/O – i.e. not to slip or end the epoch.
        - It is an error for peer processes to use an epoch passed in this way after the original opener has closed the epoch.
    - Passing the HCE on reads guarantees consistency across an arbitrary set of reads within the epoch's scope.
        - All reads by any process specifying this epoch read from the same snapshot created when the epoch was opened.
        - Concurrent writers may continue to make progress without conflicting with readers.
    - Readers may use epoch numbers other than the HCE.
        - These reads are not guaranteed to return consistent data.
        - Reads with an epoch number higher than the HCE may fail or return a mix of data from the HCE and subsequent epochs up to the epoch number specified.
        - Reads in an epoch number lower than the HCE may return a mix of data from prior epochs, including epoch 0 (all 0s).
    - Writers must use epoch numbers greater than the HCE.
        - Any number of write transactions may execute concurrently.
        - Writes in the same transaction must use the same epoch (scope and number).
        - Conflicting writes in the same transaction apply in arbitrary order.
        - Conflicting writes in different transactions apply in epoch number order – i.e. irrespective of time order.
        - The process that opened the epoch may commit an epoch to notify DAOS when all writes by it and its peers in the epoch's scope, up to and including the epoch's number have completed and been flushed.
    - Multiple processes may open epochs in the same scope for read and/or write.
        - These processes are effectively transaction leaders for their sub-groups of peers.
        - These processes may initiate an epoch slip or commit independently.
        - HCE snapshots remain in existence until all relevant epochs have been slipped or closed.
        - The HCE never advances beyond the minimum epoch committed by any process that opened the scope for write.
        - If there is an error during commit, the transaction leaders must cooperate to determine how to handle the failure.
        - Groups of processes performing transactions independently within the same epoch scope may ***catchup*** to ensure one group does not hold back the HCE unnecessarily.
            - A series of long running transactions using consecutive epochs will consume epochs more slowly than a similar series of short running transactions. Since epochs are totally ordered, these long running transactions will prevent the short running transaction commit from completing.

            - Catchup on an open epoch returns an epoch with a number approaching the maximum recently ended in same scope.
    - An epoch is ***aborted*** if it is open in a process that terminates unexpectedly or if the 'abort' flag is set when the epoch is closed.
        - This causes all other open instances to abort.
        - All operations on an aborting epoch complete with failure.
        - If the epoch is stuck in a commit, all writes above the stuck epoch are discarded. Otherwise all writes above the HCE are discarded.
            - Such writes that have already been applied at storage targets are rolled back
            - All other writes become no-ops and complete with failure.
        - On completing rollback, epoch shards record a new HCE above the maximum write rolled back on any shard.
        - All new attempts to open an epoch with the same scope block until recovery completes and all open epochs sharing the same scope have closed.
        - If recovery fails to complete on any storage target, the epoch becomes stuck in recovery. Only processes able to handle a stuck epoch may then open the epoch, enumerate stuck shards, disable them and retry the abort.

5. Caching

- Caching behavior applies to whole objects, not object extents.
- Writes are globally visible when their epoch is committed irrespective of cache behavior.
    - Reading at the HCE guarantees a consistent view.
- Objects opened for shared write are not cached.
- Objects opened for exclusive write may be cached.
    - Other processes may not open these objects for write.
    - Flush must complete before the epoch may be ended.
- Objects opened read-only may be cached.
    - Slipping the HCE may invalidate the cache.

6. Non-blocking I/O

- All DAOS operations that cannot complete immediately return after initiating the operation and post an ***event*** to an ***event queue*** to signal completion asynchronously.
    - A pointer to a caller-allocated event structure is passed in all such calls. When the operation is initiated the event becomes ***active***.
    - An active event becomes ***idle*** once its associated operation has completed and it has been removed from its event queue.
    - Some fields in the event (e.g. completion status) become volatile while the event remains active.
    - It is an error to initiate an operation with an event that is active.
    - Events may be aggregated so that a single parent event can be used to signal completion of multiple child events.

- Events are initialized once and may then be used multiple times before being finalized.
    - Events are initialized with an owner and are posted to the owner on completion.
    - The owner may be an event queue, or a parent event.
    - The parent event must be idle when a child event is added.
    - The parent event becomes active when any child becomes active and completes when all children have completed. All child events become idle when the parent becomes idle.
    - Callers may iterate safely over the children of a parent event at any time.
    - The completion status of a parent event is success if no child event completed with error.
        - Callers may iterate over child events to determine specific completion status.
    - Event aggregation may only be 1 level deep – i.e. it is an error to aggregate aggregated events.
    - When a child event is finalized, it is disassociated from its parent. The parent event must be idle at this time.
- In-flight operation may be aborted via its corresponding event.
    - Abort may be applied to a single event or a parent event. In the latter case, the abort applies to all child events.
    - Abort is fundamentally "racy" since it executes concurrently with normal completion. It simply signals that the caller wishes the associated operation to complete as soon as possible and that completion with failure is acceptable.
- DAOS allows callers to poll or block on an event queue.
    - Multiple events may be returned in a single call.
        - Child events are never returned directly – only the parent event is returned and the caller may then iterate over the children
    - If any events have been posted, return is immediate.
- Higher levels in the I/O stack may build their own notification subsystems – e.g. using a thread pool with multiple event queues to build a callback system that scales on SMP nodes.

Use Cases

1. Simulation

Normal operation

A parallel job continues a simulation from where it left off, adding every 'n'th timestep into the DAOS container that holds the data for the complete simulation run.

Rank 0 in the simulation job opens the container. It uses the handle so obtained to open an epoch, to query the number of shards in the container and to read object 0 on shard 0 which contains the simulation metadata table describing the simulation data in the container, including the number of timesteps already calculated and addition metadata about each timestep dumped.

Rank 0 then converts both handles to global handles and broadcasts these along with the shard count and the contents of the metadata table to all ranks. The peers all convert these handles back to local handles and all ranks increment the epoch number.

All ranks agree on the object number they will write timestep 't' data to using the formula (shard = rank % nshards, object = rank << 32 + t).

All ranks now proceed with the simulation. When any rank has completed an agreed number of timesteps, it initiates write of the timestep data, waits for these to complete, initiates flush of the epoch and proceeds to compute the next timestep. Rank 0 also updates the simulation metadata to record the timestep dumped and how the timestep data is distributed before it flushes.

Each rank checks periodically when its flush has completed and when it has, participates in a pair of non-blocking collective communications over all ranks.

The first of these is the global AND of I/O success across all ranks. When this completes and all ranks had success, rank 0 proceeds to commit the epoch and broadcast the commit status using the second collective.

Failure

If the job terminates unexpectedly in the scenario above, DAOS aborts the epoch and all writes are rolled back to the highest committed epoch.

If any process has an I/O error, this is signaled by the first collective. Rank 0 may then abort the epoch to cause all writes to roll back, query the storage system to check and/or wait for all storage targets hosting container shards to return to server and use the second collective to inform all ranks to retry all writes. This process of abort and retry may be abandoned at any time by terminating the job and all uncompleted writes will be rolled back.

2. Producer/consumer

An analysis program opens the simulation output container while the simulation is running. It opens its own container for analysis output and reads metadata describing the number of timesteps previously analysed. It then reads the simulation output container and checks whether there are new timesteps to analyse. If there are some, it proceeds to read and analyse the new timesteps and dump the results into its own container. It then waits for the highest committed epoch to increase, and repeats the process.

3. Replication over container shards

An HA library is used by the applications described above to be tolerant to server failures. This library exploits reserves high bits of the object address space for its own (replicated) metadata objects recording application object usage and maps application object addresses to replicated objects using a deterministic pseudo-random mapping that ensures replicated objects do not share the same node.

When there is an I/O failure or a commit failure, the HA library retries in case the failure is transient, but in case of persistent failure it stops using and disables the relevant container shard to ensure it does not participate in commit. If the container is open for write, it proceeds, concurrently with the application to reconstruct new replicas of objects that were replicated on the disabled shard.

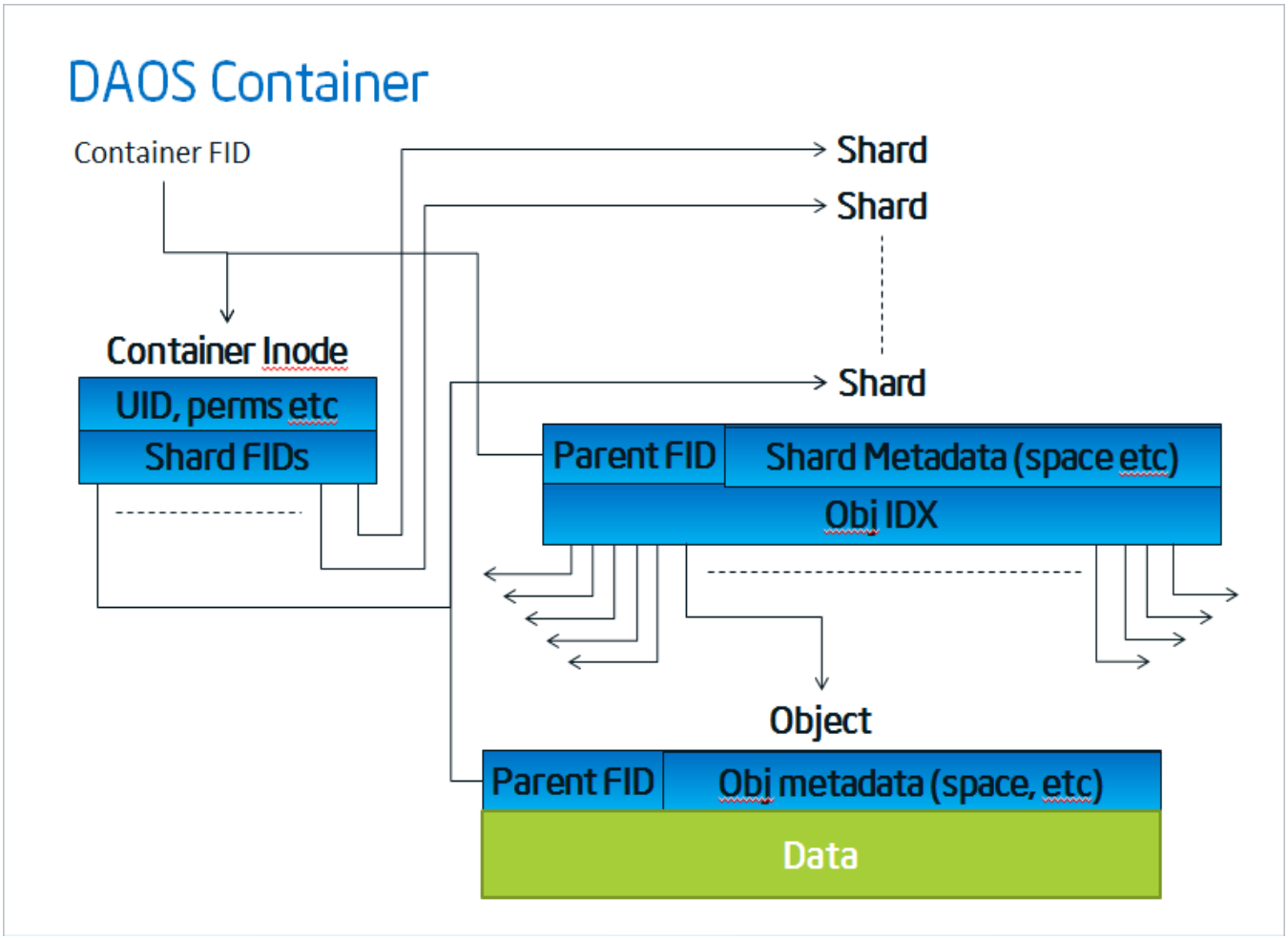
Solution Proposal

1. Containers

A container will be implemented as an inode of type "DAOS container" stored on a Lustre MDS. The layout will consist of a set FIDs that refer to shard objects distributed over the Lustre OSTs. The implementation of shard objects depends on the VOSD implementation – either a conventional Lustre index object referencing Lustre data objects in a single ZFS fileset, or a new type of BTRFS file with an extent map indexed by object number, then offset. In either case, it must be possible to snapshot the entire shard in O(0) time.

The layout stored on the MDS will be definitive, however in case of MDS failure, the layout will also be replicated and timestamped in a reserved object on all shards so that the MDS inode can be reconstructed from the shards on any OSTs that survive whatever calamity befell the MDS. These replicas will be updated using normal DAOS transactions to ensure consistency.

Additional reserved objects and/or extended attributes will be used for shard metadata such as the HCE and the shard's version intent logs.



2. Collective Container Open

When an application opens a DAOS container the container layout listing all shard FIDs will be returned. A secure implementation should also return a capability that the client can present on all shard requests to verify it has passed permission checks on the MDS. Information needed to check the capability will be broadcast to all shards using a server collective.

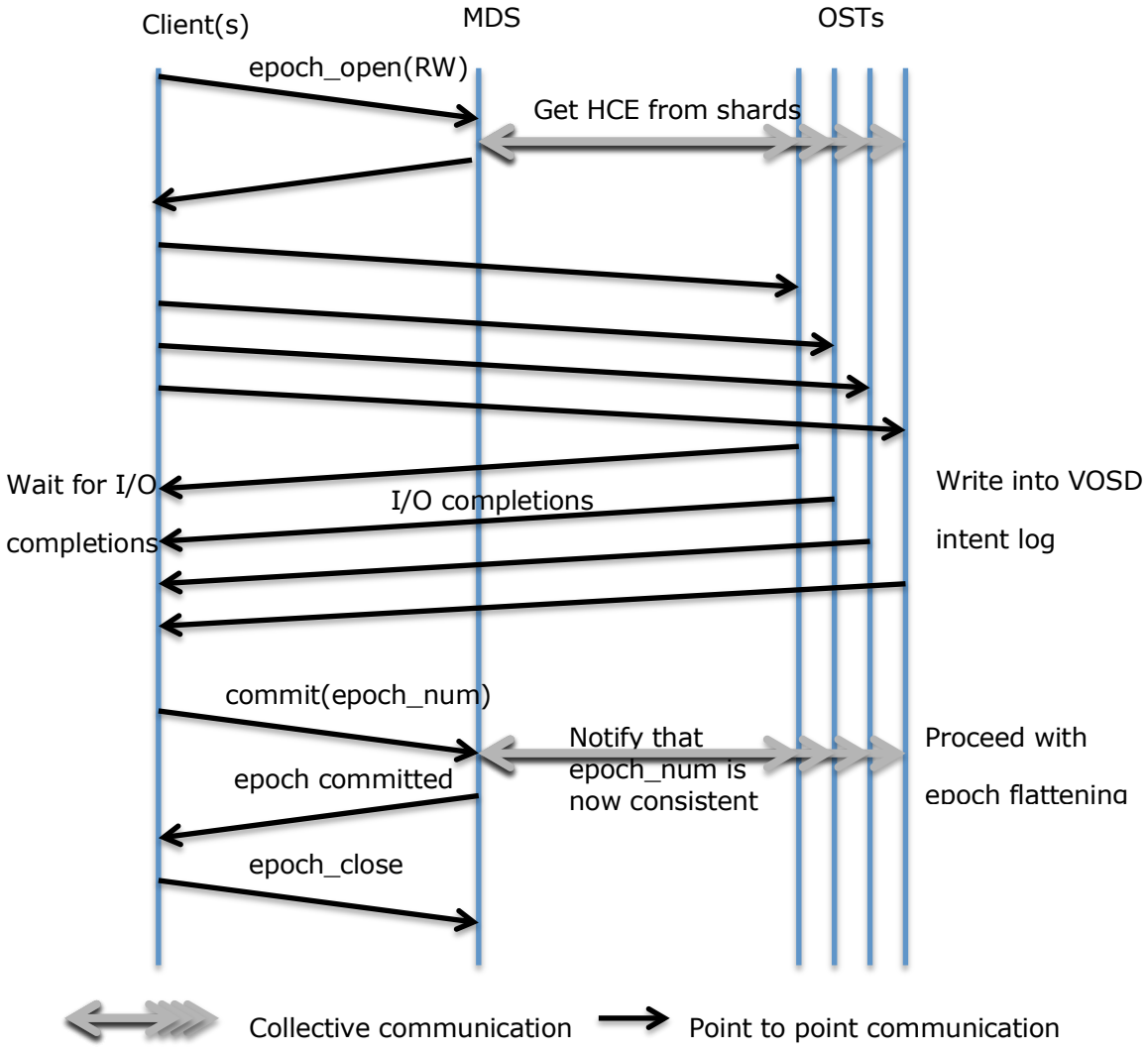
When the client creates a global handle to distribute to its peers, DAOS packs the container's FID and layout (plus capability) in the global handle and these are saved when DAOS converts back to a local handle at the peer for use in requests to the OSTs.

3. Epochs and Recovery

Epoch open communicates with the MDS to establish any necessary state required to obtain and slip the HCE and to commit or abort epoch. In the initial prototype implementation, each container is a separate epoch scope, therefore this state

references the container's inode and uses its container shards on the OSTs to maintain epoch state for each scope shard.

Commit only occurs after all clients with the epoch open for write initiate commit. It uses the minimum epoch number committed by any client. This initiates a server collective which informs every enabled container shard to commit and computes the global AND of success. Commits initiated by all clients with this epoch number are then completed with this status. If the commit was successful, the HCE is advanced and any clients waiting for this are notified. Otherwise the commit is stuck. Clients may keep retrying the commit, possibly disabling one or more shards until commit succeeds. Otherwise the epoch stays stuck in commit.



If a client aborts an epoch or is evicted, the MDS uses a server collective to notify all enabled container shards to start to fail I/O in the aborting epoch and any higher epochs, and roll back any writes that have already been applied. Meanwhile the MDS blocks all new requests to open the epoch and fails all requests on epochs open for write until these have closed and recovery has completed or become stuck.

A stuck epoch may be abandoned, but recovery must continue if it is opened again. On the first open of an epoch, a server collective is used to query the HCE across all enabled container shards.

- If all shards return the same HCE and there are no uncommitted writes on any shards, the epoch open is successful.
- If all HCEs are the same but there are uncommitted writes, it means the epoch is stuck in abort.
- If the HCE on some shards is different from that on others it means a previous commit failed. If there are only 2 different values of HCE across all the shards the epoch is stuck in commit.
- Otherwise the container is corrupt and administrative action is required to repair it.

If a stuck epoch is being opened for write the open succeeds only if the client specified it could handle stuck epochs and a special status is returned to indicate whether the epoch is stuck in commit or abort.

Unit/Integration Test Plan

1. Unit tests

Event Queues and Events

- EQ create/destroy/poll
  - Add test functions to launch an abortable timed NOOP.
  - Create an EQ.
  - Initiate a set of timed NOOPs.
  - Verify # inflight events and completion occurs as expected.
  - Repeat same test (without verifying inflight events) with 2 threads, one initiating and the other polling for completion.
  - Destroy the EQ, possibly while some operations are in-flight.
  - Verify all in-flight operations (if any) complete.
  - Verify last event is EQ destroy completion.
- Parent event operations
  - Verify that completion of child event is invisible to poll.
  - Verify parent event completes only if all child events are completed.
  - Enumerate all child events, racing with completion.
- Abort event
  - Ensure poll should returns aborted event with error code.

- Ensure parent event aborting aborts all child events.

DAOS storage tree

- Sys open and close
  - Verify functionality of sys-container open and close
- Enumerate storage tree
  - Verify DAOS API returns correct number of site/rack/node/target in DAOS storage system.
  - Verify correct enumeration all storage targets
  - Verify correct query on all storage targets
  - Verify process locality queries

Containers

- Create and unlink container
  - Simply verify functionality of container creation and unlink
- Collective open and close
  - Verify functionality of collective operations, all peers share the collective open can verify open handle by querying the opened container.
- Change container layout by adding/disabling shards
  - Verify functionality of adding/disabling shards for a container
  - Verify functionality of shard query
  - Ensure container layout can be changed only if container has been opened for write
  - Change shards in many different epochs, commit/abort any of those epochs and ensure container layout is correct in each epoch.
  - Repeat previous test while rebooting any set of storage targets and the MDS
- Container snapshot
  - Create a snapshot for a container then change layout of the same container, ensure layout of snapshot is unchanged.

Object tests

- Object start and stop
  - Verify functionality of object start and stop
  - Verify exclusive mode can exclude other process from starting the same object.
- Object I/O
  - Verify functionality of read/write calls
  - Verify writing to a nonexistent shard fails.
  - Verify object can be written only if it's started with writable mode
  - Verify scatter/gather I/O
  - Verify HCE read can see all flattened changes in earlier epochs
  - Verify HCE read can see consistent data, whether objects are in the same shard or in multiple shards.
  - Verify read on non-existent object, or non-existent extent of an object gets all-zero bytes.
  - Verify reading an object in a non-existent epoch gets all-zero bytes.
  - Verify aborting epoch will discard all object changes in that epoch scope.
- Object punch

    - Verify object punch can zero bytes in the specified range.
    - Verify object punch from 0 to infinite can recycle the object ID and space.
    - Verify aborting an epoch can restore all punched data and object IDs.
- Verify all I/O using readers and writers on separate nodes and with and without object caching enabled.
- Object enumeration
    - Verify functionality of object enumeration in a shard
    - Verify object enumeration can see object ID difference between epochs
- Container snapshot with objects
    - Create snapshot for a container with objects, then continue to change objects, verify object in snapshot is still consistent.

Epoch tests

- Epoch commit, abort and HCE
    - Verify epoch commit updates HCE
    - Verify aborting epoch will not change HCE.
    - Ensure abort or commit epoch earlier than HCE fails
    - Ensure simulated failure or absence of a storage target during commit causes commit to become stuck
    - Ensure retry of commit when simulated failure is removed or storage target is restarted or affected container shard is disabled succeeds
    - Ensure simulated failure during abort causes abort to become stuck
    - Ensure retry of abort when simulated failure is removed or storage target is restarted or affected container shard is disabled succeeds.
- Multiple reader processes
    - Test with multiple readers and one writer
    - Verify reader processes can slip to the latest HCE
    - Ensure slip blocks until a new HCE commits.
    - Verify old HCE is garbage collected only when all epochs have slipped to a newer HCE.
- Multiple instances of an epoch open for write
    - Test with collaborating processes writing and one reading.
    - Verify commit completes only when all writers have committed
    - Verify when an epoch is aborted by one writer, all further operations in the epoch or higher fail.
    - Verify functionality of epoch "catch up"

4. Integration tests

IOD/HDF5

- Verify I/O using HDF5/IOD end-to-end integrity checks
    - Direct (no function shipping)
    - CN/ION configuration
        - Via the Burst Buffer
        - Bypassing the Burst Buffer
    - Producer/consumer

        - Verify progress of reader and writer and transactional consistency at reader
- Verify client failure
    - Check transactional consistency after random compute node and I/O node failures
- Verify server failover
    - Check transparent handling of server restart and failover

Acceptance Criteria

Acceptance occurs when the mandatory requirements listed above have been implemented and the unit and integration tests listed above pass.