Date: Feb 22, 2013	# High Level Design – IOD # 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
Feb 1, 2013	V0.1, initial version.	Xuezhao Liu
Feb 22, 2013	V0.2, some revisions about array object layout, resharding, transaction semantics etc, based on comments from John Bent.	
Feb 22, 2013	V0.3, Small editing.	John Bent
Mar 2, 2013	V0.31, some revisions based on comments from Johann and Ruth.	

1. Introduction

I/O dispatcher (IOD) runs over the I/O nodes (IONs) which are equipped with persistent solid-state burst buffers, together with other layers' innovations it will change both the hardware storage tiering and software I/O stack, tends to sustain both the scalability and performance requirements for extreme scale HPC storage system. IOD absorbs application's I/O and buffers it on local SSD – stores raw data based on PLFS[2] and stores metadata based on KV-store[3], IOD further migrates/pre-fetches the data to/from central storage (DAOS) by application's demand. IOD handles the impedance mismatch between the smooth streaming I/O required for efficient backend disk utilization and the bursty, fragmented and misaligned I/O that frontend extreme scale applications will produce.

Four main characteristics of IOD are: object storage, transactional, semantics aware, and asynchronous.

- Object storage. IOD discards traditional POSIX semantics and maps complex science data models to container and objects, provides direct access to underlying storage objects to avoid lock contention, allows applications can choose the degree of parallelism related to access needs.
- Transactional. IOD provides transaction which ensures a group of operations be executed across an arbitrary set of processes within a single parallel job be applied atomically – i.e. all or none will success. It can be used to guarantee the integrity and isolation of the stored science data models.
- Semantics aware. IOD can understand the structure of science data models based on which can do layout resharding according user's demand to make the data be affinitive to computing. IOD leverages the fast network interconnect between the IONs and can do MPI communications between them for data shuffling and synchronization. IOD provides APIs to control burst buffer's pre-fetch from or flush to central backend storage with optimized layout, as well as multi-format replicas with analysis workload preferred layout.
- Asynchronous. IOD strives for asynchrony to allow user can build fully non-blocking applications through which further improves parallelism by overlapping computing and I/O. One IOD API's success return just means the request has been submitted to IOD, a related completion event can be polled by user when it finally finishes executing.

2. Definitions

IOD

I/O dispatcher[1]

CN

Computing node

ION

I/O node

BB

Burst buffer

DAOS

Distributed Application Object Storage[4]

Shard

There is a "shard" concept at both IOD layer and DAOS layer. At DAOS layer it means the virtual storage target of DAOS container, similar with current Lustre's OST. At IOD layer it stands for the split data pieces across multiple storage devices comprising an IOD object.

PLFS

Parallel Log-structured File System[2]

Function shipper

An I/O forwarding layer that ships function calls from CN to ION[6]. It is client-server model that client runs on CN and server on ION.

Process group

The "process group" in this document stands for client side application's process group. It is a collection of n processes. Each process in the group is assigned a rank between 0 and n-1.

IOD Container

- IOD layer's data representation which is 1:1 mapped to HDF5 file and DAOS container.
- Can contain any number of objects inside which stores user's metadata and data.
- ION's local storage can be POSIX in which case one IOD container will correspond to one special directory at every ION. The directory path is IOD container's path and can be visible by POSIX namespace.

IOD Object

- IOD layer's data representation which is 1:1 mapped to HDF5 object but not 1:1 mapped to DAOS object. One IOD object can be sharded across multiple DAOS objects.
- Three types: array, blob and KV.
    - Array and blob object are used to store user's structured and unstructured raw data respectively. They are stored via PLFS and can be sharded across multiple IONs.

- KV object is for storing user's metadata. It uses parallel key-value stores for data storing.

Need comparison of this metadata management and Ceph's. Ceph has hierarchical, global name space that automatically balances.

3. Changes from Solution Architecture

Only one change from IOD SA document is about the object versioning. Based on the relevant discussions, we decided what we need is container-level versioning which can be implemented by container level snapshot.

4. Specification

4.1 High level system view

High-level system view is depicted as figure 1. The possible server-side VOL plugin is not shown on the diagram.

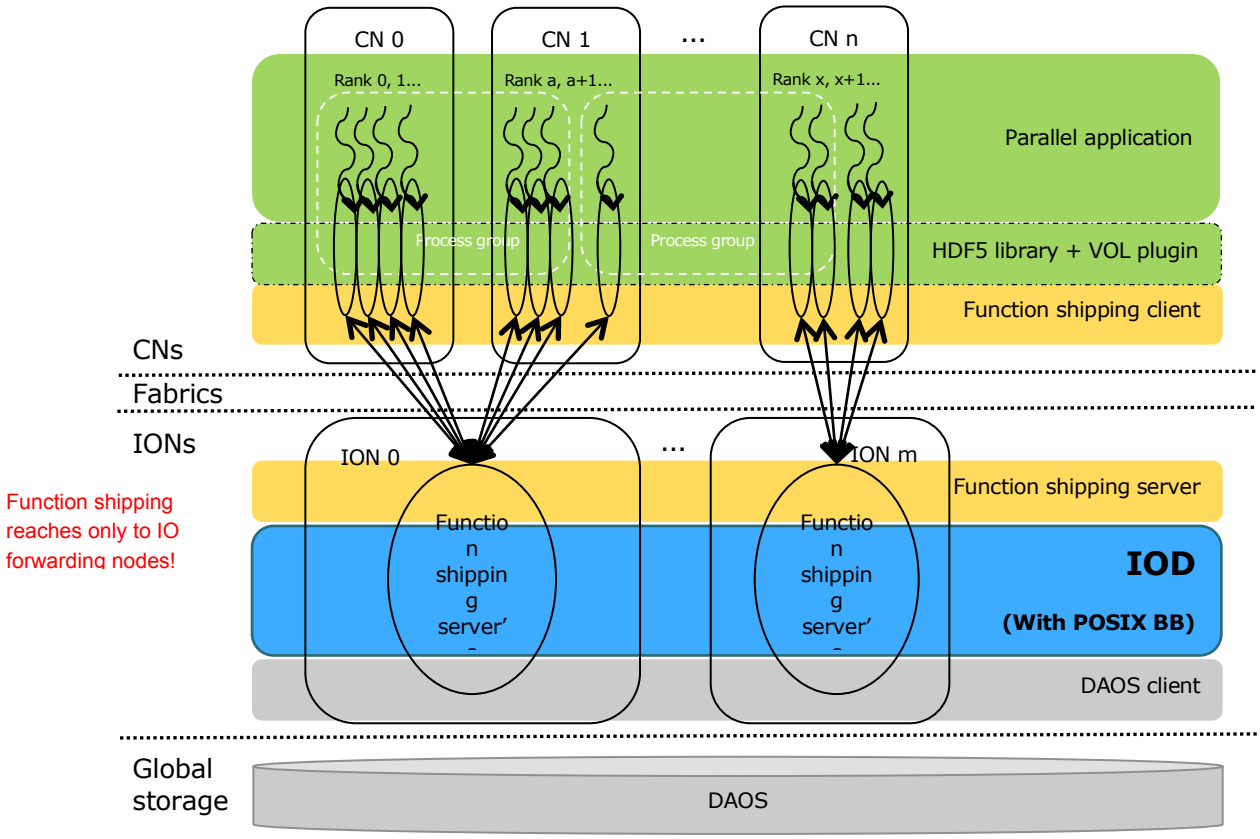

Figure 1 – high level system view

IOD is a library which provides I/O services to upper-layers. IOD doesn't has itself process space, instead it is linked into application's process space. In the case there is function shipper between CNs and IONs the IOD is linked into function shipping server's process space. IOD will create some service threads within caller's process space. Every ellipse in figure 1 corresponds to one process which has independent process space.

The function shipper forwards VOL function calls from CNs to IONs. The function shipper server is a MPI program runs over IONs cluster, it calls IOD's initialization routine and passes in the MPI communicator by which IOD can do kinds of MPI communications. This forwarding architecture offloads I/O functionalities from CN to ION, improves system's performance and scalability, but does add some extra complexities to both function shipper and IOD:

- Function shipper only forwards CN ranks' I/O call 1:1 to ION, and one function shipping server provides service to many CN ranks. This may cause IOD to receive some duplicate function calls. For example all CN ranks call IOD object open, one function shipper server possibly will open the same object multiple times, same for object close. So IOD will need to maintain an open ref-count for safe open/ close. The object create is more tricky, to avoid possible race, IOD restricts the object create can only be called once by one CN rank, that rank can get back an object ID and can share it to other ranks for further open.
- Function shipper isolates application's process topology (process group information) from CNs to IONs. For example, CN ranks can have multiple process groups which possibly will participate in transaction independently. But IOD cannot know which set of CN ranks belong to which process group. It may be too difficult for function shipping server to create appropriate process group based on CN ranks' process topology and how CN ranks are connected with it. This brings some difficulties for transaction status/consistency synchronization, details in section 4.5.2.
- The event allocated and polled at function shipper server can be polled only within that address space as there is no globally meaningful event id; just an event instance within the caller's address space.

This might be a consequence of designing IOD as a library that is linked to application space

Restart of shipped function when server crashes?
Shipped function needs to be idempotent!

4.2 IOD sub-modules overview

Figure 2 shows the high-level overview of IOD's sub-modules.

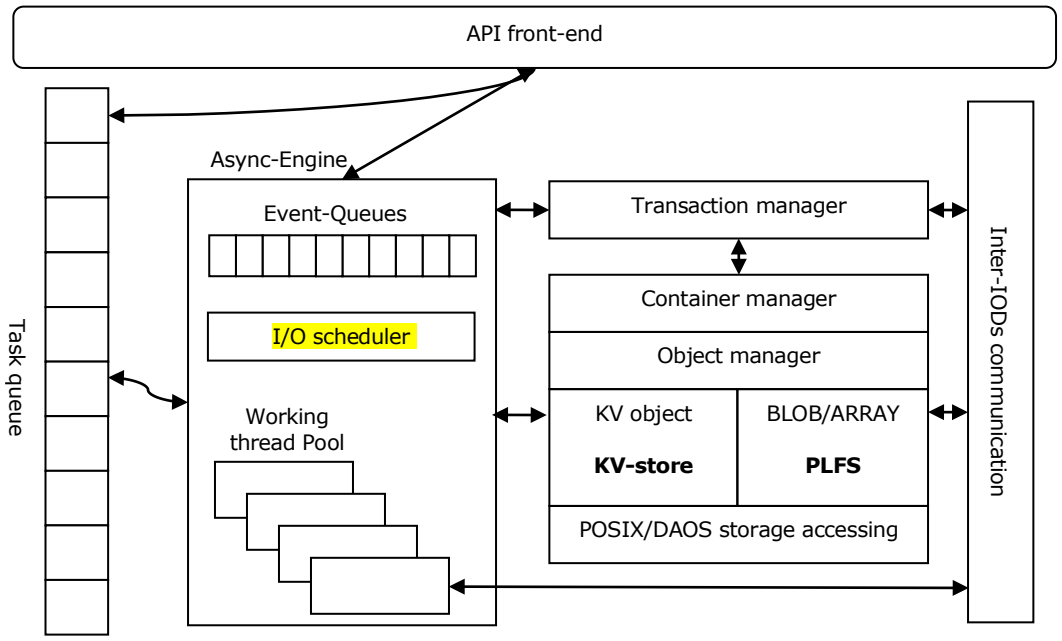

Figure 2 – IOD sub-modules overview

The API front-end provides kinds of IOD APIs to upper layer. When one API is called by upper layer, IOD inserts an appropriate task to the internal task queue and bonds it to an asynchronous completion event. The task queue is not exactly one single queue or linked list, instead it is a set of task lists which may belong to container or objects, and bonds to transaction. The event can be allocated and passed in by caller.

All tasks are executed by async-engine which is the center for executing and progressing all asynchronous operations. Three main components for the async-engine are thread pool, I/O scheduler, and events and the event queue manager. The thread pool is the executing unit of async-operations, it needs to be initialized inside iod_initialize() with configurable threads number. The I/O scheduler progresses asynchronous operations, it also has chance to do I/O optimizations such as stream transformation/merging. The event and event queue manager maintains the relationship between event and pending tasks.

The transaction manager provides transaction semantics. One specific transaction's status is managed by the transaction manager which is selected by hashing transaction ID. All transaction's final status is tracked by container manager which is selected by hashing container path name.

One IOD container corresponds to one POSIX directory on every ION, and has a backend DAOS container for data migration and central storage. It can have many objects inside one container. IOD's local object storage is based on KV-store and PLFS. IOD have a DAOS accessing layer for migrating data and synchronizing IOD container's operations (for example open, create, unlink etc.) to DAOS.

The inter-IODs communication layer is another important module used for communications between IODs. The communications include container/transaction status query/synchronization, possible data shuffling/movement, and internal collective communications etc. There are possibilities that IODs need to do internal collective communication such as for global PLFS index building, object creating etc. IOD can build

an internal spanning tree based on which to do the asynchronous collective communications; this optimization is a potential roadmap addition in future quarters. IOD will have special threads listening on communication request with some pre-defined messaging formats. The communication is based on MPI mechanism, so IOD's caller (function shipping server in our scenario) should be a MPI program and should create the MPI communicator of all IODs. However the function shipping server needs only create the MPI communicator and pass it to IOD when calling iod_initialize(), no other IOD functions take an MPIC communicator as a parameter.

IOD has debug/diagnose supporting layer which is not depicted at above diagram.

4.3 Object storage

4.3.1 Three object types – KV, ARRAY and BLOB

IOD provides three types of object abstractions.

- KV[3]

  It is used to store user metadata, such as HDF5 group/attribute/link etc. IOD provides KV store based on MDHIM (Multi-Dimensional Hierarchical Indexing Middleware). IOD exports the KV-store to upper layer through a set of KV-APIs to allow caller to create/open/set/get/list/close/unlink the KV object's content. Besides the KV objects created by upper layer, IOD will create and store some internal KV objects for its own metadata for other object (for example the data space and layout of array object) and other objects' scratchpad data.

  IOD can use other internal KV-stores to store some internal metadata such as the object list within container, and the mapping between IOD object and DAOS objects etc.

- Array

  Array is a multi-dimensional data array which corresponds to HDF5 dataset. The array object has spatial structure; IOD can be structure aware based on which to make the semantic resharding be possible. IOD array object can be one dimension extendable, more precise it can be extended only along the first dimension to allow IOD to calculate the determined logical address space and with user preferred layout. IOD supports similar concepts as HDF5 dataset's contiguous layout and chunked layout[8] to make it smooth to bridge to HDF5 users' common usage.

- Blob

  A data blob is simply an unstructured stream of bytes corresponding to HDF5 committed data type, and is semantically identical to a standard POSIX file with the addition of transactional semantics

Both the array and blob objects are stored by PLFS. IOD uses a PLFS logical file to represent the array and blob object; one PLFS logical file is implemented with a PLFS-container which is a set of POSIX directories across IONs. In ION's local storage, one IOD container is a POSIX directory, every array or blob object is a sub-directory inside the parent container's directory. The array or blob's data is stored in PLFS logging file. By leveraging the log-structure of PLFS, IOD provides fast writing speed based on local SSD. User can migrate object's data to DAOS, for the migration IOD will rebuild the consistent view of the object based on PLFS' indexing mechanism and do an optimized layout

placement to DAOS. Essentially the array and blob objects will be logging on IONs and flattening on DAOS. The details will be introduced at section 4.4.

4.3.2 Object mapping between HDF5/IOD/DAOS layers

The below table is a mapping between high-level HDF5 objects and IOD abstractions.

HDF5 Object	**IOD abstraction**
H5File	Container
H5Group	KV object
H5DataType H5DataSpaces H5Attribute H5Properties H5Reference H5Link	KV pair in KV object
H5Dataset	Array object
H5CommittedDatatype	Blob object

Figure 3 gives out a more detailed example to illustrate the object mapping at different layers. The example includes HDF5 group/link/dataset/committed-data-type/attribute objects and how they are mapped and stored on both IOD and DAOS (after migration).

In the example, there are 3 H5Group objects which are implemented by IOD KV objects, one H5Dataset and H5CommittedDatatype objects which can be implemented by IOD array object and blob object respectively. For the root group, the 2 H5Attribute objects "ID=XX, verified=y" can be implemented by KV-pairs. The first created IOD KV object is root group. After creating the root group, which is always assigned IOD object ID 1, user can create other kinds of objects and use H5Link to establish the relationship between them, for example the "visualizations" link points to root group's sub-group and a further link "view1" points to the H5Dataset object.

User can create similar "child-parent" relationship between some objects, for example to store the H5Attribute objects belong to H5Dataset or H5CommittedDatatype object ("resolution=z" and "version=3" in the example). Note that the child-parent relationship (the child KV objects on figure 3) can be implemented by the "scratchpad" of parent object. IOD provides fixed length (32 bytes for example) storage as a scratchpad which can be associated with any type of IOD object. User can store the child object ID inside that scratchpad. IOD provides interface to get/set the scratch. This is similar to POSIX extended attributes. IOD will internally store all objects' scratchpad using one special KV object. Because the IOD KV objects are, of course, also transactional, these scratchpads will be as well. The "ID=XX" and "Verified=Y" attributes in figure 3 possibly also will be implemented by a separate KV object and store the object ID in root group's scratchpad.

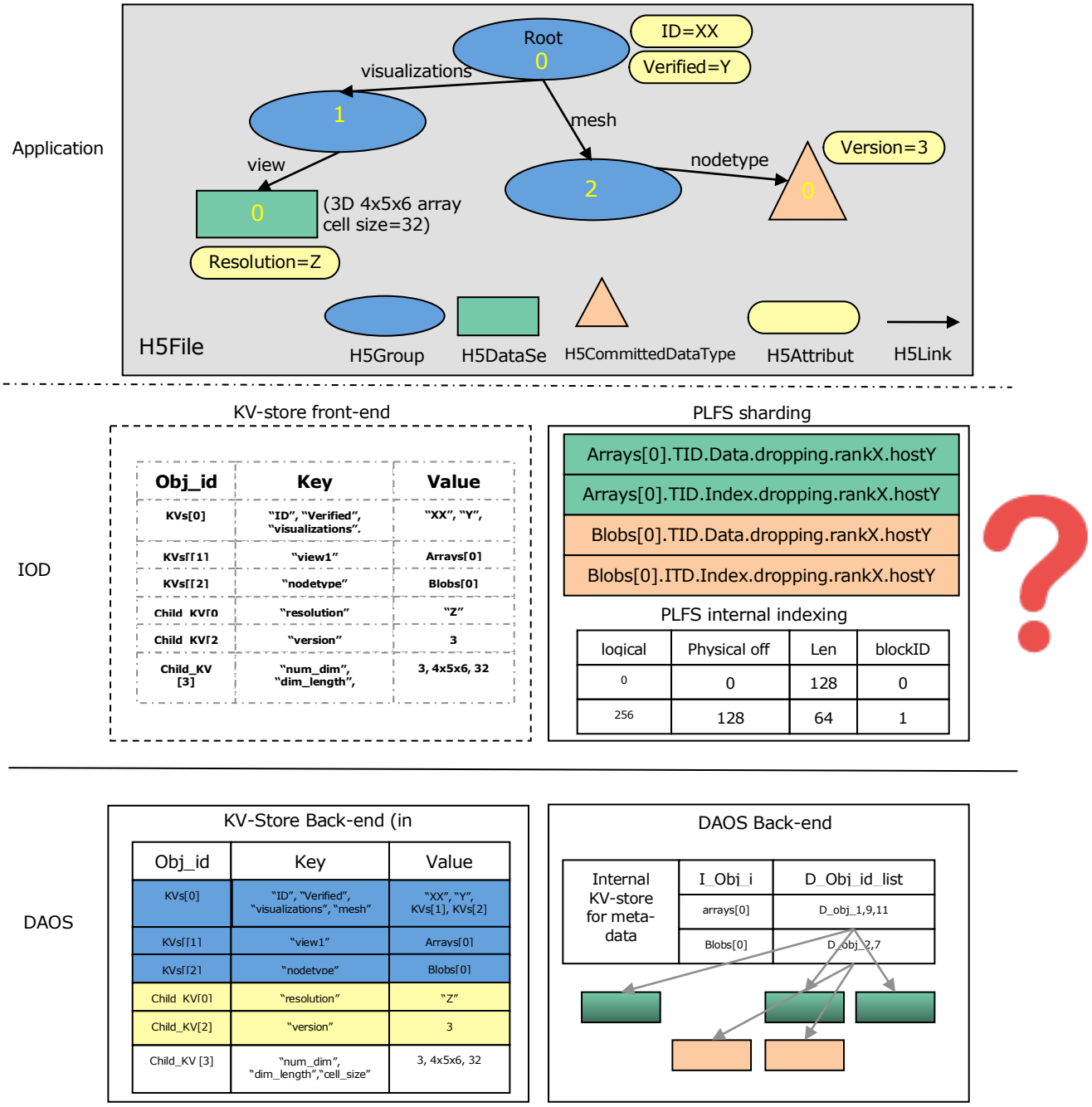

Figure 3 – an example object mapping

For array's data space information such as dimensions, length, cell size etc, IOD will create a corresponding internal KV object ("3D 4x5x6, cell_size=32" as above example) and associate it with the array object. The array or blob object's data can be sharded to multiple IONs' local SSD by PLFS logic. After migration IOD will create a set of DAOS objects to store the data into DAOS, and IOD will store its own metadata such as the mapping between IOD object and DAOS object list using an internal KV-store (the

"Internal KV-store for metadata" shown on figure 3) which uses another set of DAOS objects as its backend storage.

4.4 Layout, data migration and reorganization

One of the most important and complicate aspect of IOD is the data layout, migration and reorganization. For this, IOD's has three major goals: a) be semantic/structure aware and provide flexible interfaces to allow callers to control the data layout, sharding granularity and placement across DAOS shards and IONs; b) support users' traditional usage model of HDF5 data model and make it well suit for underlying storage; c) reduce the metadata needed for logical/physical mapping, sharding placement, etc.

There are mainly two kinds of mappings that IOD needs to maintain:

1) Maps the logical spatial space of multi-dimensional data set to physical storage object's one-dimensional address space – we call this "layout",
2) Maps the flattened object address space of the resulting layout to underlying storage: a set of DAOS shards or IONs – we call this "sharding" or "resharding".

4.4.1 For IOD array object

4.4.1.1 One dimension extendable data array

In FastForward project, we decided to support HDF5 multiple dimensional dataset with at most one extendable dimension, i.e. the dataset has either all fixed dimension lengths or can be extended by one dimension.

To support this requirement, IOD's data array object is one dimension extendable multi-dimensional array. To allow IOD to calculate a determined address space for that data array, IOD restricts the array object to be extended only along the first dimension and all other dimensions are with fixed length. It is like HDF5 original "external dataset" except:

- The HDF5 original external dataset's content must be stored by external files. IOD will store the data in array object within the container.
- The HDF5 original external dataset only can use contiguous layout with fixed dimension layout sequence, whereas IOD provides both contiguous layout with changeable dimension layout sequence and chunked layout supporting. Details in next sub-section.

4.4.1.2 Array's layout, migration and resharding

Layout mapping

The original HDF5 dataset have two kinds of layout – contiguous layout and chunked layout[9]. The below illustrates HDF5 original semantics of layout and IOD's extensions/changes to it:

- contiguous layout
    - The HDF5 original contiguous layout simply flattens the dataset in a way similar to how arrays are stored in memory, serializing the entire dataset into a monolithic block on disk which maps directly to a memory buffer the size of the dataset. And the dimension layout sequence is fixed – the first dimension is the slowest changing dimension and the higher dimensions are faster changing, the last dimension is the fastest changing on disk. For example a three dimensional dataset A, then the first element on disk

would be A[0][0][0], the second A[0][0][1], the third A[0][0][2], and so on. If the application read by the same layout sequence such as from A[0][0][8] to A[0][0][520] the underneath will be well behaved as sequential read from disk. But in the case if application wants to read from A[8][0][0] to A[520][0][0] it will have worse performance because the under layer will do either lots of random reads or read back large un-needed blocks and sieving the wanted items.

- IOD supports contiguous layout with extended capability that allows user can change the dimension sequence of layout. For a 3-dimensional data array with X-Y-Z axes, user can control how to flatten those axes on object space – either X-Y-Z sequence or Z-Y-X sequence etc. Same as above example, if user want to read from A[8][0][0] to A[520][0][0] it can set the logical first dimension as the physical last dimension for storing to disk. This is very useful when the simulation program and analysis program have different accessing pattern. The simulation program may write dataset by X-Y-Z axes to disk, later the analysis program wants to read it by Y-Z-X axes then it can pre-fetch the dataset from DAOS and change the layout to Y-Z-X axes and shard it to a set of IONs with it preferred way. We call this semantic resharding.

Drepl fits here!

- Chunked layout
  - HDF5 original chunked datasets are split into multiple chunks which are all stored separately in the file. The chunks can be stored in any order and any position within the HDF5 file. Chunks can then be read and written individually, improving performance when operating on a subset of the dataset. HDF5's chunk filter, chunk cache etc are applied on chunk granularity. HDF5 uses quite complicate and sophisticated logic of B-tree or skip-list[8] to map chunk indices to file offsets for a chunked dataset.
  - IOD's array object supports chunked layout to satisfy HDF5 users' traditional usage, and HDF5 original chunk filter and similar functionality can be smoothly applied on IOD array object with chunked layout. As both IOD layer's PLFS and DAOS' object provide virtual unlimited address space, so IOD can implement the chunk layout and avoid the most complicate things of HDF5 original B-tree or skip-list chunk indexing mechanism. Everything can be calculated by IOD so IOD needs not maintain the indexing to map the chunk indices to file offsets. The chunked layout can only be set when creating the dataset and cannot be changed after create same as original HDF5. The chunk selection is a parameter can be passed in for IOD array object create.

As mentioned above, the layout maps the logical spatial space of a multi-dimensional dataset to physical storage object's one-dimensional address space. Figure 4 gives an example to show the IOD array object's contiguous layout and chunked layout mapping.

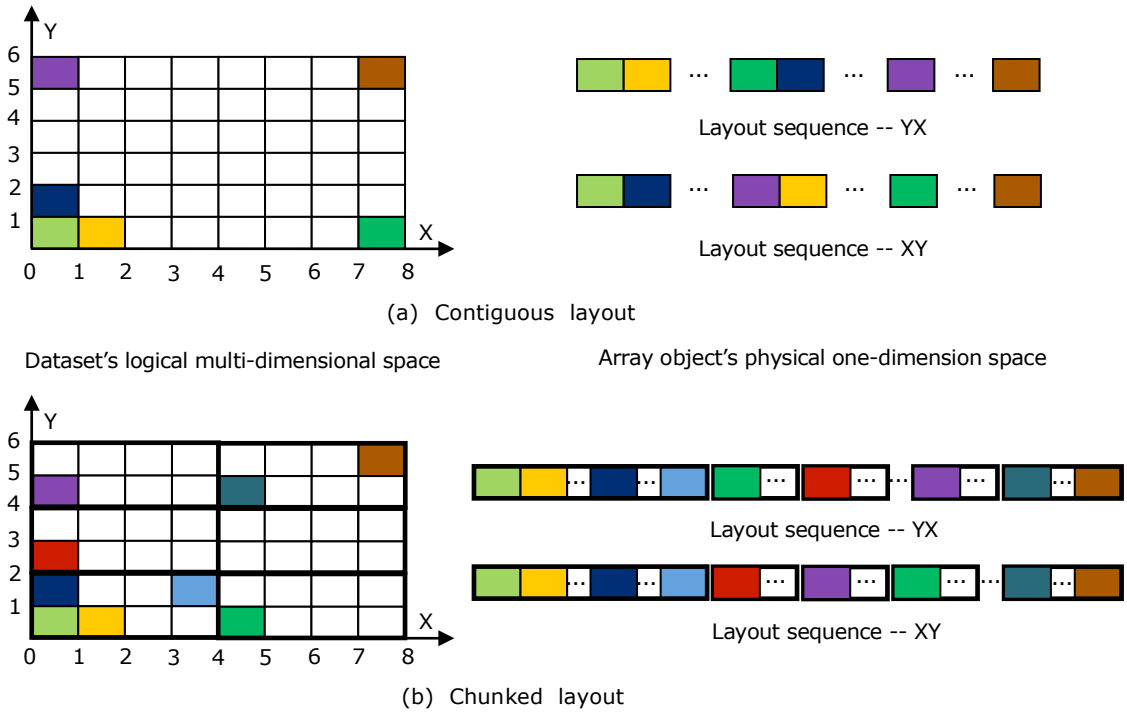

Figure 4 – an example of array's contiguous layout and chunked layout

The example is a 2D 6x8 array. Part (a) shows the contiguous layout mapping, and IOD allows users to change the layout mapping for example from Y-X sequence to X-Y sequence which causes IOD to rearrange the physical layout of data. The dataset's logical structure is not changed – it is a 2D 6x8 array regardless of the physical mapping. The layout just determines the mapping from logical space to IOD array object's storage space.

Part (b) shows the chunked layout that sets 2x4 chunk size, so the array will have 6 chunks in total. For chunked layout, IOD will do write/read with the granularity of chunk. The array's original dimensions will be chunked to smaller chunked dimensions, for the above example the chunked dimension is 2D 3x2. User can further select the layout sequence of the chunked dimension for example Y-X or X-Y. But within each chunk, IOD will not change the physical layout – it is always same as logical dimension sequence, Y-X in the above example. A note here is sometimes the number of dimensions will be changed after setting the chunked layout, for example in the above example the 2D 6x8 array will became 1D chunked array with only 2 chunks if setting chunk size as 6x4, so user needs not set the chunked dimension sequence in this case.

For the extendable multiple dimensional array which can be extended along the first dimension, user cannot change the first dimension's sequence, the extendable first dimension is always the slowest changing dimension, i.e. the logical extendable first dimension must also be the first physical dimension for both contiguous and chunked layout.

IOD will first implement contiguous layout. The chunked layout is only optional and will be implemented when the schedule permits.

Migration and resharding

By selecting appropriate layout type and dimension sequence, user can control the layout mapping from dataset's logical space to underlying object storage space. Besides this, IOD provides interface to control the data migration and resharding: user can control how to split the flattened object address space into multiple pieces (shards) and place the shards across a set of DAOS shards or IONs. The "migration" stands for data movement direction which can be BB to BB, BB to DAOS, DAOS to BB or DAOS to DAOS. The "sharding/resharding" stands for controlling the shard granularity and placement among storage targets (DAOS shards or IONs' local SSD). The sharding granularity is multiple dataset items for contiguous layout, or multiple chunks for chunked layout. It does not make sense to break the unit of item or chunk for data resharding.

For migration from BB to DAOS, IOD will not automatically free BB's storage space when the migration is done. Similarly, when pre-fetching from DAOS to BB, IOD will not free/punch DAOS object. When migration target location is BB, IOD will generate a new TID which user can get for reading the data. The "new TID" is just adding special flags on the original TID; since the TID is 64 bits, IOD will reserve 8 for its own metadata about the TID such as whether it is a replica

The data migration is always for entire transaction. User should control the reasonable transaction granularity for it. When migrating a transaction, the below parameters/behavior can be designated by caller:

Needs to be integrated with function shipping: once data is on the move and being re-layouted, why not filter, index, encode, compress at the

1) Migrate direction (BB to BB, BB to DAOS, DAOS to BB or DAOS to DAOS), IOD provides different APIs for it.
2) Target layout – the dimension sequence or chunked dimension sequence (X-Y-Z or Z-Y-X for example), if no layout is designated then IOD will use the previously set valid layout, the default layout before any special setting is same physical dimension sequence as logical dimension.
3) Number of storage targets – DAOS shards or IONs. User can set it as zero in which case IOD will use all available targets.
4) Sharding granularity – how many dataset items for contiguous layout, or how many chunks for chunked layout. All split shards will be round-robin placed on those storage targets as determined by 3). User can set it as zero in which case IOD will select a reasonable granularity (maybe 4M bytes or other value).

When object is migrated to DAOS, IOD will create some DAOS objects (one object for every DAOS shard) for storing that IOD object's data. IOD will maintain related metadata to store the mapping from IOD object to DAOS objects. IOD can use an internal KV-store to store those internal metadata, the below metadata need to be maintained by IOD:

- The list of IOD objects within container
- The mapping from each IOD object to DAOS objects
- The layout, sharding placement of the IOD object
- The valid ranges of object's physical address space within the IOD object

IOD will shard one IOD object to a set of DAOS objects, and can keep the exactly same address space between IOD object to DAOS objects because DAOS object's address space is virtual and unlimited, so within every DAOS object there is possibly many big holes inside its address space – IOD combines all those DAOS objects' address space for one IOD object's address space. For storing on DAOS, IOD will consider DAOS' shard

property to select appropriate shards. For example some DAOS shards are optimized for bandwidth, IOD will use these shards to store bulk raw data; and some shards are optimized for IOPS, IOD can use for storing metadata.

User can control kinds of layout and sharding policies, but it should be mainly used for analysis program when pre-fetching from DAOS to BB, or resharding from BB to BB. It is not recommended to frequently change the layout especially for changing layout on DAOS, because the underneath VOSD[10] needs a relative fixed object layout to detect the changed and unchanged extents/ranges. Changing layout on DAOS will cause basically all data to be re-read and then re-written to DAOS.

4.4.2 For IOD blob object

The IOD blob object is much simpler compared to array object. Blob size can be grown by appending to it, and user needs not setting/changing the layout as it is only one-dimension. For blob's migration, user only needs to decide the migration direction, number of storage targets and sharding granularity. The blob's sharding granularity is byte as blob is just a bytes stream.

4.4.3 Multi-format replicas

The replication is merely a special case of migration. Same as other migrations, user can set the layout, number of targets, and sharding granularity. The only difference is the replica is a kind of duplicate data in logical concept, but possibly with different physical layout or shards placement. The replication is also in the granularity of transaction.

User can select to do the replication only for one TID's incremental data, or for all data till this TID:

- For this TID only replication, IOD will only read the data that changed/appended within this TID and do the resharding according other parameters setting. As the former TIDs' data is not read back and possibly be migrated to DAOS and be flattened/merged by DAOS, so IOD cannot ensure the data consistency if user wants to read the data that is not changed/appended within this TID, IOD just returns zero for un-existed ranges. The common usage for this kind of replication is user clearly know that it only needs to read the changed/appended data within this TID.
- For all data till this TID replication, IOD will read/merge all data including this TID's changed/appended parts and all former TIDs' data. The old TIDs' data will be overwritten by new TID's data for the overlapped parts. By this method, the data consistency can be ensured, but will lead to read/replicate large amount of data.

Infrastructure for proposal with Chris Brislawn: our proposa generalizes reshardin to distributed encoding.

After the success of the replication, user can get back a TID, with a special flag set in the IOD reserved bits, which indicates it is a replica. User should explicitly purge the replication's data to free BB's storage space. Multi-format replication on array objects can be referred to as semantic resharding. KV stores may also be semantically resharded by specifying how to partition (and repartition) the key-ranges across the IOD's.

4.5 Transaction

IOD provides transaction semantic to upper layer with the following properties (similar as DAOS[4]):

- **Atomic writes** – either all writes in a transaction are applied or none of them are.
- **Commutative writes** – concurrent writes are effectively applied in TID order, not time order.
- **Consistent reads** – all reads in a transaction may "see" the same version data even in the presence of concurrent writers.
- **Multiple objects** – any number of DAOS objects within one container may be written in the same transaction. IOD transaction is at container level.
- **Multiple threads** – any number of threads and/or processes may participate in the same transaction.

Every transaction has an identifier as **TID**. All IOD I/O operations include a TID parameter.

Transactions are very light-weight. So we don't need to combine reads and writes into one transaction.

- Read specifies a TID to ensure multiple reads "see" a consistent version of the data. Write (including unlink) specifies a TID to ensure multiple writes are applied atomically. IOD does not allow user to read and write to one TID at the same time.
- IOD TID is not 1:1 mapped to DAOS epoch as some transactions may be only buffered at BB. When transaction being migrated and so persistent on DAOS, IOD will use a DAOS epoch number equal to TID.
- TID is a 64 bits value. IOD reserves the highest 8 bits for internal using, for example as replica flags etc.
- For every container, IOD maintains:
    - lowest_durable TID. It is the lowest TID which had been migrated to DAOS and hold a referenced DAOS HCE snapshot.
    - latest_rdable TID. It is the latest (highest) TID which is readable on BB, it possibly has not or has been migrated to DAOS.
    - latest_wrting TID. It is the latest (highest) TID which is started for writing.

4.5.1 Transaction status

One transaction can have 7 different statuses:

- Invalid – this TID has not been started
- Started – user has started the transaction, but there are still participators have not finished it
- Finished – all participators have finished but one or more earlier transactions are not finished
- Readable – all writers have finished and all earlier transactions are also finished or aborted
- Aborted – a participator aborted it, for writing transaction it's data will be discarded by IOD
- Durable – readable and has been migrated and so be persistent on DAOS.
- Stale – previously durable or readable TID which is exceeded by lowest DAOS HCP snapshot, this can happen when user does not purge it in time. In this case

IOD cannot ensure the consistent readability of it. IOD will not automatically purge stale TID, when later user reads it IOD just returns "-ESTALE" and user needs to explicitly purge it and read by a higher TID.

The "finished" and "stale" can be just IOD internally managed statuses. The state diagram is given out as figure 5.

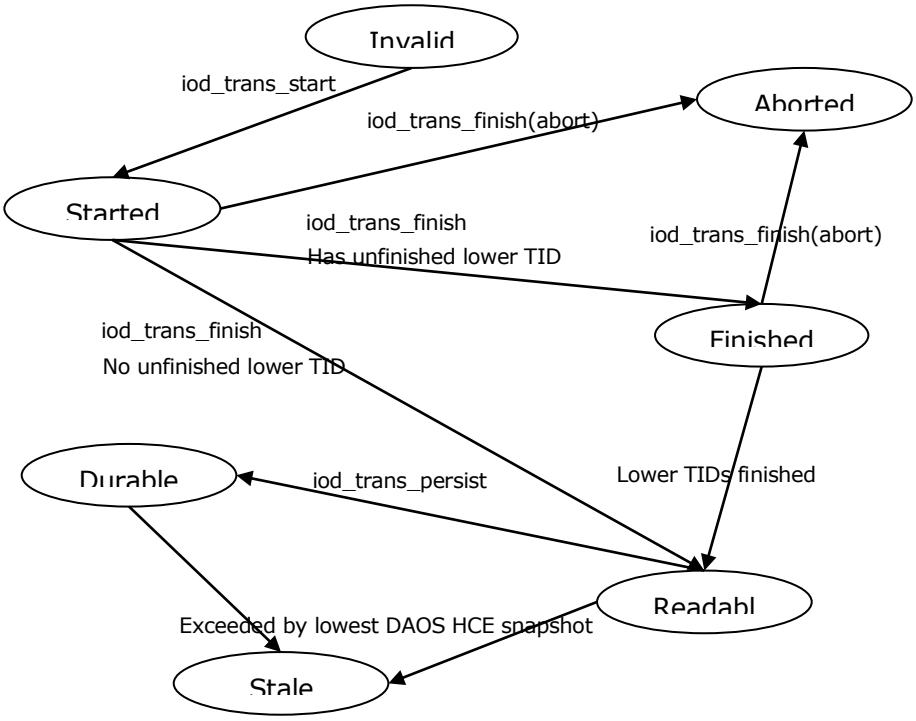

Figure 5 – transaction state diagram

4.5.2 Participate in transaction

TID selection

To participate in a transaction, user should first get an appropriate TID by either of the below two methods:

- Can call iod_container_query_tids to query this container's TID status.
    - The TID higher than latest_wrting is writable, common use case is to start TID (latest_wrting + 1).
    - For read, the TIDs between lowest_durable and latest_rdable can be readable but possibly there are some invalid or aborted TIDs within the range.
- User can pass in "IOD_TID_UNKNOWN" if it does not want to do the query. IOD will select an appropriate TID and returns to user.
    - For writing, IOD will select (latest_wrting + 1) and return to user.
    - For reading, user can pass in hints to say "I want to read the lowest readable TID" or "I want to read the latest readable TID" as there might be multiple readable TIDs between lowest_durable and latest_rdable TID.

Transaction status synchronization

For every TID, IOD will hash it to one IOD instance as its transaction leader which corresponds to manage/track the transaction status. Besides this, the container manager will track all transactions' final status as well as TID allocation. By this method the transaction leader can partake some workloads from container leader to avoid the container leader being overloaded.

Any number of participators (CN ranks) can participate in transaction, so IOD needs to have an approach to determine the final transaction status among all participators. IOD supports two kinds of transaction status synchronization and consistency ensuring mechanism:

1) Application does the transaction status synchronization.

   Application ranks will need to select one transaction leader. The leader rank starts and finishes/slips the TID, other ranks can participate in the TID after leader having started it, and the leader should ensure all other ranks have finished I/O operations within this TID before it finishes/slips this TID. As application ranks are process topology aware, they can do fast group collective communication (with possible high-efficient special hardware supporting features) for the transaction status synchronization.

   This is the method similar as DAOS epoch's requirement.

2) IOD internally does the transaction status synchronization.

   Application ranks only need to start and finish/slip this TID separately and independently. For this method, user needs to pass in the number of participators (num_ranks) for this transaction. IOD needs this number to track whether or not all participators have finished this transaction. The "num_ranks" is number of CN-side ranks as function shipper 1:1 forwards/translates I/O calls from CN to ION. IOD will need to do lots of internal P2P message passing for transaction status synchronization. As IOD does not know CN ranks' process group information and function shipping server does not create appropriate process group based on CN ranks' process topology, so IOD cannot use group collective communication. When the number of participators is large, the status synchronizations will introduce considerable overhead and latency at IOD layer.

Why?

   A possible optimization exists if IOD can know that this TID is for all CN ranks – we can call it as global transaction. For global transaction, IOD can use the global communicator across all IODs to do similar collective communication by building collective spanning tree to reduce the lots P2P message passing. For this optimization, IOD needs to know two extra parameters: 1) total number of CN ranks and 2) the number of CN ranks which are connected to this IOD. User can pass in these two parameters when calling iod_initialize. If application can create dynamic processes, then user should re-call iod_initialize when dynamic processes are added. This possibly is a too high requirement to upper layer, so basically IOD can only use lots of P2P message passing for transaction status synchronization.

Motivation for project with Jay.

   However, even the higher cost of the multiple P2P messages can be mostly avoided when applications are not extremely asynchronous. Each IOD will count the number of open references for each TID and only communicate with the transaction leader when it sees the TID for the first time and then again when the reference count goes to zero. In the best case, this will be two messages

between each IOD and the transaction leader. In the worst case, when no two processes sharing an IOD ever have the transaction open simultaneously, then each IOD will communicate with the transaction leader once to start the transaction and then again for every process participating in that transaction on that IOD. It should be noted that this extreme worse case is expected to be extremely unlikely as this would mean that the asynchrony of the application would be so large that processes would be 1000's of transactions removed from each other. In such an extreme case, the application is encouraged to do its own monitoring of transaction completion as described in method 1.

Commonly application should use above method 1) to avoid too much overhead and latency at IOD layer.

Besides the possible performance difference, the semantics/usage of method 1) and method 2) have some differences which need to be understood by caller:

- By method 2), application cannot use IOD_TID_UNKNOWN to start a transaction. Why?
- By method 1), application's different process groups can independently participate in same or different TID at the same time; by method 2) different process groups cannot participate in the same TID at the same time.

Start and finish, slip

All processes that want to participate in the TID need to call iod_trans_start() with same parameters of TID, number of writers etc. The "num_ranks" is needed for IOD to track that TID's status, zero value means application ensures the status synchronization. Every participator needs to call iod_trans_finish() to mark the finish of transaction. The iod_trans_finish() can be an asynchronous operation which immediately returns after being submitted to IOD, but the completion of the async-event will be stalled until:

See Jay's concern

1) All participators of the TID have called iod_trans_finish(), and
2) If for writing, all former TIDs become readable/durable or aborted as IOD transactions are ordered.

The slip is like a combination of "finish(old TID) and start(new TID)". The internal ref-count will also be slipped from old TID to new TID.

Abort

The iod_trans_finish() can carry an "abort" parameter to abort a transaction, any writer of that transaction can abort it. After successful abort, all writings/updating within this transaction are discarded, IOD will roll back to the status before this TID started.

For writing transaction, user can select two different kinds of abort semantics:

User defines dependencies between transactions.

1) Only abort this TID. This means only abort this transaction and will not affect its later transactions. User should select this option if it knows that all higher TIDs have no dependency on this TID.
2) Abort this TID and all higher TIDs. User should select this option if the higher TIDs have dependency on this TID. [This mirrors the abort semantics on DAOS.]

User can only abort a transaction before it becomes readable.

4.5.3 Consistent semantic

IOD transaction semantic is at container level, after one TID becoming readable on BB user can call iod_trans_persist to protect the whole transaction's state to DAOS. As IOD

transactions are in strict order, when user persisting one TID IOD will make all lower readable TIDs and this TID's state be persist. After the completion of persisting, the transaction becomes durable on DAOS. That transaction is still kept on BB, IOD will not automatically purge it. On BB, there might be multiple readable TIDs, higher TIDs' data might have dependencies on lower TIDs. IOD needs to keep those readable TID's consistent readability. To keep the consistent readability, IOD must ensure the lowest_durable TID on DAOS cannot exceed the lowest readable TID on BB. IOD will follow these rules:

1) At beginning of iod_trans_persist, IOD calls daos_epoch_scope_clone() to get a reference on last DAOS HCE (create a HCE snapshot for it on DAOS); at last step of iod_trans_persist, IOD calls daos_epoch_commit(…, sync, …) to force VOSD to commit this TID without any merge.
2) At 1)'s completion, that TID becomes durable on DAOS. IOD calls daos_epoch_scope_clone() to get a reference of that TID (create a HCE snapshot for it on DAOS).
3) When later user purges that TID from BB, IOD will call daos_epoch_slip() to release the reference taken at 2).

User's purging can create some complexities due to automatic flattening on DAOS. For example, TID 6, 7, 8, 9 are readable on BB, the lowest_durable TID is 6 and latest_rdable TID is 9. At this time point user can only purge 6. Later user persists 9 on DAOS which then renders 6 no longer readable on DAOS. Then 7 and 8 may become unreadable even though they still reside on the ION since any data they rely on from 6 is no longer available anywhere in the system. At this point, they are no longer considered readable on IOD but they remain cached in case any reads of 9 may require data from them. The user must explicitly purge them when desired. The API does, of course, allow a list of TID's to be purged in one function call.

4.6 Versioning

There is some change from the original SOW in how we will implement versioning. There is no longer any notion of persist object versioning. This is replaced with temporary views which approximate versioning and persistent container level snapshots.

4.6.1 Using temporary views to approximate object versioning

An approximation of versioning is possible at the IOD layer using transaction id's. Since the user is solely responsible for managing the contents of the burst buffer, they can preserve multiple transactions on the burst buffer. An example of this would be that they can then open handles on these multiple transactions for time-series analysis of the last three state dumps. However, DAOS does automatic flattening of transactions so using transactions as an approximation of versioning on DAOS is more difficult but could potentially be attempted using open epoch handles to temporarily prevent flattening.

4.6.2 Container snapshots

Permanent versions can be created into the namespace with DAOS container snapshots which are a space-efficient, copy-on-write mechanism for entire container snapshots. This is like object versioning as proposed in the original SOW except:

- It is for containers and not single objects. A user wanting snapshots on a single object could, of course, create a container with only a single object.

- The namespace is a bit different. Instead of version ID's on a single entry, this creates multiple entries within the namespace.
- This will initially only be done on the DAOS layer. A user wanting snapshots must first migrate the container, and then snapshot it. In future quarters, we will consider implementing this fully at the ION layer. In the meantime, users wanting versioning at the ION layer can use the transaction scheme as described above and can also migrate and snapshot each of the views thereby achieving both the temporary versioning approximation and the permanent one. This will work for streaming data. However, if a user wants to preserve data on DAOS and will pre-stage it back into IONs, then they can only use the snapshot mechanism since multiple transactional views cannot be persisted on DAOS.

4.7 Data Integrity

IOD API will have parameters for checksums to be passed to and from HDF. Internally, IOD will implement checksums as a third log in addition to its index logs and data logs. If DAOS participates in checksumming then IOD will pass checksums also between it and DAOS. Otherwise, IOD will be solely responsible for these checksums on storage. Partial reads will of course require IOD to read the full chunk, check the integrity, and then create a new checksum for the partial read. HDF will link with IOD and share the checksumming function.

4.8 Asynchronous operation and event

IOD strives for asynchrony to allow user can build fully non-blocking applications. One IOD API's success return just means the request has been submitted to IOD, a related completion event can be polled by user when it finally finishes executing.

For event queue (EQ) and event:

- A queue that contains events inside, user can create event queue at any time and allocate/initialize an event to bond it to one EQ.
- Events are used by all asynchronous IOD APIs. Most IOD APIs are asynchronous (except API to create EQ, initialize event…).
- User can register a callback to the event, so later after that event finishes, the callback will be triggered.
- Event queue (EQ) and events are used for tracking completion event of IOD functions.
    - IOD function can return immediately only means that request has been submitted to IOD but doesn't mean it has completed, the only way to know completion of operation is polling completion event.
    - If caller passes in NULL event pointer then means synchronous call, the caller will be blocked until finish.

IOD's async-engine is the center for executing all asynchronous operations. When an asynchronous request is received, IOD will insert it to an appropriate task list and bond the task with the async-event. The IOD I/O scheduler will pick up the task and use thread to execute it.

4.9 Impact on HDF5 users

- IOD makes some extensions/restrictions to HDF5 dataset:
  - The dataset is one dimension extendable, can only be extended along the first dimension, i.e. all other dimensions must have fixed length.

    This is same as HDF5's original external dataset except that it needs not to be stored by external dataset files. IOD will store it in one array object within the container.
  - IOD adds supporting of changing the layout mapping between logical dimensions to physical dimensions sequence. This is the basic idea for semantic resharding.

    For extendable array, the first logical dimension must also be the first physical dimension – to make the address space be calculable.
- Transaction is the basic unit of data migration/purging/replica. User should control the reasonable transaction granularity.

5. API and Protocol Additions and Changes

The basic IOD data type definitions and all IOD APIs embedded as txt header files as below.

Basic data type definitions

iod_types.h header file embedded (can click to open): iod_types.h

IOD APIs

iod_api.h header file embedded (can click to open): Iod_api.h

6. Open Issues

- Need we support the use case that there is no BB in system, so IOD will need to run directly over DAOS? For this, IOD will do data shuffling with its sibling peers running across the DAOS storage servers and write flattened object data to DAOS objects. IOD will provide transaction semantic directly based on DAOS epoch. It will cause one difference for the semantics of IOD layer's transaction abort: when IOD runs over BB, abort transaction N will not automatically abort N+1; when IOD runs directly on DAOS, abort transaction N will cause N+1 be aborted as in this case IOD's transaction is directly based on DAOS epoch. And in this use case, the TID's readable and durable status is indeed the same, and iod_trans_persist may need to do nothing.
- Can function shipping server filter out some duplicate calls? For example all CN ranks open/close the same object. And can the function shipping server

create appropriate process groups based on the CN ranks' process topology and how CN ranks are connected with server?

7. Risks & Unknowns

The array object's layout (contiguous and chunked), layout changing (dimension sequence which determine the mapping between logical space to physical space), and migration/resharding/pre-fetching/replication/purging with data consistency requirement introduce complicate handlings to IOD, mixed with some possible different transaction/epoch statuses at IOD/DAOS layer in asynchronous environment. This adds some complexities to IOD.

References

[1] John Bent, "IOD solution architecture", Fast forward internal document.

[2] John Bent, etc., "PLFS: A Checkpoint Filesystem for Parallel Applications", in Proceedings of SC09, Nov. 2009.

[3] Zhenhua Zhang, "IOD KV store high level design", Fast forward internal document.

[4] Eric Barton, "DAOS solution architecture", Fast forward internal document.

[5] Zhen Liang, "DAOS API and DAOS POSIX design", Fast forward internal document.

[6] Jerome Soumagne, etc., "Function Shipping Design & Framework Demonstration", Fast forward internal document.

[7] Quincey Koziol, "HDF5 solution architecture", Fast forward internal document.

[8] Quincey Koziol, "Indexing Chunked HDF5 Datasets with One Unlimited Dimension", http://www.hdfgroup.uiuc.edu/RFC/RFCs/HDF5/ReviseChunks/skip_lists/SkipListChunkIndex.html.

[9] HDF5 document, "Chunking in HDF5", http://www.hdfgroup.org/HDF5/doc/Advanced/Chunking/.

[10] Paul Nowoczynski, "VOSD solution architecture", Fast forward internal document.