Date: December 14, 2012	# SOLUTION ARCHITECTURE – LUSTRE RESTRUCTURING AND PROTOCOL CHANGES ## 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
2012-12-14	1.0 Draft for Review	Johann Lombardi, Intel Corporation

Introduction

The purpose of this document is to provide a high-level description of the Lustre core changes required to support the ***DAOS API***. All requirements below that are not explicitly marked as *optional*, are *mandatory* for successful completion of acceptance criteria.

Lustre DAOS Client

DAOS defines a new object-based API fundamentally different from the legacy POSIX interface. To support this new API, we will take advantage of the CLIO modularity to share as much code as possible between the Lustre POSIX and DAOS clients. Some new extensions (epoch support, versioning cache, collective communications, capabilities …) will be added to the bottom CLIO layers to support the DAOS protocol.

1. DAOS Top Layer

The CLIO layering imposes a clear separation between the semantics implemented on top of Lustre objects (e.g. POSIX, Win32, pNFS, …) and the core object management (RPC engine, OSC pages/extents, …).

It is the responsibility of the top layer to define how Lustre objects are going to be used to support the desired semantics. Lustre currently provides three different CLIO top layers:

- the Linux kernel client, namely VVP (stands for VFS, VM, POSIX),
- the liblustre client, namely SLP (stands for Sysio Library POSIX),
- the echo client, used by the Lustre testing sub-system to access Lustre objects directly, bypassing the LOV layer.

The POSIX client stack is represented in figure 1 below.

A new top layer, namely DCL (stands for DAOS CLient), will be developed to implement the DAOS semantic on top of Lustre objects. The DCL layer will be in charge of:

- providing handlers for the DAOS library calls (I/O controls, open/close & read/write) through llite,
- implementing support for containers, shards, DAOS objects and epochs,
- providing a notification mechanism to be used by the DAOS library to implement event queue.
-

Similarly to the echo-client, the DCL layer will sit directly on top of the OSC layer, providing direct access to the Lustre objects via the DAOS API. Figure 2 below provides an overview of the DAOS client stack.

Fig1. Lustre POSIX client

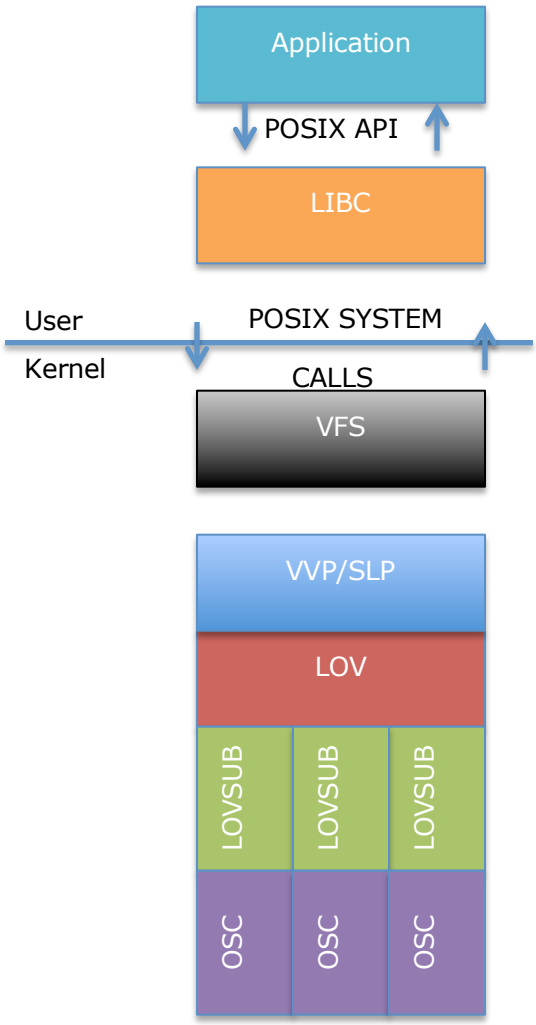
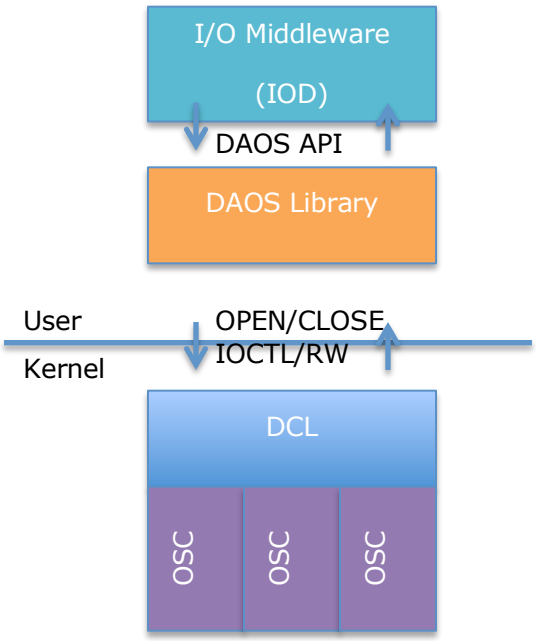


Fig2. Lustre DAOS client



2. OSC Changes

Extensions to the existing OSC layer will be developed to support:

- Global handles & Capabilities: a collective open (see next section) allows a set of clients to open a container and share the open handle. This global handle includes capabilities to enforce security and has to be packed by the OSC layer in RPCs.
- Epochs: all DAOS I/O operations are done in the context of a transaction identified by an epoch number. The Lustre RPC formats should thus be modified to include this epoch number.
- Versioning cache: multiple versions of the same object might have to be maintained in the client cache. This requires modifying the per-object page management data structures (i.e. the per-object radix tree and red-

black tree) to allow different versions of the same object to be accessed. For simplicity, partial pages resulting from misaligned writes may not be cached.

3. Metadata Changes

CLIO is only relevant to object operations. The metadata layers (i.e. MDC & LMV) will be extended to support:

- Containers: as detailed in the next section, a container has a special layout which has to be understood by the client metadata layer. In addition, the metadata protocol needs to be extended to support container creation/destruction and shard creation/deactivation.
  Moreover, a container should be presented as a file in the namespace. It can only be accessed/modified via the DAOS API and all POSIX I/O calls should return EINVAL. One exception is stat/fstat which should return valid time and #blocks attributes.
- Collective opens: global handle allocations are managed by the MDS which then propagates capabilities associated with this global handle to the OSTs via a collective communication. The MDC layer thus has to be modified to support this new type of handle.
- Epoch open/close/commit: epoch lifecycle is orchestrated by the MDS. The metadata protocol will thus be extended to support epoch open & close as well as notification (via a collective communication initiated by the MDS) of epoch commit.

Container & Shard Representation

A DAOS container consists of a set of shards, each shard being placed on a single OST. A shard is assigned an index inside the container and a Lustre FID. The layout (i.e. LOV EA) associated with a container is thus composed of a list of FIDs assigned to each shard. Unlike regular Lustre files, stripe pattern and stripe size attributes are irrelevant since it is up to the middleware to determine how objects inside the collection are used.

A shard is responsible for managing all DAOS objects belonging to the container within a single OST. The pair <shard FID, index-within-shard> uniquely identifies a DAOS object. A shard is thus materialized on an OST by an index object (similar to a private object index) listing all the objects belonging to the container shard.

While shards have to be created explicitly through the DAOS API, DAOS objects are created and inserted into the shard index on write.

The MDS issues OST_DESTROY RPCs on all the shard FIDs when the container is destroyed. Destroying a shard involves destroying all the objects referenced in the shard index as well.

Operating on a Container

1. Collective Open

As for regular files, a DAOS container must first be opened to perform I/O operations. Open requests are managed by the MDS which returns back the container layout (i.e. list of shard FIDs) to the client. Some flags might be passed to open:

- create the container if it does not exist already (equivalent to O_CREAT)
- open mode (read/write). While multiple processes can open the same container for read, there can only be one single opener for write.

An open lock is systematically granted back to an opener. A lock callback is issued on the open lock if some other clients attempt to open the file in an incompatible manner.

When receiving an open request for a container, the MDS should create a spanning tree across the shards and notify OSTs that the client has now opened the container. The client shall then pack this open handle in any RPCs sent to the OSTs which will use the capabilities associated with the open handle to verify that the client isn't exceeding permissions (e.g. can't write/punch if opened for read).

An opener can also decide to share the open handle with peer processes running in the same cluster. To do so, the opener has to convert the local open handle into a global handle that will be then sent to its peer processes. The latter's may then convert the global handle back to a local handle which can be used for I/Os.

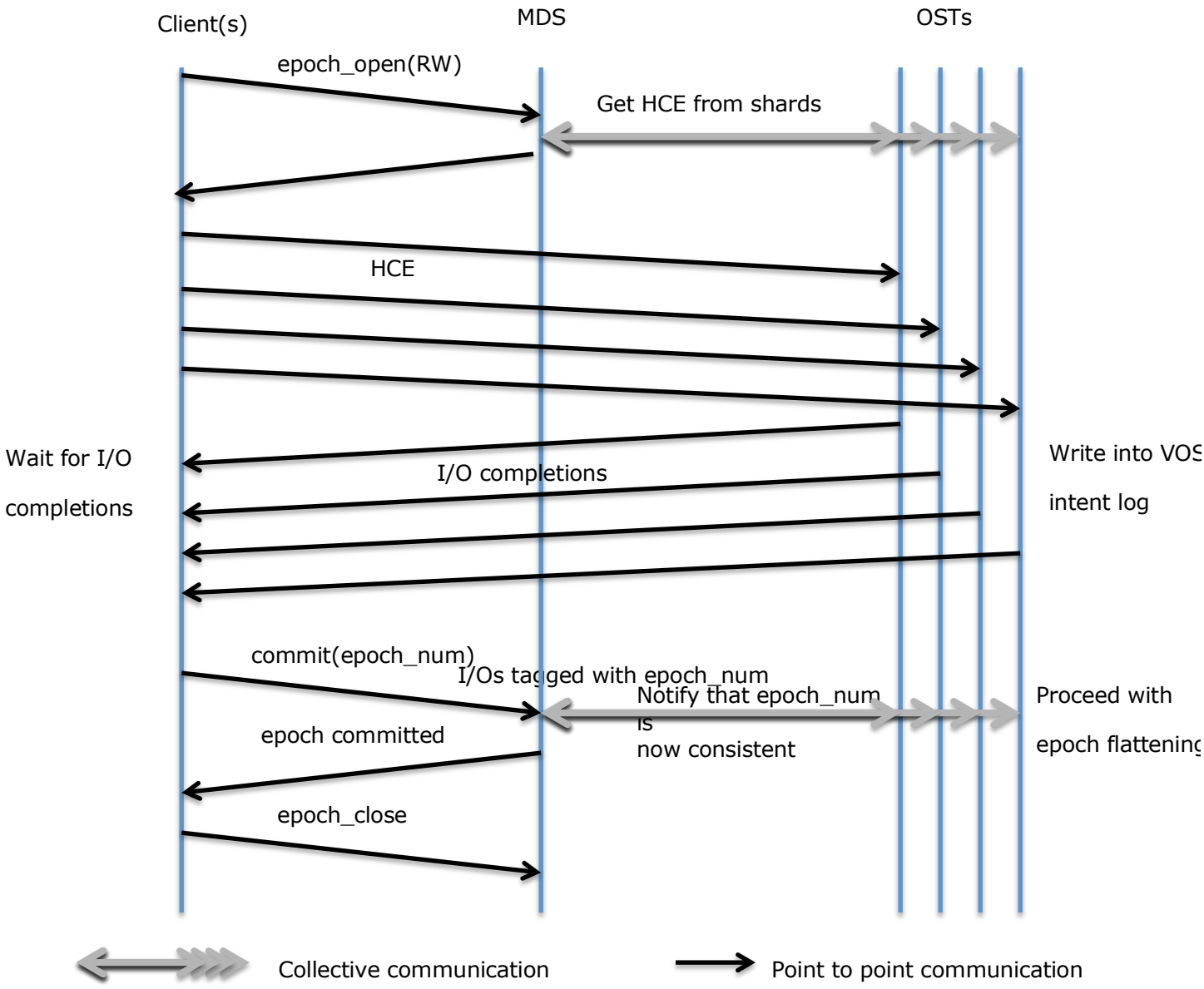
The original opener gets notification of layout changes through ASTs on the layout lock. Other processes participating in the collective open may continue to use the old layout, but will re-execute the local open when they get an updated global handle.

Peers processes are not guaranteed continued access to the container if the original opener closes its handle, exits or gets evicted. Upon close or eviction, the MDS revokes the capabilities associated with the open handle via a collective communication.

2. Epoch Lifecycle

All DAOS I/O operations are executed in the context of an epoch. Once a container is opened, the application should open an epoch scope for read and/or write. This sends a RPC to the MDS which uses the collective network to get the HCE from the container shards. Once the epoch is opened, the application can then tag I/O requests with the epoch number it would like to use and sends them to the OSTs. OSTs store those requests in the VOSD intent log.

Once all I/Os inside an epoch have completed, the parallel application shall commit the epoch. This sends an RPC to the MDS which initiates a collective communication across the OSTs to notify them that the epoch inside this container is now consistent (i.e. all I/Os inside the epoch have completed). OSTs can then commit the epoch by flattening the epoch at the VOSD level. Once the MDS has received the commit acknowledgements from all OSTs, it notifies all the clients involved in this transaction that the epoch is now committed.



If any OST involved in the epoch scope fails to commit, the epoch becomes stuck and the commit completes with failure. Clients are notified of the failure and the HCE isn't updated. The error is raised to the middleware which may enumerate the shards in the scope and disabled the ones that are stuck. Once this is fixed, the middleware may retry the commit to discover which are stuck so they can be disabled.

3. DAOS Object Operations

A DAOS object is a mutable byte array in which arbitrary bytes may be written or read. Initially, objects consume no space and all reads return 0s. Here is the list of I/O operations that can be executed:

- Write: causes block allocation on the underlying storage system. Writes are stored in the intent log until the epoch scope associated with the write is flattened. There is no extent locking which means that overlapping writes in the same epoch scope apply in unpredictable order.
- Read: on the HCE is guaranteed to be consistent thanks to snapshots. Using an epoch number different from the HCE is supported. In this case, reads aren't guaranteed to return consistent data:
    - if the epoch number is higher than the HCE, reads may fail or return a mix of data from HCE and subsequent epochs up to the epoch number specified
    - if the epoch number is lower than the HCE, reads may return a mix of data from prior epochs.
- "Punch hole": is used to reset an extent to 0s and to free up the blocks allocated in this extent, if any. This requires the backend filesystem to support hole punching, otherwise the space won't be released. In any case, [0,infinity] punch is always guaranteed to release space.
- Flush: is used to ensure that all writes reach the OSTs.

Connection Management

1. Connection Establishment

The Lustre client has to maintain an increasing number of connections (one to each target), which can become a scalability bottleneck on some large clusters. Similarly, servers have to manage thousands of exports, which consume both memory and CPU cycles and also increase the overall recovery time. Lustre clients should be able to establish connection to a group of servers by connecting only once and then set up individual connections lazily on demand.

One solution would be to divide clients among all the servers, so that a given client just has to connect to a single server at start up. Upon connection, the server initiates a collective communication to notify all the other servers that a new client is present. Then the Lustre client will connect/disconnect to the other servers lazily when needed.

2. Global Eviction

Lustre manages eviction on a per-target basis. This means that a client can get evicted from the MDS without losing connection to the OSTs. This lack of global eviction prevents some important features like Size On MDS (aka SOM) or CReate on Write (aka CROW) to be implemented properly and officially supported.

With the introduction of collective communications, Lustre can now support global eviction. When a server decides to evict a client, it can initiate a collective communication to ask all the other servers to evict the client. Each time a server completes the eviction of the client, it notifies the server that initiated the collective communication. Notifications are aggregated and reported to the first server, which now knows that all the servers have successfully evicted the client.

Likewise, connection termination (i.e. client unmount) could be handled in a collective manner.