Date: **March 4, 2013**	**Design Document – HDF5 IOD VOL** **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. 04, 2013	1.0	Mohamad Chaarawi, The HDF Group
Feb. 05, 2013	2.0	Mohamad Chaarawi, The HDF Group
Feb. 20, 2013	3.0	Mohamad Chaarawi, The HDF Group
Mar. 2, 2013	4.0 Format figures to fit correctly. Wordsmithing and expanding open issues section	Mohamad Chaarawi, Jerome Soumagne, Ruth Aydt, Quincey Koziol The HDF Group
Mar.-4, 2013	5.0 More wordsmithing, accept older changes	Mohamad Chaarawi, Jerome Soumagne, Ruth Aydt, Quincey Koziol The HDF Group

Introduction

The design of the HDF5 IOD VOL Plugin is described in this document. The application will call the HDF5 library while running on the system's compute nodes (CNs). Using the VOL architecture, the IOD VOL plugin will use a function shipper (FS) to forward the VOL calls to a server component running on the I/O nodes (IONs). At that point, the VOL calls are translated into I/O Dispatcher (IOD) API calls and executed at the IONs. The IOD will be responsible for storing the data on distributed storage using DAOS. Note that the IOD does not migrate the data right away to the DAOS library, but uses a local Burst Buffer and a log file system (PLFS) on temporary storage for better performance and defensive I/O.

Definitions

CN = Compute Node

EFF = Exascale FastForward

FS = Function Shipper

IOD = I/O Dispatcher

ION = I/O Node

VOL = Virtual Object Layer

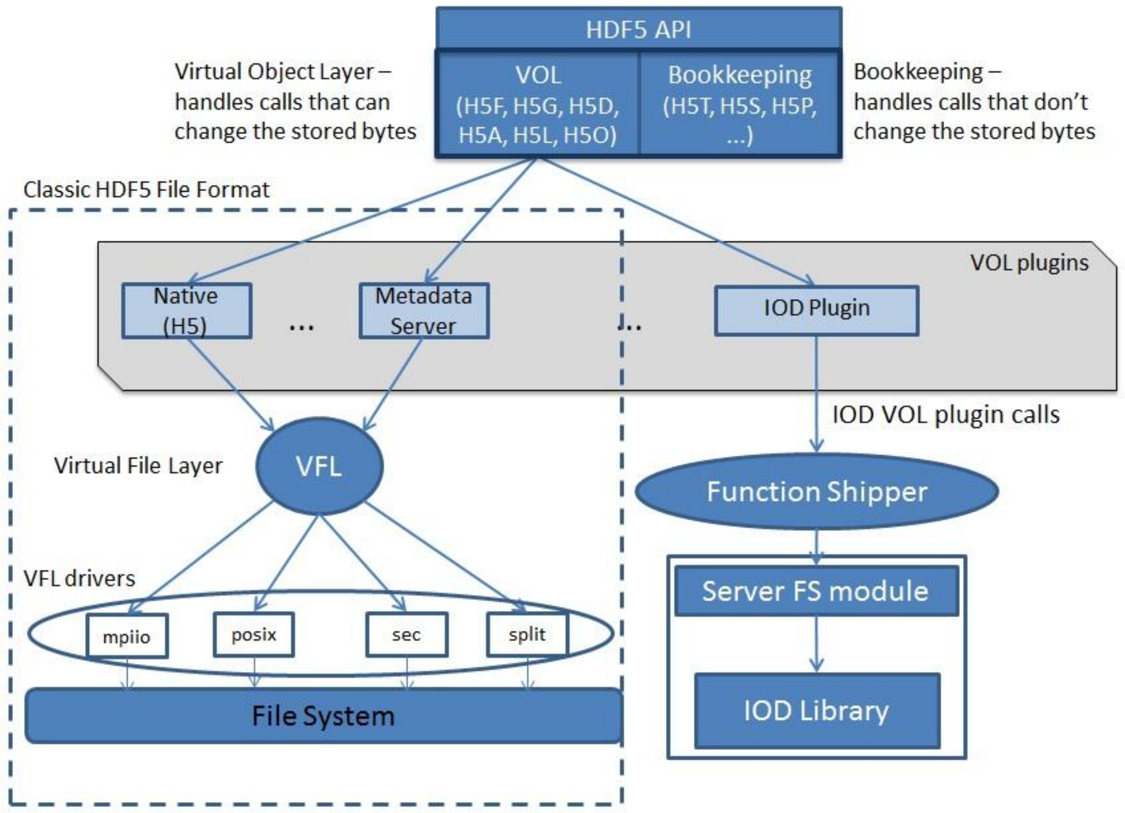
Changes from Solution Architecture

There are currently no changes from the Solution Architecture descriptions.

Specification

The HDF5 VOL intercepts all HDF5 calls that would potentially touch the storage and routes them to an internal or user-developed plugin. This allows for HDF5 objects to be stored in different file formats or storage abstractions that are hidden from the application, allowing the application to continue using the same HDF5 API and data model while benefiting from new storage methods and architectures.

Developing an IOD plugin at the HDF5 VOL level would be the ideal approach, because the VOL abstraction is high enough to provide metadata information to the plugins about the objects and raw data. The overall architecture of the HDF5 library with the addition of the IOD plugin will look like this:



Components of the Stack

The EFF storage software stack contains several components essential to the proper functioning and performance of the application I/O. The scope of this RFC is from the HDF5 library to the high level IOD API routines. This means that we will not discuss here how IOD implements its API internally or its interaction with the DAOS library and underlying distributed storage.

1.1 The HDF5 Library and the Client Side IOD VOL

The application that is running on the CNs of the Exascale system uses the HDF5 library for I/O and selects the IOD VOL plugin for storing its data. The VOL layer in the HDF5 library captures HDF5 API calls that modify data on disk and routes them through the IOD plugin. The IOD plugin at the application (client) side will make use of the Function Shipper [2] to forward the VOL operations to the IONs (server side).

1.2 The Function Shipper

The Function Shipper (FS) is an RPC mechanism that forwards the VOL calls made at CNs on the client side to the IONs, where the FS server is located. The function shipper extends the IOFSL package [3], a general forwarding framework developed at Argonne National Lab. The FS is general enough to allow its users to ship any type of operation and has a framework for extending the operations it supports. The FS API is asynchronous, which supports the asynchronous functionality needed in this project. The asynchronous request objects received from FS operations are stored and tracked in the IOD VOL plugin described above.

1.3 The Server FS Module

A server FS module must operate at the IOD nodes to receive operations from clients. The module can be considered a server-side component of the IOD VOL plugin that is responsible for translating VOL operations into IOD API calls. Each VOL operation maps to one or more IOD API calls. Those VOL operations are inserted into an asynchronous engine running on the ION, possibly with a dependency on another VOL operation indicated by the client-side VOL plugin.

1.4 The Asynchronous Engine

One of the new features proposed in this EFF project is the addition of asynchronous operations to the HDF5 library. Asynchronous HDF5 operations will return immediately to the application, and the application may test or wait for their completion. Both raw data as well as metadata operations can be asynchronous. The semantics for asynchronous operations are detailed in the "HDF5 asynchronous I/O, data integrity and data layout extensions" design document.

1.5 IOD Library

IOD daemons running on the IONs respond to requests from the function shipper's server-side IOD VOL plugin, which will issue IOD API calls through tasks enqueued in the asynchronous engine. The IOD API and functionality are well defined for us, and we can assume an implementation would be available for us to use at this point, where the scope of this document ends.

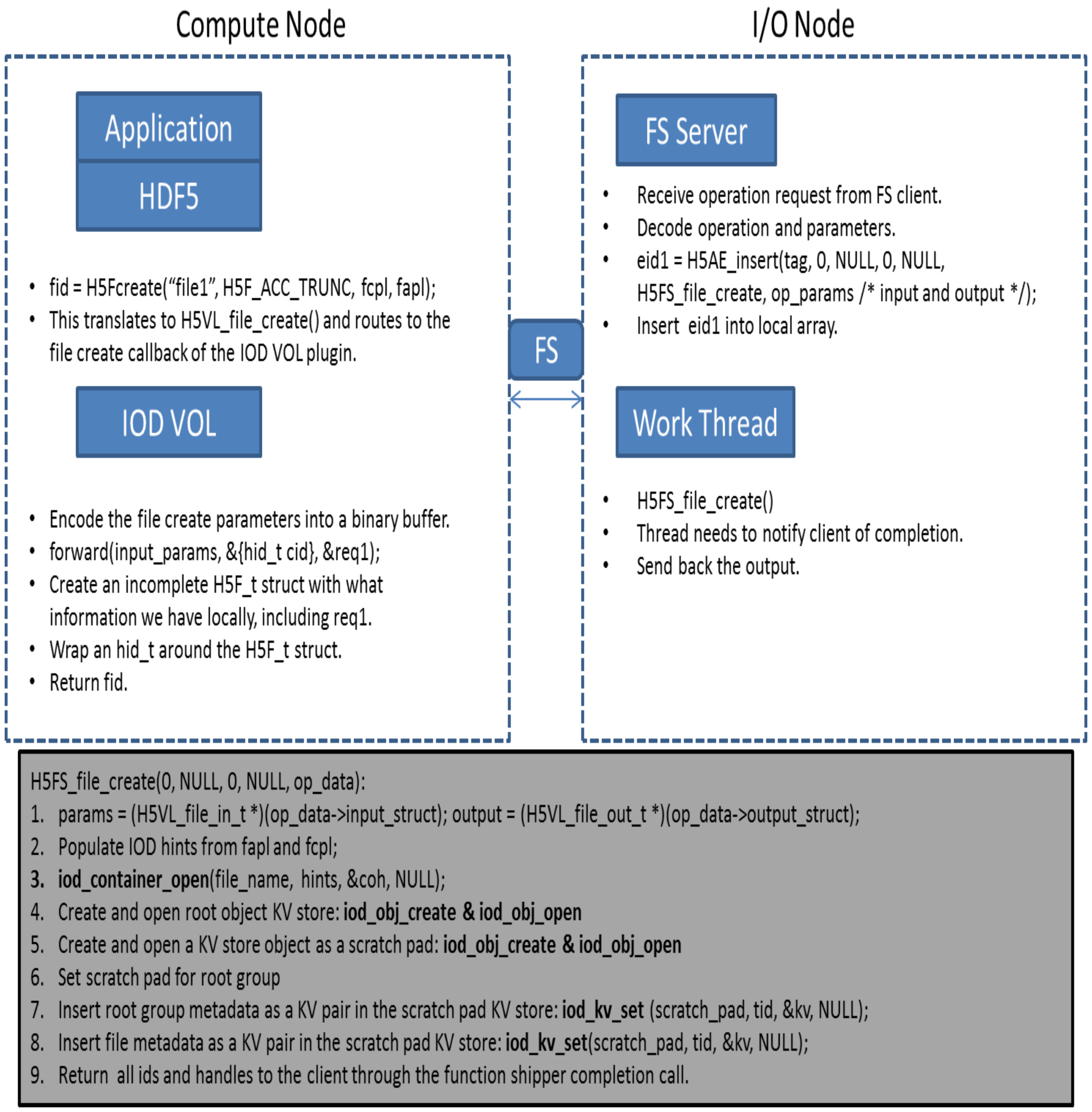
Prototype of the IOD VOL Plugin

This section delves into some technical details on how the IOD VOL plugin is implemented, showing pseudo-code and flow diagrams of how selected HDF5 operations are implemented. Operations shown in this document are limited to object creates and dataset writes and will be expanded to include other HDF5 routines in the future. New functionality proposed in this research will expand the VOL interface, so we will use an incremental approach to implementing the IOD VOL plugin.

1.6 Initialize the EFF Stack

The first step is to initialize the entire software stack by calling a new routine named EFF_init(). This routine establishes communication between the CNs and the IONs through the function shipper, in addition to initializing the HDF5 and IOD packages.

1.7 Create a File/Container



Compute Node:

- fid = H5Fcreate("file1", H5F_ACC_TRUNC, fcpl, fapl);
- This translates to H5VL_file_create() and routes to the file create callback of the IOD VOL plugin.

- Encode the file create parameters into a binary buffer.
- forward(input_params, &{hid_t cid}, &req1);
- Create an incomplete H5F_t struct with what information we have locally, including req1.
- Wrap an hid_t around the H5F_t struct.
- Return fid.

I/O Node:

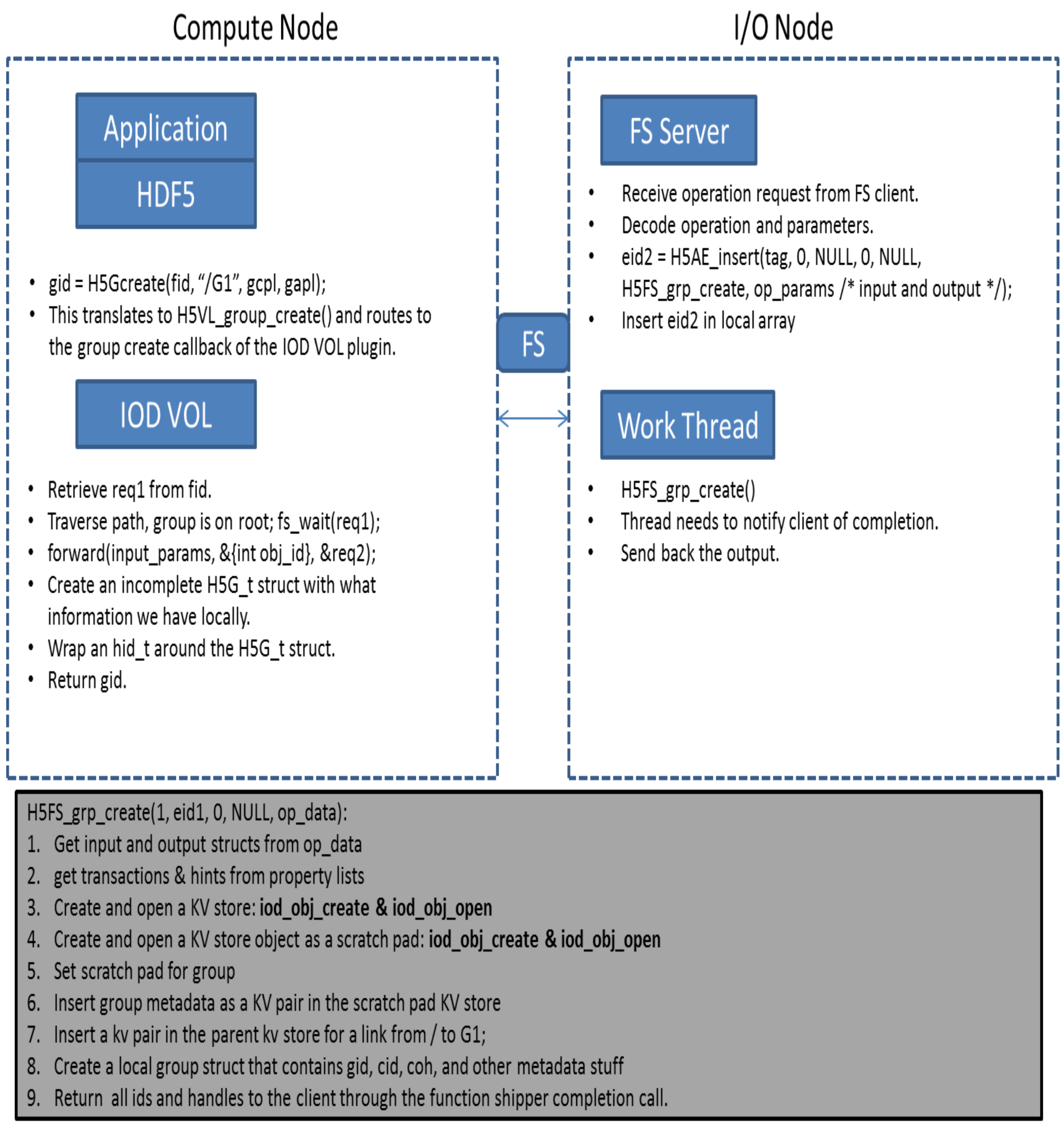
- Receive operation request from FS client.
- Decode operation and parameters.
- eid1 = H5AE_insert(tag, 0, NULL, 0, NULL, H5FS_file_create, op_params /* input and output */);
- Insert eid1 into local array.

- H5FS_file_create()
- Thread needs to notify client of completion.
- Send back the output.

H5FS_file_create(0, NULL, 0, NULL, op_data):

1. params = (H5VL_file_in_t *)(op_data->input_struct); output = (H5VL_file_out_t *)(op_data->output_struct);
2. Populate IOD hints from fapl and fcpl;
3. **iod_container_open**(file_name, hints, &coh, NULL);
4. Create and open root object KV store: **iod_obj_create & iod_obj_open**
5. Create and open a KV store object as a scratch pad: **iod_obj_create & iod_obj_open**
6. Set scratch pad for root group
7. Insert root group metadata as a KV pair in the scratch pad KV store: **iod_kv_set** (scratch_pad, tid, &kv, NULL);
8. Insert file metadata as a KV pair in the scratch pad KV store: **iod_kv_set**(scratch_pad, tid, &kv, NULL);
9. Return all ids and handles to the client through the function shipper completion call.

1.8 Create a Group



Compute Node:

- gid = H5Gcreate(fid, "/G1", gcpl, gapl);
- This translates to H5VL_group_create() and routes to the group create callback of the IOD VOL plugin.

IOD VOL:

- Retrieve req1 from fid.
- Traverse path, group is on root; fs_wait(req1);
- forward(input_params, &{int obj_id}, &req2);
- Create an incomplete H5G_t struct with what information we have locally.
- Wrap an hid_t around the H5G_t struct.
- Return gid.

FS Server:

- Receive operation request from FS client.
- Decode operation and parameters.
- eid2 = H5AE_insert(tag, 0, NULL, 0, NULL, H5FS_grp_create, op_params /* input and output */);
- Insert eid2 in local array

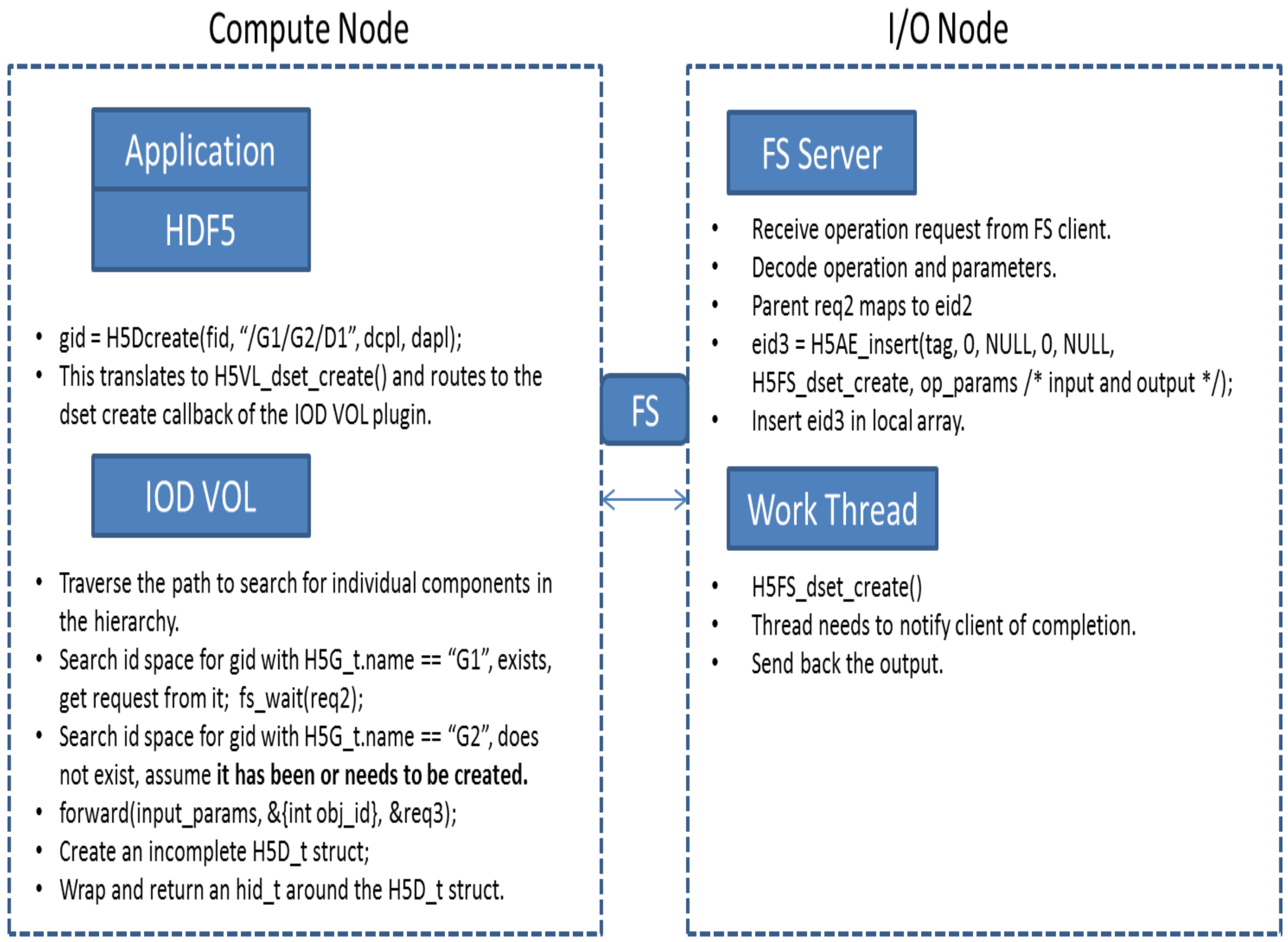
Work Thread:

- H5FS_grp_create()
- Thread needs to notify client of completion.
- Send back the output.

H5FS_grp_create(1, eid1, 0, NULL, op_data):

1. Get input and output structs from op_data
2. get transactions & hints from property lists
3. Create and open a KV store: **iod_obj_create & iod_obj_open**
4. Create and open a KV store object as a scratch pad: **iod_obj_create & iod_obj_open**
5. Set scratch pad for group
6. Insert group metadata as a KV pair in the scratch pad KV store
7. Insert a kv pair in the parent kv store for a link from / to G1;
8. Create a local group struct that contains gid, cid, coh, and other metadata stuff
9. Return all ids and handles to the client through the function shipper completion call.

1.9 Create a Dataset



- gid = H5Dcreate(fid, "/G1/G2/D1", dcpl, dapl);
- This translates to H5VL_dset_create() and routes to the dset create callback of the IOD VOL plugin.

- Traverse the path to search for individual components in the hierarchy.
- Search id space for gid with H5G_t.name == "G1", exists, get request from it; fs_wait(req2);
- Search id space for gid with H5G_t.name == "G2", does not exist, assume **it has been or needs to be created.**
- forward(input_params, &{int obj_id}, &req3);
- Create an incomplete H5D_t struct;
- Wrap and return an hid_t around the H5D_t struct.

- Receive operation request from FS client.
- Decode operation and parameters.
- Parent req2 maps to eid2
- eid3 = H5AE_insert(tag, 0, NULL, 0, NULL, H5FS_dset_create, op_params /* input and output */);
- Insert eid3 in local array.

- H5FS_dset_create()
- Thread needs to notify client of completion.
- Send back the output.

H5FS_dset_create(1, eid2, 0, NULL, op_data):

1. Get input and output from op_data
2. Input should contain group "G1" information which includes gid, handle, etc...
3. Lookup "G2" in KV store of gid. Should not exist, so Create a group KV, gid2, store and link it to gid.
4. Create a KV store object as a scratch pad;
5. Create the new array object after setting the array structure using the information from the space and type decoded;
6. Insert dset metadata as a KV pair in the scratch pad KV store
7. Insert a kv pair in the parent kv store for a link from G2 to D1
8. Return all open IDs and handles for dataset

1.10 Writing to a dataset

The main challenge in writing data elements to an HDF5 dataset is translating the memory and file space information from the HDF5 API call to actual locations in memory and the container respectively. Using the HDF5 memory datatype and space selection we can construct an array of {offset, length} pairs that represents the data in memory. The selection in the file space then needs to be mapped into a hyperslab IOD struct, which consists of four arrays of length equal to the object dimension and includes the start, count, stride, and block parameters of where the data needs to be written.

The IOD VOL plugin on the client will handle any type conversion needed before sending data over to the IONs.

Open Issues

The current prototype assumes all HDF5 metadata operations are asynchronous and independent. We will add an option to allow the application indicate whether such operations are independent or collective. The collective mode could provide a performance benefit depending on how all the ranks in an application access data.

The issue we encountered in case of collective metadata access is that a fully asynchronous model will be hard to implement. A single rank has to talk to the function shipper server to issue the IOD operations, then broadcast the result to all other ranks so they can go and access the objects. The collective communication in that case will cause other ranks to wait on the leader rank for information, which limits the asynchronous capability.

A different approach to solve this problem is to have all ranks talk asynchronously to the function shipper server to create object, for example. We can use the IOD KV store objects to manage access to the container. The first process to update a KV store object would be the leader which would create the object while other are polling on the KV store. This might not seem efficient (translates as locks on a file system), but certain applications may benefit from the full asynchrony that would be achieved at the client side. However, the main limitation from the IOD library in that case, is that objects that are created during a transaction need to be committed before being accessed, which might not suit most applications.

Risks & Unknowns

As the changes to the HDF5 library are dependent on capabilities added to multiple lower layers of the software stack (the function shipper, IOD and DAOS layers), it is likely that changes at those layers will ripple up through the HDF5 API and cause additional work and modifications at the VOL layer.

Conversely, the demands of the applications that use the HDF5 API may pull the features and interface in unexpected directions as well, in order to provide the necessary capabilities for the application to efficiently and effectively store its data. These two forces must be balanced over the course of the project.