Date: January 10, 2013	**IOD KV Store Design Document** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

Table of Contents

Revision History

Date	Revision	Author
2013-01-10	0.1	Zhenhua Zhang
2013-01-15	0.2	Zhenhua Zhang
2013-01-17	0.3	Zhenhua Zhang
2013-01-24	0.4 Reduce the IOD KV API number	Zhenhua Zhang
2013-02-16	0.5 align same semantic with blob/array objects And change APIs to version of Feb 7.	Zhenhua Zhang

Introduction

IOD distributed KV store as part of EFF project which targets to redesign the whole IO stack towards exscale computing provides KV interfaces for storing small metadata for HDF objects. It's a hierarchical system which is built on top of multiple DBMSs (PBL ISAM in our case) running on the IONs of a supercomputer. In IOD KV system, MDHIM client will use a pre-defined sharding method to direct each record to a range server. Then the range server will store the record in its local DBMS. Besides the existing KV store functionality, IOD KV store introduces versioning to provide data consistency and availability so user can keep multiple states for each key at the same time. IOD KV system also utilizes the fast interconnecting network between the IONs to pass request and result between MDHIM client and range server. DAOS, a new object oriented storage system invented by Intel, will be used as IOD KV system's long term storage and IOD will be able to migrate its KV stores between the burst buffers and the DAOS system. Together all the new features, we provide a new distributed KV store which provides extreme scalability and data consistency and high throughput.

Definitions

CN	Computer nodes of a supercomputer
ION	I/O nodes of a supercomputer
HDF Client	HDF client is the application runs on CN which uses IOD API.
Function shipper	A forwarding layer that exports APIs from IONs to CNs. It's client-server model that client runs with application on CN and server on ION.
IOD	I/O Dispatcher running on IONs
DAOS	Distributed Application Object Storage
MDHIM	Multi-Dimensional Hierarchical Indexing Middleware
PBL	Program Base Library
ISAM	Indexed Sequential Access Method
SAL	Storage Abstraction Layer
MDHIM client	MDHIM client running on ION is used for accepting requests from user (HDF Function shipping server) and directing to range server for handling.
Range server	Range server running on ION is a role in MDHIM which accepts requests from MDHIM client and read/write its local DBMS accordingly.

Changes from Solution Architecture

Specification

IOD distributed KV store consists of IOD KV APIs which will internally call MDHIM client routines, request dispatcher, range server which supports multiple backend. The MDHIM client API uses a pre-defined hashing method to route each request to its range server. The range server will then execute the request by reading/writing to its POSIX/DAOS backend. Figure 1 describes the high level architecture of IOD KV Store.

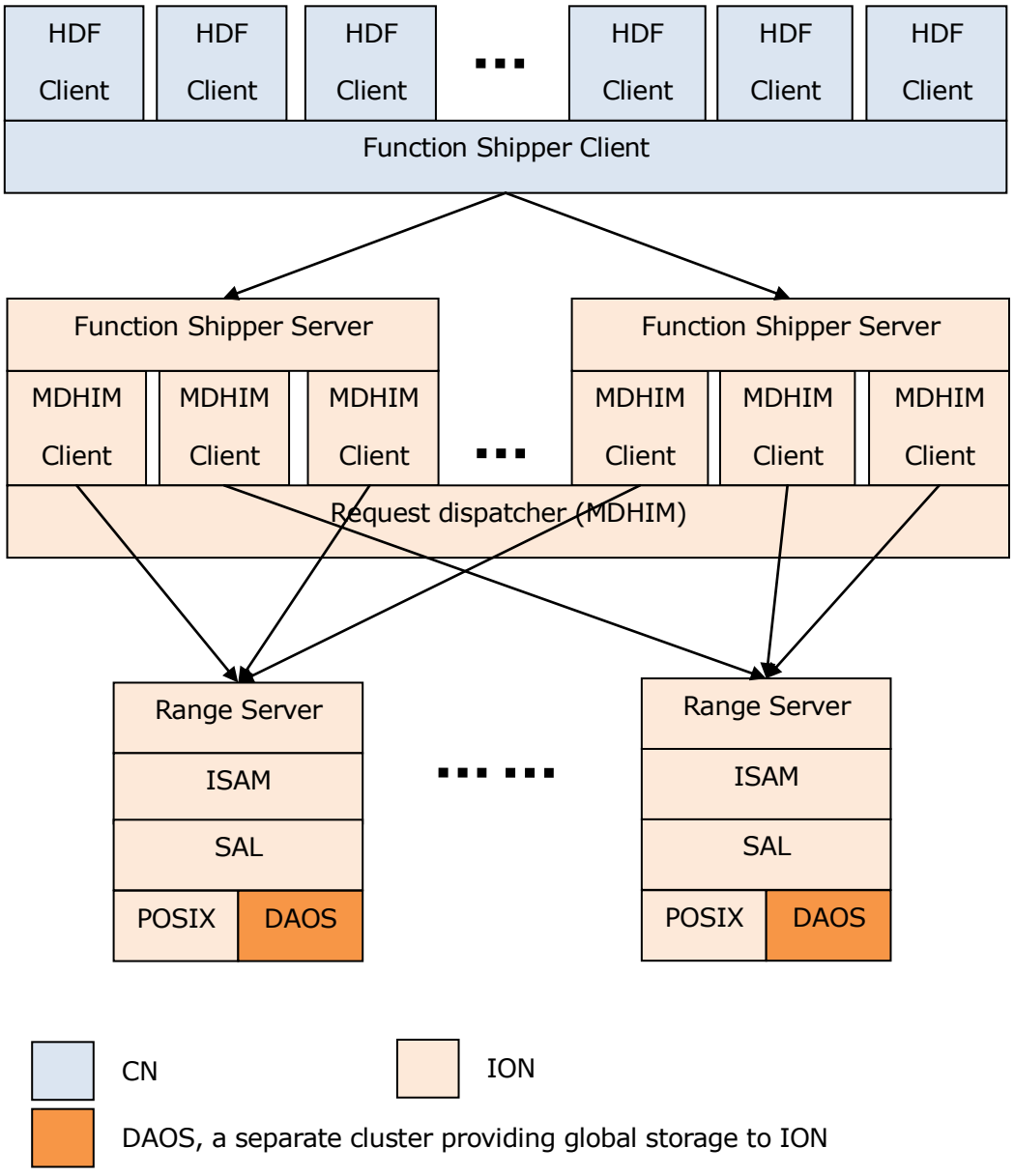

Figure 1: IOD KV store high level architecture

Following is a sample process example of how IOD KV store works:

1. HDF clients running on CNs call iod_init() to start the IOD service. This request will be forwarded to function shipper server to be executed. Iod_init will then call iod_kv_init which internally calls mdhim_init to start the IOD KV service. All MDHIM clients collectively call the init procedure with one MPI communicator. Rank0 will read in configuration file and determine the range servers and broadcast to all. Each range server will then start a thread to receive requests from client.
2. HDF clients can send create/set/get/list/delete request command which will be forwarded to to function shipper server. These requests will be handed out to MDHIM client to route them to different range server independently using a predefined hash method.
3. Range server will execute client's request and communicate with client asynchronously.
4. HDF clients call iod_finalize() to do resource cleanup.

Environment and limits

The IOD KV store will be deployed on the IO nodes of a supercomputer, which have fast interconnected network. At the IOD init time, user must provide one MPI communicator which evolves all the processes in the workload. Following parameters can be configurable,

Global parameters for IOD KV services:

- KV store path

Parameters for IOD KV object:

- Range server list
- Number of range servers
- Maximum length of key
- Maximum length of value
- Key type (string/integer/float)

The object parameters can be passed in as hints when creating a KV object.

Versioning

Besides the normal key value system functionality, IOD KV store introduces versioning concept which means it can preserve multiple states at different "timestamps" for each key. The following record format will be used to support versioning.

Key	Tag	Value

Here tag is a 64 bit integer which can be timestamp, transaction ID and so on. User must ensure it to be unique for each key. So compared with traditional KV system in which value can be retrieved by its key, now the value is determined by <Key, Tag> pair.

Two pre-defined tags are reserved for special usage. NULLTAG can be used to disable multiple versioning for some records. For the NULLTAG records, user can update its value in position freely. NULLTAG can also be used for the operations that are not in a transaction. MAXTAG can only be used in get operation to retrieve the latest version of value but can't be used in set operation.

```
#define NULLTAG 0X0

#define MAXTAG 0XFFFFFFFFFFFFFFFF
```

If user inserts a non NULLTAG record, it represents a consistent state for the key. User can update its state by overwriting the value with the same tag or inserting a new record with a new tag.

For a get operation, IOD KV store will return most recent version compared with the tag passed in.

All the states for each key will be preserved unless user implicitly deletes a record.

Request dispatcher

IOD KV system runs on IONs of supercomputer which has fast interconnect between all nodes and uses given MPI communicator to send/receive message between client and range server. The request dispatcher needs to shard key space and store the record set using multiple servers to provide scalability, capacity and high throughput. MDHIM, an open sourced project developed by LANL, satisfies all the requirements. Our IOD KV system will use MDHIM as its request dispatcher.

Figure 2 describes MDHIM architecture.

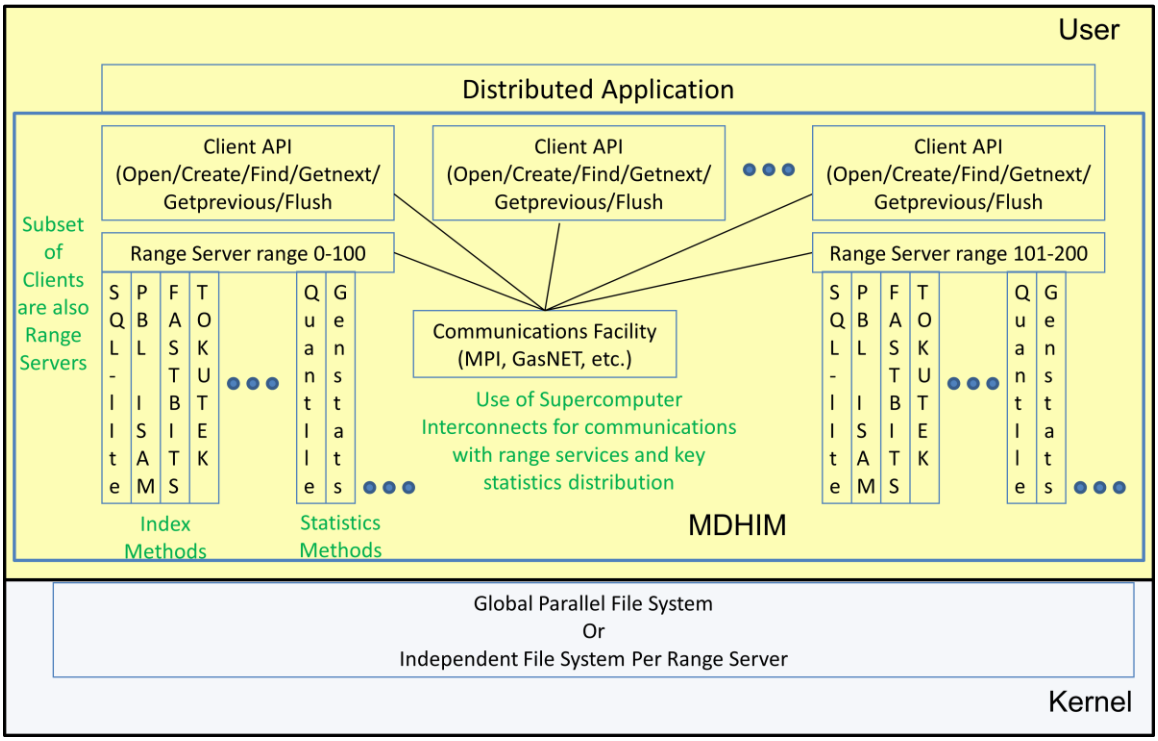

Figure 2: MDHIM architecture

At IOD KV store create phase, IOD instance will determine which rank will work as range server according to the parameter list. And MDHIM clients will then use a pre-defined hash method to shard the whole key space into many pieces of ranges and assign each range to different range server. Each range server will act as execute unit to interact with its POSIX or DAOS backend using ISAM. In future, more accessing methods and more backend type will be added into MDHIM.

Table 1 describes the request formation used by client and range server. Each operation can be implemented by corresponding PBL ISAM routines. For now, pblIsamReadData() and pblIsamReadDatalen() are not support by MDHIM.

Operation	Parameter list	Corresponding PBL ISAM routine	Supported by current MDHIM
create		pblIsamOpen	yes
Insert	Number of records[IN], records list[IN]	pblIsamInsert	yes
get	Number of keys[IN], key list[OUT]	pblIsamGet	yes
readdata	Buffer[IN], buffer size[IN]	pblIsamReadDatalen	no
		pblIsamReadData	no
list	Key list[OUT]	pblIsamGet	yes
unlink	Number of keys [IN]	pblIsamFind	yes
		pblIsamDelete	yes

Table 1: request command formation

Range server

Range server is responsible for receiving client's request and executing it. In current version of MDHIM, the range server uses PBL ISAM to access its POSIX backend. For EFF project, we will need range server to interact with DAOS storage. Figure 3 describes the ISAM changes to satisfy IOD KV store needs. New components in blue will be added.

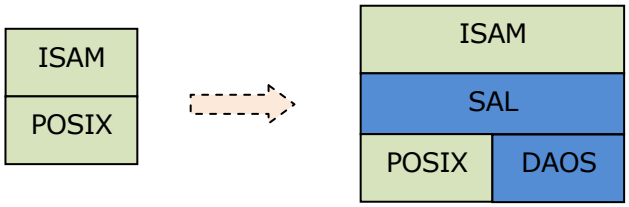

Figure 3: ISAM architecture

SAL, short for Storage Abstraction Layer, is required to expose a set of storage interfaces for ISAM usage and block the differences for each backend. It's responsible for translating an up layer IO request to correct underlying backend APIs. SAL need to support at least POSIX and DAOS backend. While POSIX and DAOS are really different because POSIX is a file based interface standard and DAOS exposes a new set of object oriented APIs, their IO operations are all based on offset. This will greatly simplify the design of SAL.

Besides SAL, DAOS backend support will be added to current PBL ISAM code. In that case, ISAM will treat a DAOS object as a logical file, which can be accessed by offset. A DAOS object may stripes on columns of many DAOS shards, this kind of metadata need to be kept to map a logical <offset, length> pair to correct DAOS <shard, column> list in read/write operations.

The following are the main data structures in SAL.

```
enum SAL_BKD_TYPE {

      SAL_POSIX = 1,

      SAL_DAOS  = 2

};

typedef enum SAL_BKD_TYPE SAL_BKD_TYPE;

typedef struct {

      char *path; // NULL terminated

      SAL_BKD_TYPE type;

      Union{

            int fd;

            iod_obj_id_t oid;

      }handle;

} sal_handle; // structure that associates path and underlying backend

               // storage system's handle
```

The minimum set of SAL IO API abstractions are as follows. Because SAL is designed at file/object level, file system APIs and many unused file based APIs are not listed here.

```
//malloc memory internally for sal_handle

sal_handle *handle sal_open(char *path, int mode);


int ret // number of bytes written

sal_write(sal_handle *sal_h,
```

```
          char *buffer,  // user passed in
          size_t length, // length of buffer
          off_t offset     // the logical offset to write
          );


int ret // number of bytes read out
Sal_read(sal_handle *sal_h,
         Char *buffer,  // user passed in to receive data
         Size_t length, // the length of buffer
         Off_t offset     // the logical offset to read)
        };


// free the memory belong to sal_h
Int sal_close(sal_handle *sal_h);
```

KV migration

In the EFF project, IOD will preserve the KV store in ION's local burst-buffer POSIX storage and migrate it to DAOS after receiving migration request from user. After migration, the whole record set will be divided in to two parts, one in BB and the other in DAOS. The following record insertion will go to BB by default for performance. And following get/read requests will be satisfied from BB if available, otherwise it will retrieve data from DAOS.

The IOD API iod_obj_set_layout() can be used to migrate the KV store to DAOS by passing an KV object ID. Hints can be set to control the semantic resharding policy.

The following is a sample process for migrating IOD KV store from ION to DAOS.

1. One MDHIM client receives the KV migration request via iod_obj_set_layout(TID) call, it will send migrate command to all range servers.
2. Every range server will stop accepting new request after getting migration notification.
3. Each range server will do bulk query and find out all records with a smaller transaction ID than TID and store them into DAOS.
4. Range servers continue to accept new request after migration is done.

API and Protocol Additions and Changes

Many of the data structures and APIs are copied from IOD API document, may need to change accordingly. Now they are based on version of Feb 7, 2013.

IOD KV object, as one of the supported object types in IOD, will use the IOD object create/unlink/open/close/set_layout routines. Besides these operations, IOD KV store need to provide following APIs.

API	Description
iod_kv_set	Set Key-Value pair to one KV object
iod_kv_set_list	
iod_kv_get_num	Get the number of Key-Value pairs of one KV object
iod_kv_get_list	Get a list of KV pairs from one KV object
iod_kv_list_key	Retrieve a list of keys in the KV object
iod_kv_get_value	Get one KV pair belong to the key from one KV object
iod_kv_unlink_keys	Unlink a list of keys in the KV object

The main data structure describing a KV record,

```
/** IOD KV store supported key type */

typedef enum {

      IOD_KV_STRING, // for char array

      IOD_KV_INT,    // for int

      IOD_KV_FLOAT,  // for float

} iod_kv_key_type_t;

/** A first-class structure for a Key-Value pair */

#define IOD_KV_KEY_MAXLEN(1024)

typedef struct {

      const char   *key;        /** must be NULL termed */

      const void   *value;

      iod_size_t   value_len;

      } iod_kv_t;
```

The main IOD KV API

```
/**
   * A collective call to initialize the IOD KV service.
   * Called internally by IOD.
   */
Iod_ret_t
iod_kv_init(iod_kv_info_t *info,
            MPI_COMM comm);
/**
   * A collective call to end IOD KV service.
   * This routine will do resource cleanup before ending IOD KV service.
   * Called internally by IOD.
   */
Iod_ret_t
iod_kv_finalize(iod_kv_info *info);


/**
 * Set one Key-Value pair to one KV object.
 *
 * \param oh [IN]  object handle
 * \param tid [IN] transaction ID
 * \param hints[IN]      pointer to hints and can be NULL when no hint
 * \param kv [IN]  KV pair pointer
 * \param event [IN]     pointer to completion event
 *
 * \return         zero on success, negative value if error
 */
iod_ret_t
iod_kv_set(iod_handle_t oh, iod_trans_id_t tid, iod_hint_list_t *hints,
         iod_kv_t *kv, iod_event_t *event);
```

```
/**
 * Set a list of Key-Value pairs to one KV object.
 *
 * \param oh [IN]  object handle
 * \param tid [IN] transaction ID
 * \param hints[IN]      pointer to hints and can be NULL when no hint
 * \param num [IN] number of KV pairs
 * \param kvs [IN/OUT]   pointer to KV parameters packet list
 * \param event [IN]     pointer to completion event
 *
 * \return         zero on success, negative value if error
 */
iod_ret_t
iod_kv_set_list(iod_handle_t oh, iod_trans_id_t tid, iod_hint_list_t
*hints,
             iod_size_t num, iod_kv_params_t **kvs, iod_event_t *event);


/**
 * Get the number of Key-Value pairs of one KV object.
 *
 * \param oh [IN]  object handle
 * \param tid [IN] transaction ID
 * \param num [IN/OUT]   pointer of number of KV pairs
 * \param event [IN]     pointer to completion event
 *
 * \return         zero on success, negative value if error
 */
iod_ret_t
iod_kv_get_num(iod_handle_t oh, iod_trans_id_t tid, iod_size_t *num,
              iod_event_t *event);
```

```
/**
 * Get a list of KV pairs from one KV object.
 * Caller needs to provide and free the buffer for KV-pairs.
 *
 * \param oh [IN]  object handle
 * \param tid [IN] transaction ID
 * \param hints[IN]      pointer to hints and can be NULL when no hint
 * \param offset [IN] offset of kv pair inside the KV object(begin from 0),
 *                 ordered by ascending alphabetical order of key (from
 *                 'a' to 'z').
 * \param num [IN] number of KV pairs
 * \param kvs [IN/OUT]   pointer to KV parameters packet list
 * \param event [IN]     pointer to completion event
 *
 * \return         zero on success, negative value if error
 */
iod_ret_t
iod_kv_get_list(iod_handle_t oh, iod_trans_id_t tid, iod_hint_list_t
*hints,
             iod_off_t offset, iod_size_t num, iod_kv_params_t **kvs,
             iod_event_t *event);


/**
 * Retrieve a list of keys in the KV object.
 * Caller needs to provide and free the buffer for key list, every key is a
 * buffer of IOD_KV_KEY_MAXLEN length. Needs not malloc buffer for value
 * inside iod_kv_params_t, it will be ignored by this call.
 *
 * \param oh [IN]  object handle
```

```
 * \param tid [IN] transaction ID
 * \param hints[IN]      pointer to hints and can be NULL when no hint
 * \param offset [IN]    offset of key inside the KV object(begin from 0),
 *                 ordered by ascending alphabetical order (from 'a'
 *                 to 'z').
 * \param num [IN] number of KV pairs
 * \param kvs [IN/OUT]   pointer to KV parameters packet list
 * \param event [IN]     pointer to completion event
 *
 * \return         zero on success, negative value if error
 */
iod_ret_t
iod_kv_list_key(iod_handle_t oh, iod_trans_id_t tid, iod_hint_list_t
*hints,
             iod_off_t offset, iod_size_t num, iod_kv_params_t **kvs,
             iod_event_t *event);


/**
 * Get the value belong to the key from one KV object.
 * Caller needs to provide and free the buffer for value, if user passed in
 * buffer length \a len is not enough to hold the value then when this
routine
 * returns the \a len is set as the actual length of value.
 *
 * \param oh [IN]        object handle
 * \param tid [IN]       transaction ID
 * \param key [IN]       passed in key
 * \param value [IN/OUT] returned value
 * \param len [IN/OUT]         pointer of length of passed in buffer of
value
 * \param event [IN]           pointer to completion event
```

```
 *
 * \return               zero on success, negative value if error
 */
iod_ret_t
iod_kv_get_value(iod_handle_t oh, iod_trans_id_t tid, const char *key,
              char *value, iod_size_t *len, iod_event_t *event);


/**
 * Unlink a list of keys in the container.
 * Caller needs to provide and free the buffer for key list, every key is a
 * NULL termed string(within length of IOD_KV_KEY_MAXLEN). Needs not malloc
 * buffer for value inside iod_kv_params_t, it will be ignored by this
call.
 *
 * \param oh [IN]  object handle
 * \param tid [IN] transaction ID
 * \param hints[IN]      pointer to hints and can be NULL when no hint
 * \param num [IN] number of KV pairs
 * \param kvs [IN/OUT]   pointer to KV parameters packet list
 * \param event [IN]     pointer to completion event
 *
 * \return         zero on success, negative value if error
 */
iod_ret_t
iod_kv_unlink_keys(iod_handle_t oh, iod_trans_id_t tid, iod_hint_list_t
*hints,
               iod_size_t num, iod_kv_params_t **kvs, iod_event_t *event);
```

Open Issues

1. In future, more data stores other than ISAM may be added to MDHIM. It's on the developer's roadmap now. We can leverage or add any DBMS if needed.

2. IOD APIs are still in review process now. IOD KV APIs may need to change accordingly.

Risks & Unknowns