CMM DLD

Mike Pershin

May 8, 2006

Contents

1 Requirements

The CMM is a new layer in the MDS which cares about all clustered metadata issues and relationships. The CMM does the following:

- acts as layer between the MDT and MDD;
- provides MDS-MDS interaction;
- queries and updates FLD;
- does the local or remote operation if needed;
- does rollback - epoch control, undo logging (details are in rollback DLD).

2 Functional specification

2.1 CMM functionality

CMM chooses all servers involved in operation and sends depended request if needed. The calling of remote MDS is a new feature related to the CMD. It means the new stage in command processing on MDS.

Current design is based on MDS Layering DLD in sections of device initialization and operation.

2.1.1 Phases of operation

1. RECEIVED
2. PROCESSING
   (a) REMOTE PROCESSING

(b) REMOTE PART IS DONE

(c) LOCAL PROCESSING (STARTING THE TRANSACTION IF NEEDED, LOCAL LOCKING)

3. DONE
4. COMMITTED LOCALLY
5. COMMITTED GLOBALLY

The remote processing stage is optional and the most common case has only 1 MDS involved. If operations should be done on several MDS then it is split in remote and local parts. The MDAPI already contains new operations appeared due to such splitting. All remote operations should be done before any local one to avoid doing RPC inside the local transaction. This is main principle while handling operations in CMM.

2.1.2 Local and remote objects

The CMM can allocate two types of object - local and remote. Remote object can occur during metadata operations with more than one object involved. Such operation is called as cross-ref operation. The struct md_object is extended to support the remote object - one which is stored on different MDS. Such object has only valid fid when allocated.

2.1.3 Restrictions on command processing

There are several rules for command processing:

1. remote part of operations must be done at first. This is requirement of rollback functionality because the epoch should be negotiated during the RPC;
2. there should be only one remote call. The reason for this is the same as above - during second RPC epoch can be changed, but first RPC was done already in old epoch;
3. the local part of operation should be in one transactions on the local filesystem. So any write operation should be done in CMM as single call to the MDD.
4. There should be no RPC inside disk transaction.

Note: the rollback DLD is not finished at the moment of writing this one so restrictions due to rollback can be changed in future.

2.2 Basic structures

CMM is `md_device` in lustre stack - the extension of more generic `lu_device`. Therefore CMM supports several set of operations.

`struct lu_fid` - FID of the object. Unique identifier in Lustre;

`struct lu_device` - generic device structure;

`struct md_device` - extension of lu_device for metadata devices;

`struct cmm_device` - CMM device parameters;

`struct lu_object` - generic object structure to be used between the layers;

`struct md_object` - extension of lu_object for metadata devices;

`struct cmm_object` - CMM private data for the local object;

struct cmm_robject - CMM data for the remote object.

2.3 CMM configuration

CMM is configured through `ldo_process_config()` method like all other layers on MDS. There are two commands that are used by CMM - `LCFG_SETUP` and `LCFG_MDC_ADD`. The first command sets the local index and the second one adds MDC targets.

2.4 CMM-MDC

CMM maintains data about all other MDS and communicates with them. The MDC will be used for each server and CMM have to manage all MDC like LMV on client does. The initial approach is to use old MDC OBD as real client device and create new MDC md_device which will wrap new md API to old OBD API. This MDC md_device will be able to replace MDC OBD.

2.4.1 MDC handling

There are fields in cmm_device to support the handling of MDC devices:

`__u32 cmm_local_num` - index of current MDS;

`__u32 cmm_tgt_total` - number of controlled MDC;

`__u32 cmm_tgt_active` - number of active MDC;

`struct list_head cmm_targets` - CMM support the list of MDC and process through that list to add new MDC, find proper one and remove MDC from the list. List is protected by spinlock, because MDC can be added dynamically.

2.4.2 MDC Configuration

The MDC is added to CMM by processing config from MGS in ldo_process_config. The CMM has `cmm_add_mdc()` method to handle this.

`cmm_add_mdc()` does the following:

1. Allocate and initialize new MDC device;
2. Invoke `ldo_process_config()` on just created device with corresponding struct lustre_cfg;
3. add new device in list of MDC, increase counter and take reference on CMM device.

2.5 CMM device type operations

\begin{lstlisting}
static struct lu_device_type_operations cmm_device_type_ops = {
.ldto_init = cmm_type_init,
.ldto_fini = cmm_type_fini,
.ldto_device_alloc = cmm_device_alloc,
.ldto_device_free = cmm_device_free,
.ldto_device_init = cmm_device_init,
.ldto_device_fini = cmm_device_fini
};
\end{lstlisting}

2.5.1 cmm_device_alloc()

\begin{lstlisting}[frame=none]
struct lu_device cmm_device_alloc(struct lu_context *ctx,
struct lu_device_type *ldt,
struct lustre_cfg *cfg);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - current context;

`struct lu_device_type *ldt` - pointer to the device type initialized while module insert;

`struct lustre_cfg *cfg` - set of configuration parameters from MGS.

Return Values: pointer to the allocated device or error code.

Description: Initialize new device. The CMM allocates new device and fill it with initial data.

2.5.2 cmm_device_free()

\lstinline|void cmm_device_free(struct lu_context *ctx, struct lu_device *ld);|

Parameters:

`struct lu_context *ctx` - current context;

`struct lu_device *ld` - pointer to the device.

Return Values: None.

Description: Free the device.

2.5.3 cmm_device_init()

\begin{lstlisting}[frame=none]
int cmm_device_init(struct lu_context *ctx, struct lu_device *ld,
struct lu_device *next);
struct cmm_device *m = lu2cmm_dev(d);
int err = 0;
ENTRY;
INIT_LIST_HEAD(&m->cmm_targets);
m->cmm_tgt_count = 0;
m->cmm_child = lu2md_dev(next);
RETURN(err);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - current context;

`struct lu_device *ld` - pointer to the initialized device;

`struct lu_device *next` - pointer to the child device for this layer;

Return Values: `0` or error code.

Description: Initialization procedure. It called from top-level device when stack of devices is constructed. All values are initialized and child device is connected.

2.5.4 cmm_device_fini()

\begin{lstlisting}[frame=none]
struct lu_device *cmm_device_fini(struct lu_context *ctx,
struct lu_device* ld);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - current context;

`struct lu_device *ld` - pointer to the device;

Return Values: `NULL` or pointer to the child device.

Description: Finalize the device and return its child to continue operation through device stack.

2.6 CMM lu_device operations

\begin{lstlisting}
static struct lu_device_operations cmm_lu_ops = {
.ldo_object_alloc = cmm_object_alloc,
.ldo_process_config = cmm_process_config
};
\end{lstlisting}

2.6.1 cmm_process_config()

\begin{lstlisting}[frame=none]
int cmm_process_config(struct lu_context *ctx, struct lu_device* ld,
struct lustre_cfg *cfg);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct lu_device *ld` - pointer to the device;

`struct lustre_cfg *cfg` - configuration parameters.

Return Values: `0` or error code.

Description: Read configuration parameters and apply changes related to the CMM. Currently they are related to the underlying MDC devices - addition, configuration, etc.

2.6.2 cmm_object_alloc()

\begin{lstlisting}[frame=none]
struct lu_object *cmm_object_alloc(struct lu_context *ctx,
struct lu_object_header *loh,
struct lu_device *ld);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation,

`struct lu_object_header *loh` - header with lu_fid needed for allocation of local or remote object,

`struct lu_device* ld` - pointer to the device;

Return Values: `NULL` or pointer to the new object.

Description: Allocate `cmm_object` as part of the compound MDS object. CMM has two types of object - remote and local. First one is for operations on remote MDS and second one is for local operations. These objects have different sets on operation. The `fld_lookup()` is called to determine the object type - local or remote.

2.7 CMM md_device operations

\begin{lstlisting}
static struct md_device_operations cmm_md_ops = {
.mdo_root_get = cmm_root_get,
.mdo_statfs = cmm_statfs
};
\end{lstlisting}

This set of methods are implemented in CMM just to redirects the request to the child device and return the result.

2.7.1 cmm_root_get()

\begin{lstlisting}[frame=none]
int cmm_root_get(struct lu_context *ctx, struct md_device *md,
struct lu_fid *fid);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_device *md` - CMM md_device;

`struct lu_fid *fid` - lustre FID of the root to be returned.

2.7.2 cmm_statfs()

\begin{lstlisting}[frame=none]
int cmm_statfs(struct lu_context *ctxt, struct md_device *md,
struct kstatfs *sfs);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_device *md` - CMM md_device;

`struct kstatfs *sfs` - structure for the result of the operation.

2.8 CMM lu_object operations

\begin{lstlisting}
static struct lu_object_operations cmm_obj_ops = {
.loo_object_init = cmm_object_init,
.loo_object_free = cmm_object_free,
.loo_object_print = cmm_object_print,
.loo_object_exists = cmm_object_exists
};
\end{lstlisting}

2.8.1 cmm_object_init()

\lstinline|int cmm_object_init(struct lu_context *ctx, struct lu_object *lo);|

Parameters:

`struct lu_context *ctx` - context of operation;

`struct lu_object *lo` - just allocated object to be initialized.

Return Values: `0` or error code.

Description: MDD object is allocated as child for local object and MDC - for remote one.

2.8.2 cmm_object_free()

\lstinline|void cmm_object_free(struct lu_context *ctx, struct lu_object *lo);|

Parameters:

`struct lu_context *ctx` - context of operation;

`struct lu_object *lo` - object to free.

Return Values: None.

Description: Free the `cmm_object` as part of compound object.

2.8.3 cmm_object_print()

\begin{lstlisting}[frame=none]
int cmm_object_print(struct lu_context *ctx, struct seq_file *f,
const struct lu_object *lo);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct seq_file *f` - where to output the CMM object info;

`struct lu_object *lo` - object to be printed.

Description: Print CMM-related information about the given `lu_object`.

2.8.4 cmm_object_exists()

\lstinline|int cmm_object_exists(struct lu_context *ctx, struct lu_object *lo);|

Parameters:

`struct lu_context *ctx` - context of operation;

`struct lu_object *lo` - object to check.

Description:

- Local object: Simple method that calls the same on child object and returns result.
- Remote object: returns true always. All checks will be done on remote server.

2.9 CMM md_object operations

\begin{lstlisting}
static struct lu_object_operations cmm_mo_ops = {
.moo_attr_get = cmm_attr_get,
.moo_attr_set = cmm_attr_set,
.moo_xattr_get = cmm_xattr_get,
.moo_xattr_set = cmm_xattr_set,
.moo_ref_add = cmm_ref_add,
.moo_ref_del = cmm_ref_del,
.moo_object_create = cmm_object_create,
.moo_object_open = cmm_open,
.moo_object_close = cmm_close

};
\end{lstlisting}

The CMM calls a local child object operations while processing these methods and do nothing else. The return values of all operations are results of child's operations. Therefore only function prototypes and parameters are listed below.

The remote object returns `-EINVAL` for all these operations because they are local in nature.

2.9.1 cmm_attr_get()

\begin{lstlisting}[frame=none]
int cmm_attr_get(struct lu_context *ctx, struct md_object *mo,
struct lu_attr *attr);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - the object affected;

`struct lu_attr *attr` - attributes of the object to return.

2.9.2 cmm_attr_set()

\begin{lstlisting}[frame=none]
int cmm_attr_set(struct lu_context *ctx, struct md_object *mo,
struct lu_attr *attr);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - the object affected;

`struct lu_attr *attr` - attributes of the object to set.

2.9.3 cmm_xattr_get()

\begin{lstlisting}[frame=none]
int cmm_xattr_get(struct lu_context *ctx, struct md_object *mo,
void *buf, int buflen, const char *name);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - the object affected;

`void *buf` - buffer with extended attribute;

`int buflen` - length of buffer;

`const char *name` - the name of the EA.

2.9.4 cmm_xattr_set()

\begin{lstlisting}[frame=none]
int cmm_xattr_set(struct lu_context *ctx, struct md_object *mo,
void *buf, int buflen, const char *name);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - the object affected;

`void *buf` - buffer with extended attribute;

`int buflen` - length of buffer;

`const char *name` - the name of the EA.

2.9.5 cmm_ref_add()

\lstinline|int cmm_ref_add(struct lu_context *ctx, struct md_object *mo);|

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - the object for reference addition.

Description: This method is needed to increase `nlink` for object as part of cross-ref operation.

2.9.6 cmm_ref_del()

\lstinline|int cmm_ref_del(struct lu_context *ctx, struct md_object *mo);|

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - object to be unlinked.

Description: Unlink the object. Method is used as part of cross-ref operation.

2.9.7 cmm_object_create()

Prototype:

\begin{lstlisting}[frame=none]
int cmm_object_create(struct lu_context *ctx, struct md_object *mo,
struct lu_attr *attr);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - just allocated object to be created;

`struct lu_attr *attr` - attributes for newly created object.

Description: This operations is called from another MDS as part of cross-ref operation to create only object on persistent storage without name.

2.9.8 cmm_open()

\lstinline|int cmm_open(struct lu_context *ctx, struct md_object *mo, int flags);|

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - object to open;

`int flags` - open flags.

Description: Open the object by its FID. This operation is local in CMM and returns `-ERESTART` if the object is placed on remote MDS. The operations is finished by calling the MDD.

2.9.9 cmm_close()

\lstinline|int cmm_close(struct lu_context *ctx, struct md_object *mo);|

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo` - object to close.

Description: Close the object by calling operation on the local child and return the result.

2.10 CMM directory operations

These operations are different if there is local object and remote one.

\begin{lstlisting}
struct md_dir_operations cmm_dir_ops = {
.mdo_lookup = cmm_lookup,
.mdo_create = cmm_create,
.mdo_link = cmm_link,
.mdo_unlink = cmm_unlink,
.mdo_rename = cmm_rename,
.mdo_rename_tgt = cmm_rename_tgt
};
\end{lstlisting}

2.10.1 cmm_lookup()

Prototype:

\begin{lstlisting}[frame=none]
int cmm_lookup(struct lu_context *ctx, struct md_object *mo,
const char* name, struct lu_fid *lf);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of the operation;

`struct md_object *mo` - md_object of parent directory;

`const char* name` - file name for lookup;

`struct lu_fid *lf` - pointer to the FID to be filled.

Description: Lookup the FID by name - it is the local operation that is passed to the lower layers. This method can be called for remote object in rename operation and should return -EREMOTE

2.10.2 cmm_create()

\begin{lstlisting}[frame=none]
int cmm_create(struct lu_context *ctx, struct md_object *mo_p,
const char *name, struct md_object *mo_c,
const char *path, struct lu_attr *attr);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo_p` - md_object of parent directory;

`struct md_object *mo_c` - md_object for the child to be created on disk;

`const char* name` - file name of the new object;

`struct lu_attr *attr` - attributes of new object;

`const char *path` - path for symlink operation.

Description: This method consist of two parts: inserting the name into the parent directory locally and creating the new object on disk.

1. Local object `mo_c`: do local create by calling the child object operation;
2. Remote object `mo_c`:
   (a) do remote request by calling the `moo_object_create()` operation on child object of `mo_c`;
   (b) do local name insertion into parent directory by calling the `mdo_name_insert()` operation on child object of `mo_p`;

2.10.3 cmm_link()

\begin{lstlisting}[frame=none]
int cmm_link(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, const char *name);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo_p` - parent directory object;

`struct md_object *mo_s` - linked object;

`const char* name` - file name of the new link to the object.

Description: Creates the new name for the existing object. The object can be placed on remote server, so this operations consist of two branches:

1. If the source object is local then normal `mdo_link()` operation is called on the child object.
2. Otherwise:
   (a) the remote call will be done to the proper MDS with `moo_ref_add()` operation on child of `mo_s` object to increase the `nlink`;
   (b) the new name is inserted into parent dir locally by calling `mdo_name_insert()` on child object of `mo_p`.

2.10.4 cmm_unlink()

\begin{lstlisting}[frame=none]
int cmm_unlink(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, const char *name);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo_p` - parent directory object;

`struct md_object *mo_s` - unlinked object;

`const char* name` - file name of the link.

Description: The object to unlink can be on remote MDS so this operations can have remote part:

1. If object is local then call `mdo_unlink()` on child object of `mo_s`.
2. Otherwise:
   (a) call `moo_ref_del()` on child object;
   (b) remove name from directory by calling `mdo_name_destroy()` on child object of `mo_p`.

2.10.5 cmm_rename()

Many objects can be involved in this operation. The splitting to remote/local operations is based on new parent object nature.

\begin{lstlisting}[frame=none]
int cmm_rename(struct lu_context *ctx, struct md_object *mo_po,
struct md_object *mo_pn, struct md_object *mo_s,
const char *s_name, struct md_object *mo_t,
const char *t_name);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object *mo_po` - old parent directory;

`struct md_object_*mo_pn` - new parent directory;

`struct md_object_*mo_s` - source object;

`const char *s_name` - source name;

`struct md_object *mo_t` - target object. It is not NULL if there is such name in target directory already and new parent is local;

`const char *t_name` - target name.

Description: Rename() is a complex function which can have many remote operations. It is called on MDS where **old parent** lives and there are several cases are possible:

Local object operations (new parent is local dir):

mo_pn	mo_t	operations
local	local	`mdd->mdo_rename();`
local	remote	`mdc->moo_ref_del(mo_t);` `mdd->mdo_rename(mo_t=NULL,...);`

Remote object operations table (new parent is remote dir):

mo_pn	mo_t	operations
remote	N/A	`mdc->mdo_rename_tgt(mo_pn, t_name, mo_s, mo_t=NULL);` `mdd->moo_name_destroy(mo_po, s_name);`

*blue - the remote operations.

To support rename() the MD API allow some of parameters be NULL and properly checked. Also there is one additional command in API - see below.

2.10.6 cmm_rename_tgt()

The part of rename() if new parent directory is place on remote server. This operation is sent to that server to proceed. This operation can also have the remote part - if `mo_t` exists and placed on other MDS.

\begin{lstlisting}[frame=none]
int cmm_rename_tgt(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, struct md_object *mo_t,
const char *t_name);
\end{lstlisting}

Parameters:

`struct lu_context *ctx` - context of operation;

`struct md_object_*mo_pn` - new parent directory;

`struct md_object_*mo_s` - source object;

`struct md_object *mo_t` - target object. It is not NULL if there is such name in target directory already;

`const char *t_name` - target name.

Description: This command means that only target part of rename is needed - destroy the target name (or change fid only), insert name with new fid, unlink the old object if exists.

If old name exists and referenced object is on remote MDS, then operation will be split onto remote and local parts one more time.

2.11 CMM-MDC operations

CMM-MDC is organized as new MD API -compatible device. It supports the reduced sets of operations listed below. The MDC OBD methods are used and some of them should be updated to handle NULL names correctly.

2.11.1 MDC basic structures

\begin{lstlisting}
struct mdc_cli_desc {
struct obd_connect_data cl_conn_data;
struct obd_uuid cl_srv_uuid;
struct obd_uuid cl_cli_uuid;
struct obd_export *cl_exp;
};
struct mdc_device {
struct md_device mc_md_dev;
struct list_head mc_linkage;
__u32 mc_num;
struct mdc_cli_desc mc_desc;
};
\end{lstlisting}

The `mdc_device` consist of md_device structure, linkage to the list of MDC devices in CMM, server index and description of the network part. Currently MDC is connected to the old MDC OBD.

2.11.2 MDC device type operations

\begin{lstlisting}
static struct lu_device_type_operations mdc_device_type_ops = {
.ldto_init = mdc_type_init,
.ldto_fini = mdc_type_fini,
.ldto_device_alloc = mdc_device_alloc,
.ldto_device_free = mdc_device_free,
.ldto_device_init = mdc_device_init,
.ldto_device_fini = mdc_device_fini
};
\end{lstlisting}

Only `mdc_device_alloc/free` are not empty and are used to allocate/free the needed structures

2.11.3 MDC lu_device operations

\begin{lstlisting}
static struct lu_device_operations mdc_lu_ops = {
.ldo_object_alloc = mdc_object_alloc,
.ldo_object_free = mdc_object_free,
.ldo_process_config = mdc_process_config
};
\end{lstlisting}

In this set the `process_config()` is important method. It is invoked when new MDC is added and there is the connection to the MDC OBD is established.

2.11.4 MDC md_device operations

This set of operations is not defined for CMM_MDC due to local nature of all methods.

2.11.5 MDC lu_object operations

\begin{lstlisting}
static struct lu_object_operations mdc_obj_ops = {
.loo_object_init = mdc_object_init,
.loo_object_print = mdc_object_print,
.loo_object_exists = mdc_object_exists
};
\end{lstlisting}

2.11.6 MDC md_object operations

\begin{lstlisting}
static struct md_object_operations mdc_mo_ops = {
.moo_ref_add = mdc_ref_add,

.moo_ref_del = mdc_ref_del,
.moo_object_create = mdc_object_create,
};
\end{lstlisting}
These methods prepare the all needed data and send request to the MDC OBD

2.11.7 MDC directory operations

\begin{lstlisting}
struct md_dir_operations mdc_dir_ops = {
.mdo_rename = mdc_rename_tgt,
};
\end{lstlisting}
Only `rename_tgt()` makes sense here, all other directory operations are local.

2.12 Error handling

Error handling in CMM is different from 1 MDS setup in case of cross-ref operations. There are two possible situations:

1. Remote command failed

   In this case the error code is just returned to the top layer

2. Local command failed

   This case means that the remote part is done successfully already, therefore it must be reverted back by sending another RPC. After it is complete the error code is returned. To reduce the effect of this some checks may be done before doing the remote part of operation.

2.13 Recovery

The existence of CMM will not break recovery things but adds new recovery issues due to occurrence of the MDS-MDS connection. There is separate DLD for this. Current design doesn't cover the rollback functionality also.

3 Use

THIS IS MANDATORY SECTION, IT SHOULD CONTAIN "HOW TO USE OR HOW TO CHECK NEW FUNCTIONALITY". IT IS NATURALLY TO USE IT AS DESIGN FOR UNIT OR SANITY TESTS.CASES

3.1 Cross-ref operations handling

The CMM is used as a layer which communicates with other MDS. For any completely local operation the CMM acts as transparent layer between MDT and MDD. The real job is done for cross-ref operations. CMM split them into local and remote parts and call remote one at first.

3.1.1 Object allocation

While new object allocation, the request to fld will be done and the different methods will assigned to that object in local and remote case. The MDT will call needed operation and proper cmm object will be used automatically to proceed locally or remotely.

3.1.2 Example of cross-ref operation handling

Let's review the link operation:
the helper is defined that call operation based on child type:

```
static inline int mdo_link(const struct lu_context *cx, struct md_object *m_p,
                           struct md_object *m_s, const char *name)
{
        return m_s->mo_ops->mdo_link(cx, m_p, m_s, name);
}
```

Therefore depending on type of m_s - local or remote the needed method will be called:

```
int cml_link(struct lu_context *ctx, struct md_object *mo_p,
             struct md_object *mo_s, const char *name)
{
        struct cmm_object *cmm_s = md2cmm_obj(mo_s);
        struct md_object  *lo    = cmm2child_obj(md2cmm_obj(mo_p));
        int rc;
        ENTRY;

        rc = mdo_link(ctx, lo, cmm2child_obj(cmm_s), name);
        RETURN(rc);
}
```

Local operations will call the same on MDD and return the result.

```
int cmr_link(struct lu_context *ctx, struct md_object *mo_p,
             struct md_object *mo_s, const char *name)
{
```

```
        struct cmm_object *cmm_s = md2cmm_obj(mo_s);
        struct md_object  *lo    = cmm2child_obj(md2cmm_obj(mo_p));
        struct lu_fid *fid = lu_object_fid(c->mo_lu);
        struct md_object *rem = cmm2child_obj(cmm_s);
        ENTRY;
        /* remote object link and local name insert */
        rc = mo_ref_add(ctx, rem, attr);
        if (rc == 0) {
                rc = mdo_name_insert(ctx, lo, name, fid);
        }
        RETURN(rc);
}
```

Cross-ref operation is more complex. At first the remote part should be done. If it is successful then local part will be called.

3.2 Unit tests

3.2.1 Layering

The MDS works with CMM and without it. All tests that are passed w/o CMM should pass with it in 1 MDS configuration.

3.2.2 Correctness

All operations are invoked so that all possible paths are chosen. The result is checked.

3.2.3 Error handling

For each cross-ref operation the one of the possible paths simulate error. There should be not partially completed operations and all changes should be reverted back. To check this the operation is repeated and should be successful.

To simulate errors there can be special layer introduced which just pass through all command but has ability to return the error code for any of them by request.

4 Logic

MANDATORY SECTION, MOREOVER, IT IS VERY IMPORTANT. IT SHOULD CONTAIN "HOW TO IMPLEMENT NEW FUNCTIONALITY TO MEET REQUIREMENTS"SPECIFICATION.

The CMM API has many methods which call only the same method on the child object and return the result. These methods were listed in Functional Specification and omitted here due to their simplicity. The methods below are most complex in CMM API - cross-ref operations.

4.1 CMM structures and helpers

\begin{lstlisting}
struct cmm_device {
struct md_device cmm_md_dev;
/* underlaying device in MDS stack, usually MDD */
struct md_device *cmm_child;
/* MDC-related stuff */
__u32 cmm_local_num;
__u32 cmm_tgt_count;
struct list_head cmm_targets;
};
struct cmm_object {
struct md_object cmo_obj;
/* mds number where object is placed */
__u32 cmo_num;
};
\end{lstlisting}

4.1.1 get proper child device for a new object

\begin{lstlisting}
/* get child device by mdsnum */
struct lu_device *cmm_get_child(struct cmm_device *d, __u32 num)
{
struct lu_device *next = NULL;
if (likely(num == d->cmm_local_num)) {
next = &d->cmm_child->md_lu_dev;
} else {
struct mdc_device *mdc;
list_for_each_entry(mdc, &d->cmm_targets, mc_linkage) {
if (mdc->mc_num == num) {
next = mdc2lu_dev(mdc);
break;
}
}
}
return next;
}
\end{lstlisting}

4.2 CMM device type operations

4.2.1 cmm_device_init()

\begin{lstlisting}
int cmm_device_init(struct lu_context *ctx, struct lu_object *lo)

```
{
}
\end{lstlisting}
```

4.3 CMM lu_device operations

4.3.1 cmm_object_alloc()

```
\begin{lstlisting}
    struct lu_object *cmm_object_alloc(struct lu_context *ctx,
    struct lu_device *ld,
    struct lu_object_header *loh)
    {
    struct cmm_object *co;
    struct lu_fid *fid = loh->loh_fid;
    mdsnum = cmm_fid_lookup(fid);
    if (is_local(mdsnum)) {
    struct cmm_local *cl;
    OBD_ALLOC_PTR(cl);
    ...
    co = &cl->cl_obj;
    co->cmo_obj.mo_ops = &cmm_mo_local_ops;
    co->cmo_obj.mo_dir_ops = &cmm_dir_local_ops;
    co->cmo_obj.mo_lu.lo_ops = &cmm_local_obj_ops;
    } else {
    struct cmm_remote *cr;
    OBD_ALLOC_PTR(cr);
    ...
    co = &cr->cr_obj;
    co->cmo_obj.mo_ops = &cmm_mo_remote_ops;
    co->cmo_obj.mo_dir_ops = &cmm_dir_remote_ops;
    co->cmo_obj.mo_lu.lo_ops = &cmm_remote_obj_ops;
    }
    if (co != NULL) {
    lo = &co->cmo_obj.mo_lu;
    lu_object_init(lo, NULL, ld);
    } else
    lo = NULL;
    RETURN(lo);
    }
    \end{lstlisting}
```

4.4 CMM directory operations

4.4.1 cmm_create()

\begin{lstlisting}
int cml_create(struct lu_context *ctx,
struct md_object *mo_p, const char *name,
struct md_object *mo_c, struct lu_attr *attr)
{
struct cmm_object *cmm_c = md2cmm_obj(mo_c);
struct md_object *lo = cmm2child_obj(md2cmm_obj(mo_p));
int rc;
ENTRY;
/* fully local mkdir */
rc = lo->mo_dir_ops->mdo_mkdir(ctx, lo, name,
cmm2child_obj(cmm_c), attr);
RETURN(rc);
}
\end{lstlisting}
Local object:
\begin{lstlisting}
int cml_create(struct lu_context *ctx,
struct md_object *mo_p, const char *name,
struct md_object *mo_c, struct lu_attr *attr)
{
struct cmm_object *cmm_c = md2cmm_obj(mo_c);
struct md_object *lo = cmm2child_obj(md2cmm_obj(mo_p));
int rc;
ENTRY;
/* fully local mkdir */
rc = lo->mo_dir_ops->mdo_mkdir(ctx, lo, name,
cmm2child_obj(cmm_c), attr);
RETURN(rc);
}
\end{lstlisting}
Cross-ref object:
\begin{lstlisting}
int cmr_create(struct lu_context *ctx,
struct md_object *mo_p, const char *name,
struct md_object *mo_c, struct lu_attr *attr)
{
struct cmm_object *cmm_c = md2cmm_obj(mo_c);
struct md_object *lo = cmm2child_obj(md2cmm_obj(mo_p));
int rc;
const struct lu_fid *fid = lu_object_fid(&c->mo_lu);
struct md_object *rem = cmm2child_obj(cmm_c);

ENTRY;
/* remote object creation and local name insert */
rc = rem->mo_ops->moo_object_create(ctx, rem, attr);
if (rc == 0) {
rc = lo->mo_dir_ops->mdo_name_insert(ctx, lo,
name, fid,
attr);
}
RETURN(rc);
}
\end{lstlisting}

4.4.2 cmm_link()

Local operation:

\begin{lstlisting}
int cml_link(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, const char *name)
{
struct cmm_object *cmm_s = md2cmm_obj(mo_s);
struct md_object *lo = cmm2child_obj(md2cmm_obj(mo_p));
int rc;
ENTRY;
rc = lo->mo_dir_ops->mdo_link(ctx, lo,
cmm2child_obj(cmm_s),
name);
RETURN(rc);
}
\end{lstlisting}

Cross-ref operation:

\begin{lstlisting}
int cmr_link(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, const char *name)
{
struct cmm_object *cmm_s = md2cmm_obj(mo_s);
struct md_object *lo = cmm2child_obj(md2cmm_obj(mo_p));
int rc;
const struct lu_fid *fid = lu_object_fid(&c->mo_lu);
struct md_object *rem = cmm2child_obj(cmm_s);
ENTRY;
/* remote object link and local name insert */
rc = rem->mo_ops->moo_ref_add(ctx, rem, attr);
if (rc == 0) {
rc = lo->mo_dir_ops->mdo_name_insert(ctx, lo,
name, fid,
attr);

}
RETURN(rc);
}
\end{lstlisting}

4.4.3 cmm_unlink()

Local operation:

\begin{lstlisting}
int cml_unlink(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, const char *name)
{
struct cmm_object *cmm_s = md2cmm_obj(mo_s);
struct md_object *lo = cmm2child_obj(md2cmm_obj(mo_p));
int rc;
ENTRY;
rc = lo->mo_dir_ops->mdo_unlink(ctx, lo,
cmm2child_obj(cmm_s),
const char *name);
RETURN(rc);
}
\end{lstlisting}

Cross-ref operation:

\begin{lstlisting}
int cmr_unlink(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, const char *name)
{
struct cmm_object *cmm_s = md2cmm_obj(mo_s);
struct md_object *lo = cmm2child_obj(md2cmm_obj(mo_p));
const struct lu_fid *fid = lu_object_fid(&c->mo_lu);
struct md_object *rem = cmm2child_obj(cmm_s);
int rc;
ENTRY;
/* remote object unlink and local name delete */
rc = rem->mo_ops->moo_ref_del(ctx, rem, attr);
if (rc == 0) {
rc = lo->mo_dir_ops->mdo_name_destroy(ctx, lo,
name, fid,
attr);
}
RETURN(rc);
}
\end{lstlisting}

4.4.4 cmm_rename()

The new parent is local:

```
\begin{lstlisting}
int cml_rename(struct lu_context *ctx, struct md_object *mo_po,
struct md_object *mo_pn, const struct lu_fid *lf,
const char *s_name, struct md_object *mo_t,
const char *t_name);
{
struct md_object *c_po = cmm2child_obj(md2cmm_obj(mo_po));
struct md_object *c_pn = cmm2child_obj(md2cmm_obj(mo_pn));
int rc;
ENTRY;
if (mo_t == NULL) {
rc = mdo_rename(ctx, c_po, c_pn, lf,
s_name, NULL, t_name);
} else {
struct md_object *c_t = cmm2child_obj(md2cmm_obj(mo_t));
if (cmm_is_local_obj(md2cmm_obj(mo_t))) {
rc = mdo_rename(ctx, c_po, c_pn, lf,
s_name, c_t, t_name);
} else {
/* remote object */
rc = moo_ref_del(ctx, c_t, attr);
rc = mdo_rename(ctx, c_po, c_pn, lf,
s_name, NULL, t_name);
}
}
RETURN(rc);
}
\end{lstlisting}
```

The new parent is remote:

```
\begin{lstlisting}
int cmr_rename(struct lu_context *ctx, struct md_object *mo_po,
struct md_object *mo_pn, const struct lu_fid *lf,
const char *s_name, struct md_object *mo_t,
const char *t_name);
{
struct md_object *c_po = cmm2child_obj(md2cmm_obj(mo_po));
struct md_object *c_pn = cmm2child_obj(md2cmm_obj(mo_pn));
struct md_object *c_t;
int rc;
ENTRY;
rc = mdo_rename_tgt(ctx, c_pn, NULL, lf, t_name);
if (rc == 0)
rc = mdo_name_destroy(ctx, c_po, s_name);
```

```
}
RETURN(rc);
}
\end{lstlisting}
```

4.4.5 cmm_rename_tgt()

```
\begin{lstlisting}
    int cml_rename_tgt(struct lu_context *ctx, struct md_object *mo_p,
    struct md_object *mo_t, const struct lu_fid *lf,
    const char *t_name);
    {
    struct md_object *c_p = cmm2child_obj(md2cmm_obj(mo_p));
    struct md_object *c_t = cmm2child_obj(md2cmm_obj(mo_p));
    int rc;
    ENTRY;
    rc = mdo_rename_tgt(ctx, c_p, c_t, lf, t_name);
    RETURN(rc);
    }
    \end{lstlisting}
    \begin{lstlisting}
    int cmr_rename_tgt(struct lu_context *ctx, struct md_object *mo_p,
    struct md_object *mo_t, const struct lu_fid *lf,
    const char *t_name);
    {
    struct md_object *c_p = cmm2child_obj(md2cmm_obj(mo_p));
    struct md_object *c_t = cmm2child_obj(md2cmm_obj(mo_t));
    int rc;
    ENTRY;
    rc = moo_ref_del(ctx, c_t);
    if (rc == 0)
    rc = mdo_rename_tgt(ctx, c_p, NULL, lf, t_name);
    RETURN(rc);
    }
    \end{lstlisting}
```

4.5 CMM md_object operations

4.5.1 cmm_open()

The comment is needed here about `-ERESTART` return code. When the current MDS cannot provide intent-related data it returns just lookup data with `-ERESTART` code. Therefore a client will do additional RPC to the other MDS.

```
    \begin{lstlisting}
    int cmm_open(struct lu_context *ctx, struct md_object *mo, int flags)
    {
```

```
struct cmm_object *cmo = md2cmm_obj(mo);
int rc;
ENTRY;
if (!cmm_is_local_obj(cmo))
rc = -ERESTART;
} else {
rc = lo->mo_ops->moo_open(ctx, cmm2child_obj(cmo));
}
RETURN(rc);
}
\end{lstlisting}
```

4.6 CMM-MDC operations

4.6.1 mdc_process_config()

```
\begin{lstlisting}
    int mdc_add_obd(struct mdc_device *mc, struct lustre_cfg *cfg) {
    struct mdc_cli_desc *desc = &mc->mc_desc;
    struct obd_device *mdc, *mdt;
    const char *srv = lustre_cfg_string(cfg, 0);
    const char *uuid_str = lustre_cfg_string(cfg, 1);
    const char *index = lustre_cfg_string(cfg, 2);
    struct obd_uuid *uuid = &desc->uuid;
    int rc = 0;
    ENTRY;
    mdt = class_name2obd(srv);
    if (mdt == NULL) {
    CERROR("No such OBD %s\n", srv);
    RETURN(-EINVAL);
    }
    obd_str2uuid(uuid, uuid_str);
    mdc = class_find_client_obd(uuid, LUSTRE_MDC_NAME,
    &mdt->obd_uuid);
    if (!mdc) {
    CERROR("Can't find OBD connected to %s\n", uuid_str);
    rc = -ENOENT;
    } else if (!mdc->obd_set_up) {
    CERROR("target %s not set up\n", mdc->obd_name);
    rc = -EINVAL;
    } else {
    struct lustre_handle conn = {0, };
    CDEBUG(D_CONFIG, "connect to %s(%s)\n",
    mdc->obd_name, mdc->obd_uuid.uuid);
    rc = obd_connect(&conn, mdc, &mdt->obd_uuid, NULL);
    if (rc) {
    CERROR("target %s connect error %d\n",
    mdc->obd_name, rc);
    } else {
    desc->cl_exp = class_conn2export(&conn);
    mc->mc_num = simple_strtol(index, NULL, 10);
    }
    }
    RETURN(rc);
    }
    int mdc_process_config(struct lu_context *ctx,
    struct lu_device *ld, struct lustre_cfg *cfg)
    {
```

```
struct mdc_device *mc = lu2mdc_dev(ld);
int rc;
ENTRY;
switch (cfg->lcfg_command) {
case LCFG_ADD_MDC:
rc = mdc_add_obd(mc, cfg);
break;
default:
rc = -EOPNOTSUPP;
}
RETURN(rc);
}
\end{lstlisting}
```

4.6.2 mdc_object_create()

```
\begin{lstlisting}
    int mdc_object_create(struct lu_context *ctx, struct md_object *mo,
    struct lu_attr *attr)
    {
    struct mdc_device *mc = md2mdc_dev(md_device_get(mo));
    struct obd_export *exp = mc->mc_desc.cl_exp;
    struct ptlrpc_request *req;
    struct md_op_data *op_data = &md2mdc_obj(mo)->op_data;
    int rc;
    ENTRY;
    op_data->fid1 = *lu_object_fid(mo->mo_lu);
    op_data->fid2 = { 0 };
    op_data->mod_time = attr->la_mtime;
    op_data->name = NULL;
    op_data->namelen = 0;
    rc = md_create(exp, op_data, NULL, 0, attr->la_mode,
    attr->la_gid, 0, 0, &req);
    RETURN(rc);
    }
    \end{lstlisting}
```

4.6.3 mdc_ref_add

```
\begin{lstlisting}
    int mdc_ref_add(struct lu_context *ctx, struct md_object *mo)
    {
    struct mdc_device *mc = md2mdc_dev(md_device_get(mo));
    struct obd_export *exp = mc->mc_desc.cl_exp;
    struct ptlrpc_request *req;
    struct md_op_data *op_data = &md2mdc_obj(mo)->op_data;{
```

```
int rc;
ENTRY;
op_data->fid1 = *lu_object_fid(&mo->mo_lu);
op_data->fid2 = { 0 };
op_data->mod_time = attr->la_mtime;
op_data->name = NULL;
op_data->namelen = 0;
ENTRY;
rc = md_link(exp, &op_data, &req);
RETURN(rc);
}
\end{lstlisting}
```

4.6.4 mdc_ref_del()

```
\begin{lstlisting}
int mdc_ref_del(struct lu_context *ctx, struct md_object *mo)
{
struct mdc_device *mc = md2mdc_dev(md_device_get(mo));
struct obd_export *exp = mc->mc_desc.cl_exp;
struct ptlrpc_request *req;
struct md_op_data op_data = {
.fid1 = *lu_object_fid(&mo->mo_lu),
.fid2 = { 0 },
.mod_time = attr->la_mtime,
.name = NULL,
.namelen = 0,
};
int rc;
ENTRY;
rc = md_unlink(exp, &op_data, &req);
RETURN(rc);
}
\end{lstlisting}
```

4.6.5 mdc_rename_tgt()

```
\begin{lstlisting}
int mdc_rename_tgt(struct lu_context *ctx, struct md_object *mo_p,
struct md_object *mo_s, struct md_object *mo_t,
const char *t_name);
{
struct mdc_device *mc = md2mdc_dev(md_device_get(mo_p));
struct mdc_object *c_s = md2cmm_obj(mo_s);
struct md_op_data op_data = {
.fid1 = *lu_object_fid(&mo_p->mo_lu),
```

```
.fid2 = *lu_object_fid(&mo_s->mo_lu),
.mod_time = attr->la_mtime,
};
int rc;
ENTRY;
rc = md_rename(exp, &op_data, NULL, 0, t_name,
strlen(t_name), &req);
RETURN(rc);
}
\end{lstlisting}
```

5 State management

5.1 Scalability & performance

MDS layering changes are initiated in view of clear design and code. It shouldn't hurt performance and scalability. Nevertheless the appearance of remote processing stage in operation handling will affect the performance and there is place for further improvements and optimization.

5.2 Recovery changes

The CMM doesn't affect the recovery. The only thing is the rollback but it is the scope of separate DLD.

5.3 Protocol changes

The CMM produce 'partial' operations like create object w/o name, add/del reference on object, etc. These operations can be done with using the existent functionality in MDC-MDT with small modifications:

- while reint_create the name may not exists;
- while link/unlink the name also can be NULL;
- open can be done by fid.

This affects not protocol itself but request handling in MDT and request preparation in MDC

5.4 API changes

The MD API is extended to support all 'partial' operations:

1. md_object_operations
    - moo_object_create;

- moo_ref_add;
- moo_ref_del;

2. md_dir_operations

   - mdo_name_insert;
   - mdo_name_destroy;
   - mdo_rename_tgt;