Metadata API DLD

Yury Umanets

9th February 2005

Contents

1 Functional specification

1.1 Data types and their purpose

In order to provide required functionality (see Metadata API HLD), along with methods belong to described API, there is also set of related structures. They contain different types of information, but roughly may be splitted onto three types:

- information presenting actions which should be performed against some Lustre object (file, directory, etc). This kind of structure contains operation tag and any related data supplied by system, which may be needed during operation (credits, etc).
- information presenting object action should be performed against. This kind of structure contains object identifier and any object related information.
- another structures, needed for connecting metadata stack to another Lustre subsystems, like LDLM for providing clients cache consistency, data stack, etc.

Exact structures definition is the following:

```
struct lcontext {
    intent_t          intent;    /* operation intent */
    credits_t         credits;   /* credits (uid/gid) */
    attrs_t           attrs;     /* type, access mode, etc. */
    void             *private;   /* opaque pointer */
};

typedef struct lcontext lcontext_t;
```

Main purpose of this structure is to bring to lustre what system wants to be performed in it. Many metadata operations will have it as one of arguments. This structure is prepared in system layer (VFS in Linux) and passed through the all metadata layers used on client.

Next type definition is needed for passing it to all MD layer functions as pointer to object created and managed on that MD layer. As layering says us that only owner should know details of own object presentation, this type has only common fields for all MD layers.

```
typedef struct lnode lnode_t;

struct lnode {
    /*
     * common fields, all MD layers should have the same
     * in here to provide the same fields offsets and make
     * them accesible without type casting.
     */
    struct lustre_id    id;       /* object id */
    lnode_t            *entity;   /* object entity for next
                                     MD layer */
    lcallback_t        *lrv_cb;   /* lock revocation callback */
    void               *lrv_data; /* lock revocation data */
```

```
        struct list_head    list;     /* used for linking node into
                                         different list */
        struct semaphore    sem;      /* sem for protecting fields */
        spin_lock_t         lock;     /* lock for protecting fields */
        atomic_t            refcount; /* node refcount */
        struct obd_export  *exp;      /* export node is tied to */
        /*
         * MD layer specific fields, each layer will put them
         * here.
         */
        ...
};
```

Each MD layer may have much more fields for presenting object. For instance LMV is complex thing and its objects are also complex. Thus, this lnode structure is only how **LLITE** sees object and each layer should cast passed lnode pointer to own object type to access the rest of fields. This is like the "tip of the iceberg". In fact for most of cases MD layers will not need to cast it all, as most used fields are visiable already.

Next type definition is needed for maintaining clients cache consistency via LDLM locks revocation. It looks lie the following:

```
typedef (lcallback_t *)(struct ldlm_lock *lock,
                        void *cbdata, int flags);
```

Each MD layer installs own lock revocation callback to be notified when lock gets revoked and do appropriate actions on it. For instance invalidate some cache actual on the layer.

Important thing about *locks revocation callback* using is how to use it. It may function like interrupt handlers in Linux, that is only as notificator (nothing hard is done in it). Or it may function such a way that any kind of operation (including blocking, or hard cpu bound one) may be performed in it. There are advantages and downsides in both ways of implementation. They are the following:

- using it as notifier only is more smart and will not cause such an unpleasant behavior as client evicting if callback will do something really hard (like invalidating enormous amount of local cache).
- from another point of view, if we have to perform something really hard anyway, we should maintain kind of asynchronous behavior using threads, etc. This is more complex from developing and supporting point of view. Also I'm not sure if it will play at all, because it depends on LDLM implementation. LDLM may function the way like if callback returned, local caches are counted invalidated and control may process, what is wrong, because invalidating is in progress yet. Thus, the simplest way to implement things right is to design cache invalidation routines carefully to not use algorithms with long execution time.

1.2 Layering requirements

1.2.1 Working with LDLM locks

Layering says us, that all work with locks should be implemented in last MD layer (MDC) as this is in its patch. For instance checking for lock validness in *revalidate()* control path should be done using this principle. In it, *ll_revalidate_it()* in LLITE will call *lmv_revalidate()* method and so on until last MD layer is reached. Last MD layer (MDC) then checks if asked lock is still valid and if so - no re-lookup is needed. The same is held true for locks taking and dropping. Both these operations should be delegated to MDC.

One more important thing here is that all locks are taken on underneath of operations like *lookup()*, *create()*, etc. and there is no high level dedicated function for taking locks. This makes API clear and simple, easy for maintain and less tied with system related frameworks like VFS, what makes it easy to port.

1.2.2 Nested LDLM locks

There is the issue in LMV configurations, which may be called *locks nesting*. It is allusive to the following:

In the cases when entry name and its inode lie on the different MDS, and LLITE wants to resolve the name, we have to take two locks: first one is LOOKUP, taken onto MDS1 when lookup the name to resolve it to fid, and second one is UPDATE taken onto MDS2 to fetch inode's attributes using the fid.

This issue should be taken into account from layering standing point, as only MDC is locks master and LMV in this case may be like a driver - what MDC target operation should be forwarded to. See use cases for more details.

1.2.3 Inode representation

Layering says, that each MD layer should have own inode representation (object). If we will imagine the chain of such am objects starting from LLITE, it will look like the following:

```
                              |->mdc_node
inode->ll_info->lmv_node      |->mdc_node
                              |->mdc_node
```

Thus object (inode) on each layer should have pointer to object on next layer to support such a stacking. And calling functions for next MD layer (forwarding) should pass correct inode representation for next MD layer. For instance when LLITE calls *md_read_lnode()* for next MD layer, say LMV, it should pass lnode entity initialized by LMV in *md_read_lnode()* time and so on.

1.2.4 Node refcounting

There is set of fields in *lnode* devoted to proper node refcounting. In simplest case it is not needed, as refcounting is performed on system layer (VFS in Linux, using inode refcounting). Thus, in lnode read time refcount to it should be increased and decreased when user of lnode (LLITE here) thinks it is not needed anymore. There are two fucntions, maintaining proper refcounting. They are *md_get_lnode()* and *md_put_lnode()*.

Besides, some MD layers may need it also when linking nodes into some tracing, debugging, etc. lists. In this case, node will be destroyed when last reference is gone, starting from the layer denoted by node passed to *md_put_lnode()*. Thus, in *clear_inode()* time, node will not be destroyed as it is busy. Instead its user will take care of it.

See first chapters of "Logic specificatoin" for examples of implementation.

1.3 Metadata API

Metadata API contains the following methods:

1.3.1 create() method

Prototype

```
int
md_create(struct obd_export *exp, lnode_t *parent,
          struct qstr *name, lcontext_t *ct);
```

Description

This function creates special object (symlink, pipe, etc.) or directory under passed @parent object with passed @name on MDS server. Credits and any other information is taken from passed @ct.

Parameters

- **exp** - next MD layer export.
- **parent** - parent object, child object with @name should be created in.
- **name** - name of the object to be created.
- **ct** - operation context. Credits, etc. are taken from it.

Return value

Returns 0 on success and error code otherwise.

1.3.2 open() method

Prototype

```
void *
md_open(struct obd_export *exp,
        lnode_t *node, lcontext_t *ct);
```

Description

This method is called to open the object denoted by passed @node.

Parameters

- **exp** - next MD layer export.
- **node** - object to be opened.
- **ct** - operation context.

Return value

Returns opaque point to some structure from lower layer. It will be passed back for operations on this opened object.

1.3.3 close() method

Prototype

```
int
md_close(struct obd_export *exp,
         lnode_t *node, void *entity);
```

Description

This method is called to close @entity when it is not needed anymore. @entity is returned by open() method.

Parameters

- **exp** - next MD layer export.
- **node** - object to be closed.
- **entity** - opened object handler to be closed.

Return value

Returns 0 on success and error code otherwise.

1.3.4 getattr() method

Prototype

```
int
md_getattr(struct obd_export *exp,
           lnode_t *node, lcontext_t *ct,
           void *buff, int flags);
```

Description

This function is called to fetch inode attributes and store them in passed @buff. Attributes may be either usual inode attributes like mtime, atime, mode, etc. or inode extended attributes (LOV or MEA). What should be obtained from server is specified by @flags parameter.

Parameters

- **exp** - next MD layer export.
- **node** - object node to fetch attributes.
- **buff** - attributes should be stored here.
- **ct** - operation context.
- **flags** - different flags affecting function behavior. For instance, here may be passed flag saying that we do not need fresh attributes from server, etc. Also type of attributes which should be fetched is specified here. For instance, MEA may be needed by LMV.

Return value

Returns 0 on success and error code otherwise.

1.3.5 lookup() method

Prototype

```
int
md_lookup(struct obd_export *exp, lnode_t *parent,
          struct qstr *name, struct lustre_id *cid,
          lcontext_t *ct);
```

Description

This function is called when @parent-@name pair cannot be found in local cache or invalid and system wants to fetch it from server for using in consequent operations. Found name should be resolved into object id and stored in passed @child.

In order to check if @parent-@name pair is still valid, revalidate() method is called by system. See below its description.

Parameters

- **exp** - next MD layer export.
- **parent** - parent object.
- **name** - name to check.
- **cid** - child object identifier found by name.
- **ct** - operation context.

Return value

Returns 0 of success and error code otherwise.

1.3.6 revalidate() method

Prototype

```
int
md_revalidate(struct obd_export *exp, lnode_t *parent,
              struct qstr *name, lnode_t *child,
              lcontext_t *ct);
```

Description

This method is called to check if pair @parent-@name is still valid or it should be updated from server. This is usually done when @name can be found in local dentry cache and system calls filesystem to make sure that @name is valid. If @name is not valid lookup() will be called to fetch it from server and update local cache. See lookup() description above.

Parameters

- **exp** - next MD layer export.
- **parent** - parent object name should be checked along with it.
- **name** - name to be checked.
- **child** - object name was resolved last time.
- **ct** - operation context.

Return value

Returns 1 if name is valid, 0 if it is not valid and needs to be updated and error code otherwise.

1.3.7 link() method

Prototype

```
int
md_link(struct obd_export *exp, lnode_t *parent,
        struct qstr *name, lnode_t *target,
        lcontext_t *ct);
```

Description

This method is called to create @name of @target in directory @parent.

Parameters

- **exp** - next MD layer export.
- **parent** - parent object name should be created in.
- **name** - name to be created.
- **target** - object name should point to.
- **ct** - operation context.

Return value

Returns 0 on success and error code otherwise.

1.3.8 unlink() method

Prototype

```
int
md_unlink(struct obd_export *exp, lnode_t *parent,
          struct qstr *name, lcontext_t *ct);
```

Description

This method is called to remove @name in directory @parent.

Parameters

- **exp** - next MD layer export.
- **parent** - parent object name should be created in.
- **name** - name to be created.
- **ct** - operation context.

Return value

Returns 0 on success and error code otherwise.

1.3.9 rename() method

Prototype

```
int
md_rename(struct obd_export *exp, lnode_t *old_parent,
          struct qstr *old_name, lnode_t *new_parent,
          struct qstr *new_name, lcontext_t *ct);
```

Description

This method is called to rename @old_name under @old_parent to @new_name under @new_parent.

Parameters

- **exp** - next MD layer export.
- **old_parent** - old parent object.
- **old_name** - old name to be renamed.
- **new_parent** - new parent object.
- **new_name** - new name under @new_parent object.
- **ct** - operation context.

Return value

Returns 0 on success and error code otherwise.

1.3.10 readdir() method

Prototype

```
int
md_readdir(struct obd_export *exp, lnode_t *node,
           void *filep, __u64 *offset, void *buffer,
           filldir_t filldir, lcontext_t *ct);
```

Description

This method is called to read directory entries from opened @entity starting from @offset and store them in @buffer. See open() method for understanding where @entity comes from.

Parameters

- **exp** - next MD layer export.
- **node** - object data will be read from.
- **filep** - opened object handle, data will be read from.
- **offset** - offset data should be read at.
- **buffer** - read data should be stored here.
- **filldir** - function for using it for forming buffer of dir data.
- **ct** - operation context.

Return value

Returns 0 on success and error code otherwise.

1.3.11 read_lnode() method

Prototype

```
lnode_t *
md_read_lnode(struct obd_export *exp, lcontext_t *ct,
              struct lustre_id *id, lcallback_t cb,
              void *cbdata);
```

Description

This method is called first time given inode becoming part of inode cache. The purpose is to fetch minimal set of attributes, allocate needed memory to hold the inode and initialize it. Also, in this phase modules install callbacks.

Parameters

- **exp** - next MD layer export.
- **node** - object to be initialized.
- **ct** - operation context.
- **id** - object identifier.
- **cb** - current layer callback notificator for lock revocation.
- **cbdata** - opaque data needed for @cb.

Return value

Returns 0 on success and error code otherwise.

1.3.12 clear_lnode() method

Prototype

```
int
md_clear_lnode(struct obd_export *exp,
               lnode_t *node);
```

Description

This function is called to release @node initialized by read_lnode() function.

Parameters

- **exp** - next MD layer export.
- **node** - node to be finalized.

Return value

Returns 0 on success and error code otherwise. -EBUSY code is returned if refcount of lnode is not zero and cleanup is skipped.

1.3.13 update_lnode() method

Prototype

```
int
md_update_lnode(struct obd_export *exp,
                lnode_t *node,
                struct lustre_md *md);
```

Description

This method is called when client wants to update lnode by data from server and want to inform lower MD layers of this event. Each MD layer may do whatever it wants to.

Parameters

- **exp** - next MD layer export.
- **node** - node to be updated.
- **md** - lustre object metadata.

Return value

Returns 0 on success and error code otherwise.

1.3.14 revoke_lnode() method

Prototype

```
void
md_revoke_lnode(struct obd_export *exp,
                lnode_t *node, ldlm_lock *lock,
                int flags);
```

Description

This function is called when lower layer finds the need to revoke lock and lets upper layer know that.

Parameters

- **exp** - prev MD layer device.
- **node** - node data is associated with.

1.3.15 get_lnode() method

Prototype

```
void
md_get_lnode(struct obd_export *exp,
             lnode_t *node);
```

Description

This function gets one more reference to passed node. It may be used when MD layer wants to reserve node for some futher using and does not want it gone in *clear_inode()* time.

Parameters

- **exp** - next MD layer device.
- **node** - node data is associated with.

1.3.16 put_lnode() method

Prototype

```
void
md_put_lnode(struct obd_export *exp,
             lnode_t *node);
```

Description

This function drops one reference to passed node and calls its cleanup when last refcount is gone.

Parameters

- **exp** - next MD layer device.
- **node** - node data is associated with.

2 Use cases

Here is how different MD API functions interact with each other. Important thing here is to have clear and not bloated API. Thus, each function should know its main purpose and should not do more work then needed. In these conditions all API functions should be like bricks - elementary functions used for building something complex and useful.

As lustre on client runs stack of MD layers, each layer is interested in having own inode representation - such a layer-nature-dependent object existing on particular layer in correspondence to existing inode in the system. Also each such a representation should be correctly updated on changing its state. Thus, this object should be initialized , finalized, and updated in right time. Here is set of methods for supporting such a state update.

2.1 read_inode() case

This is for initializing per-layer inode representation :

```
ll_read_inode(inode)
    ->lmv_read_lnode(lmv_node)
        ->mdc_read_lnode(mdc_node)
```

Each layer have ability to initialize own inode representation and use it further work.

2.2 clear_inode() case

The same is about clearing inode. It will have the same calling chain to clear inode representation on each layer. It is symmetrical call to *read_lnode()*:

```
ll_clear_inode(inode)
    ->lmv_clear_lnode(lmv_node)
        ->mdc_clear_lnode(mdc_node)
```

2.3 update_inode() case

Updating chain looks like the following:

```
ll_update_inode(inode)
    ->lmv_update_lnode(lmv_node)
        ->mdc_update_lnode(mdc_node)
```

2.4 lookup() case

In opposition to previous subsections here is talking about delegating functionality to different MD layers. Each of them wants to "inject" own behavior in different operations like *lookup()* or *revalidate()* and this way to implement its behavior.

```
ll_lookup_it(dentry)
    ->lmv_lookup(dentry->name, cid)
        ->mdc_lookup(dentry->name, cid)
    ->ll_prep_inode(inode, cid)
        ->iget(cid)
            ->ll_read_lnode(node, cid)
                ->lmv_read_lnode(node, cid)
                    ->mdc_read_lnode(node, cid)
        ->lmv_getattr(node, md)
            ->mdc_getattr(node, md)
        ->ll_update_inode(inode, md)
            ->lmv_update_lnode(node, md)
                ->mdc_update_lnode(node, md)
```

2.5 revalidate() case

Here is calling chain for revalidate case.

```
ll_revalidate_it(parent, dentry)
    ->lmv_revalidate(pnode, dentry->name)
        ->mdc_revalidate(pnode, dentry->name)
```

The rest of functions are used the same way.

3 Logic specification

Here we have full or partial implementatoin for most important and complex parts of MD API. This is few functions on all MD layers:

- lnode live circle related stuff like read_lnode(), clear_lnode(), etc.
- lookup() and revalidate()
- readdir()
- locks revocation handling.

See below for more details.

3.1 Use cases for lnode working functions

As lnode is lustre representation of *struct inode* (in Linux), there are lnode use cases. lnode is a container for object related information. It is initialized/finalized in inode read and clear time and updated when client inode gets updated from server. Some lnodes may do not need updating as they contain statical information, but control is passed to corresponding layer to let it make decision what to do.

3.1.1 get_lnode() and put_lnode() implementation

The purpose of these methods is to manage node usage. This is increase refcount if there is one more user and decrease it when tehre is one less user. When last user is gone - node is going to be cleaned up.

```
static inline void
md_get_lnode(struct obd_export *exp,
             lnode_t *node)
{
    LASSERT(node != NULL);
    atomic_inc(&node->refcount);
}

static inline void
md_put_lnode(struct obd_export *exp,
             lnode_t *node)
{
    LASSERT(node != NULL);
    LASSERT(atomic_read(&node->refcount) > 0);

    if (atomic_dec_and_test(&node->refcount)) {
        CDEBUG(D_INODE, 'last reference to node '
               DLID4' gone - freeing\n',
```

```
                OLID4(&node->id));
        md_clear_lnode(exp, node);
    }
}
```

3.1.2 revoke_lnode() implementation

This function should call lock revocation callback function registered in node in initialization time.

```
static inline void
md_revoke_lnode(struct obd_export *exp, lnode_t *node,
                struct ldlm_lock *lock, int flags)
{
    int rc = 0;

    /*
     * @exp here may be needed for logging,
     * tracing, etc.
     */
    ...
    if (node->lrv_cb) {
        void *data = node->lrv_data;
        rc = node->lrv_cb(lock, data,
                          flags);
    }
    return rc;
}
```

3.1.3 read_lnode() implementation for LLITE

```
static void
ll_read_inode(struct inode *inode, struct lustre_md *md)
{
    struct obd_export *exp = ll_i2sbi(inode)->ll_md_exp;
    struct ll_inode_info *lli = ll_i2info(inode);
    struct lustre_id *id = &md.body->id1;
    lnode_t *node = NULL;
    ENTRY;

    ...

    /*
     * initializing inode's lnode. Passing it local blocking
     * ast callback and inode as data for it.
     */
    lli->lli_node = md_read_lnode(exp, ct, id,
```

```
                                ll_blocking_ast,
                                inode);
    if (lli->lli_node == NULL) {
        CERROR('can't create node for 'DLID4'\n',
               OLID4(id));
        EXIT;
        return;
    }

    ...
    EXIT;
}
```

3.1.4 read_lnode() implementation for LMV

LMV private node related structures may look like the following:

```
struct lmv_obj {
    struct lustre_id    id;       /* id of particular lmv obj */
    unsigned long       size;     /* size of particular lmv obj */
    int                 flags;    /* diff. object flags */
    lnode_t            *entity;   /* next MD layer obj */
    struct lmv_lnode   *master;   /* ref to master LMV node */
};
struct lmv_lnode {
    /* common fields, aligned as for lnode */
    struct lustre_id    id;       /* object's master id */
    lnode_t            *entity;   /* next MD layer master entity */
    lcallback_t         lrv_cb;   /* upper layer lock revocation
                                     notificator. */
    void               *lrv_data; /* useful data for @cb */

    struct list_head    list;     /* used for linking node into
                                     different list */
    struct semaphore    sem;      /* sem for protecting fields */
    spin_lock_t         lock;     /* lock for protecting fields */
    atomic_t            refcount; /* node refcount */
    struct obd_export  *exp;      /* export node is tied to */
    /* LMV specific fields */
    __u32               hashtype; /* hash type used for spreading
                                     names among diff. MDSs */
    int                 objcount; /* number of subobjects */
    struct lmv_obj     *objs;     /* array of subobjects */
};
```

The following is helper LMV methods which used from LMV *read_lnode()* and *clear_lnode()*:

```
static int
lmv_clear_children(struct obd_device *obd,
                   struct lmv_node *node)
{
    int i, objsize, rc = 0;
    struct lmv_obj *objs;
    ENTRY;
    objs = node->objs;
    objsize = node->count *
        sizeof(struct lmv_obj);
    for (i = 0; i < node->count) {
        /* check for self export */
        if (!lmv->tgts[i].exp)
            continue;
        if (objs[i].entity != NULL) {
            md_put_lnode(lmv->tgts[i].exp,
                         objs[i].entity);
            objs[i].entity = NULL;
        }
    }
    OBD_FREE(objs, objsize);
    node->entity = NULL;
    node->objcount = 0;
    node->objs = NULL;
    RETURN(rc);
}
#define same_tgt(id1, id2)       \
    (id_group(id1) == id_group(id2))

#define master_tgt(node, id)     \
    (same_tgt(&node->id, id))
static int
lmv_read_children(struct obd_device *obd,
                  struct lmv_node *node,
                  lcontext *ct)
{
    int flags, i = 0, objsize, rc = 0;
    struct obd_export *tgt_exp;
    struct lmv_obj *objs;
    lnode_t *tgt_entity;
    struct mea mea;
    ENTRY;

    /*
     * initializing fake MEA needed for
     * initializing master.
```

```
     */
    mea.mea_count = 1;
    mea.mea_ids[0] = &node->id;

restart:
    /* initializing new entities */
    objsize = mea.mea_count *
        sizeof(struct lmv_obj);
    OBD_ALLOC(objs, objsize);
    if (objs == NULL)
        RETURN(-ENOMEM);
    memset(objs, 0, objsize);

    node->objs = objs;
    node->objcount = mea.mea_count;
    for (i = 0; i < mea.mea_count) {
        LASSERT(id_fid(&mea.mea_ids[i]));
        tgt_exp = lmv->tgts[i].exp;
        /* initializing child MDC entity */
        tgt_entity = md_read_lnode(tgt_exp, NULL,
                                   &mea.mea_ids[i],
                                   lmv_blocking_ast,
                                   &objs[i]);
        if (tgt_entity == NULL) {
            CERROR('can't initialize entity for %d MDC '
                   'target\n', i);
            GOTO(out_free_objs, rc = -EIO);
        }
        objs[i].id = mea.mea_ids[i];
        objs[i].entity = tgt_entity;
        objs[i].master = node;

        /* check if this is master one and we can read MEA */
        if (master_tgt(node, &mea.mea_ids[i])) {
            flags = OBD_FRESH_ATTR | OBD_MEA_ATTR;
            rc = md_getattr(tgt_exp, tgt_entity,
                            ct, &mea, flags);

            if (rc == 0) {
                /* restart to read the rest of children */
                lmv_clear_children(obd, node);
                goto restart;
            }
            if (rc != -ENODATA) {
                CERROR('can't read MEA for %d MDC '
                       'target\n', i);
```

```
                GOTO(out_free_objs, rc);
            }

            /* -ENODATA case */
            rc = 0;
        }
    }

    node->entity = objs[id_group(&node->id)].entity;
    RETURN(rc);
out_free_objs:
    lmv_clear_children(obd, node);
    return rc;
}
```

The following is LMV implementation of *md_read_lnode()* method. It is quite complex as has to handle many things like MEA readding and transforming LMV accordingly to it.

```
static lnode_t *
lmv_read_lnode(struct obd_export *exp, lcontext_t *ct,
               struct lustre_id *id, lcallback_t lrv_cb,
               void *lrv_data)
{
    struct obd_device *obd = exp->exp_obd;
    struct lmv_obd *lmv = &obd->u.lmv;
    struct lmv_lnode *node;
    int rc = 0, flags;
    ENTRY;

    /* initializing lmv node (basic) */
    node = lmv_create_node(exp, id);
    if (node == NULL)
        RETURN(NULL);

    /* locks revocation callback */
    node->lrv_cb = cb;
    node->lrv_data = cbdata;

    /*
     * initializing children entities. No need to lock
     * it here, nobody has reference yet.
     */
    rc = lmv_read_children(exp->exp_obd, node, ct);
    if (rc) {
        CERROR('can't initialize children entities, '
```

```
                'err %d\n', rc);
            GOTO(free_node, rc);
        }

        /* ganning first ref to node */
        RETURN(md_get_lnode(exp, node));
    free_node:
        lmv_free_node(node);
        RETURN(NULL);
    }
```

And the following is LMV implementation of *md_clear_lnode()* method.

```
    #define get_lmv_node(node) \
        ((struct lmv_node *)node)
    static void
    lmv_clear_lnode(struct obd_export *exp,
                    lnode_t *node)
    {
        struct obd_device *obd = exp->exp_obd;
        struct lmv_lnode *lmv_node;
        ENTRY;
        LASSERT(!atomic_read(&node->refcount));
        lmv_node = get_lmv_node(node);

        /* clear all entities */
        lmv_clear_children(obd, lmv_node);
        lmv_free_node(lmv_node);

        EXIT;
    }
```

3.1.5 read_lnode() implementation for MDC

MDC inode representation object structure looks like the following:

```
    struct mdc_node {
        /* common lnode fields */
        struct lustre_id    id;       /* object's master id */
        lnode_t            *entity;   /* next MD layer master entity */
        lcallback_t         lrv_cb;   /* upper layer lock revocation
                                         notificator. */
        void               *lrv_data; /* useful data for @cb */

        struct list_head    list;     /* used for linking node into
                                         different list */
        struct semaphore    sem;      /* sem for protecting fields */
```

```
    spin_lock_t         lock;     /* lock for protecting fields */
    atomic_t            refcount; /* node refcount */
    struct obd_export  *exp;      /* export node is tied to */

    /* MDC specific fields, seem none? */
    ...
};
```

And here is *mdc_read_lnode()* implementation:

```
#define get_mdc_node(node) \
    ((struct mdc_node *)node)
static lnode *
mdc_read_lnode(struct obd_export *exp, lcontext_t *ct,
               struct lustre_id *id, lcallback_t cb,
               void *cbdata)
{
    struct mdc_node *mdc_node;
    ENTRY;

    mdc_node = mdc_create_node(exp, id);
    if (mdc_node == NULL)
        RETURN(NULL);

    /*
     * setting up lock revocation call back
     * and data for it.
     */
    mdc_node->lrv_cb = lrv_cb;
    mdc_node->lrv_cbdata = lrv_data;

    RETURN(md_get_lnode(exp, node));
}
```

3.1.6 clear_lnode() for LLITE

```
static void
ll_clear_inode(struct inode *inode)
{
    struct ll_inode_info *lli = ll_i2info(inode);
    ENTRY;

    ...

    /*
     * here we use md_put_lnode() instead of
     * direct md_clear_lnode() as there may be
```

```
         * references to node in lower layers.
         */
        md_put_lnode(ll_i2sbi(inode)->ll_md_exp,
                     lli->lli_node);
        EXIT;
    }
```

So, as it may be seen, using this interface, LLITE has ability to inform lower MD layers of creating new inode in it and thus, each lnode may be allocated/released on all MD layers.

As for the rest of MD layers stuff is trivial we omit it for *clear_lnode()* case.

3.1.7 update_lnode() for LLITE

```
static int
ll_update_inode(struct super_block *sb,
                struct inode *inode,
                struct lustre_md *md)
{
    struct ll_inode_info *lli = ll_i2info(inode);
    int rc = 0;
    ENTRY;

    ...

    md_update_lnode(ll_i2sbi(inode)->ll_md_exp,
                    lli->lli_node, md);
    RETURN(rc);
}
```

As for the rest of MD layers stuff is trivial we omit it for *update_lnode()* case.

3.2 Use cases for lookup() and revalidate()

3.2.1 lookup() implementation in LLITE

As it was said in first chapter, all locks work should be done in last MD layer in the stack. Currently, in most of configurations it is MDC, but may be changed. So, LLITE will not issue intent locks itself on lookup path. It will call *lookup()* method of the next MD layer device. The same is held true for *revalidate()*.

The following is *ll_create_inode()* method. It is used for creating new inode having its metadata from server or some cache, does not matter.

```
static struct inode *
ll_create_inode(struct super_block *sb,
                struct lustre_id *id)
{
```

```
    struct inode *inode;
    unsigned long inum;
    ENTRY;
    LASSERT(sb != NULL);
    LASSERT(id != NULL);
    inum = ll_get_inum(id);
    if (!inum)
        RETURN(-ENOSPC);
    /*
     * getting inode (will cause ll_read_lnode()
     * calling)
     */
    RETURN(ll_iget(sb, inum));
}
```

The following method is *ll_prep_inode()*. It is used for creating new inode or updating existing one having its metadata. May be used from *lookup()* or *revalidate()* methods.

```
static struct inode *
ll_prep_inode(struct super_block *sb, struct inode *inode,
              struct lustre_id *id, lcontext_t *ct)
{
    struct ll_inode_info *lli;
    struct lustre_md md;
    int flags, rc = 0;
    ENTRY;

    ...

    if (inode == NULL) {
        /*
         * creating new inode.
         */
        inode = ll_create_inode(sb, id);
    }

    lli = ll_i2info(inode);
    /* getting all attrs from server */
    flags = OBD_FRESH_ATTR | OBD_INODE_ATTR;
    rc = md_getattr(ll_i2md_exp(parent),
                    lli->lli_node, ct,
                    &md, flags);
    if (rc)
        RETURN(ERR_PTR(rc));
```

```
    /* updating inode */
    ll_update_inode(sb, inode, &md);

    ...
    RETURN(inode);
}
```

The following is *ll_lookup_it()* method. It is called when VFS wants to resolve @dentry->d_name into inode. The main purpose of this method is to perform two actions:

- resolve @dentry->d_name into object's lustre_id.
- resolve object's lustre_id into Linux struct inode by creating new inode and filling it by object metadata. This object metadata may be obtained from server using object id. All this is done by *ll_prep_inode()* function.

```
static struct dentry *
ll_lookup_it(struct inode *parent, struct dentry *dentry,
             struct nameidata *nd, lcontext_t *ct,
             int flags)
{
    struct ll_inode_info *lli = ll_i2info(parent);
    struct lustre_id child_id;
    struct inode *inode;
    int rc = 0;
    ENTRY;

    ...

    /* take lock and resolve @name into @child_id */
    rc = md_lookup(ll_i2md_exp(parent), lli->lli_node,
                   dentry->d_name, &child_id, ct);
    if (rc)
        RETURN(rc);

    /* creating new inode */
    inode = ll_prep_inode(parent->i_sb, NULL,
                          &child_id, ct);
    if (!inode)
        RETURN(-ENOSPC);
    if (IS_ERR(inode))
        RETURN(PTR_ERR(inode));

    /* assigning found inode into @dentry */
    dentry->d_inode = inode;
```

```
    ...

    RETURN(rc);
}
```

3.2.2 lookup() implementation in LMV

The main task of LMV layer here as always may be splitted onto two parts:

- choose correct MDC target lookup will be forwarded to. This should take into account also possible -ERESTART. This is the case when during lookup MDS splits dir (on some lookup intent) and LMV is notified of this event to choose correct MDS after split and repeat request.
- take care of own, private info (lmv_node) maintained in the way of being assigned to each inode in VFS. This is important thing, as takes more than half of LMV work.

The following is LMV implementation for lookup() method.

```
#define is_splitted_node(node) \
    (get_lmv_node(node)->objcount > 1 ? 1 : 0)
static int
lmv_lookup(struct obd_export *exp, lnode_t *parent,
           struct qstr *name, struct lustre_id *cid,
           lcontext_t *ct)
{
    struct lmv_node *lmv_parent = get_lmv_node(parent);
    lnode_t *entity = NULL;
    int mds, rc = 0;
    ENTRY;

    ...

repeat:
    /*
     * check if parent has lmv node, that is
     * splitted.
     */
    if (is_splitted_node(parent)) {
        mds = raw_name2idx(lmv_parent->hashtype,
                           lmv_parent->objcount,
                           name);
        entity = lmv_parent->obj[mds].entity;
    } else {
        entity = lmv_parent->entity;
        mds = id_group(lmv_parent->id);
```

```
        }

        /* call lookup for next MD layer device */
        rc = md_lookup(lmv->tgts[mds], entity,
                       name, cid, ct);

        /*
         * handling the case when dir gets splitted
         * while we were in lookup.
         */
        if (rc == -ERESTART) {
            /*
             * protecting transition from non-splitted
             * state to splitted one.
             */
            down(&lmv_parent->sem);

            /* freeing old children (master too) */
            lmv_clear_children(exp->exp_obd,
                               lmv_parent);
            /* re-read children for @parent */
            rc = lmv_read_children(exp->exp_obd,
                                   lmv_parent, ct);
            up(&lmv_parent->sem);
            if (rc) {
                CERROR('can't read node children, '
                       'err %d\n', rc);
                RETURN(rc);
            }

            LASSERT(is_splitted_node(lmv_parent));
            goto repeat;
        }
        if (rc)
            RETURN(rc);
        ...
        RETURN(rc);
    }
```

3.2.3 MDC implementation of lookup() method

MDC mostly takes care about locks and issued RPCs. The following is MDC lookup() implementation.

```
    static int
    mdc_lookup(struct obd_export *exp, lnode_t *parent,
```

```
        struct qstr *name, struct lustre_id *cid,
        lcontext_t *ct)
{
    struct mdc_node *mdc_parent = NULL;
    struct ptlrpc_request *req = NULL;
    struct lustre_handle lockh;
    ENTRY;

    ...
    rc = mdc_enqueue(exp, name, &lockh, &req, ...);
    if (rc) {
        CERROR('error taking lock %d\n', rc);
        RETURN(rc);
    }

    mdc_parent = get_mdc_node(parent);
    LASSERT(mdc_parent != NULL);

    /*
     * saving taken lock handle. This is actually
     * needed for debugging, as lock handle may be foremd.
     */
    LASSERT(mdc_parent->lockh.cookie[0] == 0);
    mdc_parent->lockh = lockh;
    /*
     * saving found @lockh, @mode and returned data to
     * @ct. Saving found object lustre_id to @cid. It
     * will be used later for reading inode from MDS.
     */
    if (req) {
        struct mds_body *body;

        body = lustre_msg_buf(req->rq_repmsg,
                              1, sizeof(*body));
        if (!body) {
            ptlrpc_req_finished(req);
            RETURN(-EINVAL);
        }

        LASSERT((body->valid & OBD_MD_FID) != 0);
        id_assign(cid, body->id1);
        ptlrpc_req_finished(req);
    } else {
        /* should not happen */
        LBUG();
    }
```

```
    ...
    RETURN(rc);
}
```

3.2.4 revalidate() implementation in LLITE

The most important part of LLITE revalidate() looks like the following:

```
static int
ll_revalidate_it(struct dentry *de, int flags,
                 struct nameidata *nd, lcontext_t *ct)
{
    struct ll_inode_info *lli = ll_i2info(de->d_inode);
    struct ll_inode_info *plli;
    int rc = 0;
    ENTRY;

    ...

    plli = ll_i2info(de->d_parent);
    rc = md_revalidate(ll_i2md_exp(de->d_inode),
                       plli->lli_node, de->d_name,
                       lli->lli_node, ct);

    ...
    RETURN(rc);
}
```

3.2.5 revalidate() implementation for LMV

As it was said before, the main goal of LMV is to perform transparent switching between MDC targets. For this method it should choose correct MDC target to send revalidate request to.

```
static int
lmv_revalidate(struct obd_export *exp, lnode_t *parent,
               struct qstr *name, lnode_t *child,
               lconetxt_t *ct)
{
    struct lmv_node *lmv_parent;
    struct lmv_node *lmv_child;
    lnode_t *entity = NULL;
    int mds, rc = 0;
    ENTRY;

    ...
```

```
    lmv_parent = get_lmv_node(parent);
    /*
     * check if parent has lmv node, that is
     * splitted. If so we should pass control
     * to correct target.
     */
    if (is_splitted_node(parent)) {
        mds = raw_name2idx(lmv_parent->hashtype,
                           lmv_parent->objcount,
                           name);
        entity = lmv_parent->obj[mds].entity;
    } else {
        entity = lmv_parent->entity;
        mds = id_group(lmv_parent->id);
    }

    /* call revalidate() for next MD layer device */
    rc = md_revalidate(lmv->tgts[mds], entity,
                       name, child, ct);

    ...
    RETURN(rc);
}
```

3.2.6 revalidate() implementation for MDC

The goal for MDC in this method is to check if lock on client is still valid and can be found by resource id formed from passed lustre_id stored in @child.

```
static int
mdc_revalidate(struct obd_export *exp, lnode_t *parent,
               struct qstr *name, lnode_t *child,
               lconetxt_t *ct)
{
    struct lustre_handle lockh;
    int rc = 0;
    ENTRY;

    ...

    LASSERT(child != NULL);

    /* checking for cached locks */
    rc = mdc_find_lock(exp, &child->id, &lockh);
    if (rc) {
```

```
        /* saving found @lockh and @mode to @ct. */
        RETURN(1)
    } else {
        /* nothing found, we need lookup */
        RETURN(0);
    }
}
```

3.3 Use cases for readdir()

This is may be most complicated method in MD API. Especially if count LMV API in use.

3.3.1 readdir() implementation for LLITE

```
static int
ll_readdir(struct file *filp, void *dirent,
           filldir_t filldir)
{
    struct inode *inode = filp->f_dentry->d_inode;
    struct ll_inode_info *lli = ll_i2info(inode);
    unsigned long off = filp->f_pos;
    int rc = 0;
    ENTRY;

    ...

    /* reading directory content at @off to @dirent */
    rc = md_readdir(ll_i2md_exp(inode), lli->lli_node,
                    filp->f_opaque, &off, dirent, filldir);
    if (rc) {
        CERROR('can't read dir at %lu, err %d\n', off, rc);
        RETURN(rc);
    }
    filp->f_pos = off;
    filp->f_version = inode->i_version;
    update_atime(inode);
    RETURN(rc);
}
```

3.3.2 readdir() implementation for LMV

As usual for LMV, its purpose here is to call correct MDS target and manage the cases when one directory lives on different MDSs. The most difficult goal here is to manage things such a way to avoid throwing out pages from local

page cache for whole directory in the case of lock distraction. This would give us substantial speedup.

```
/*
 * helper method, finds mds number passed off is
 * located on.
 */
static int
lmv_off2mds(struct obd_export *exp, lnode_t *node,
            unsigned long *off)
{
    unsigned long noff = 0, mds;
    struct lmv_node *lmv_node;
    ENTRY;
    LASSERT(node != NULL);

    if (!is_splitted_node(node))
        RETURN(0);

    /*
     * looking for correct MDS needed dir part lives
     * on and relative offset.
     */
    lmv_node = get_lmv_node(node);
    for (mds = 0; mds < lmv_node->objcount; mds++) {
       if (noff + lmv_node->obj[mds].size >= *off) {
           /* saving relative offset to @off */
           *off = *off - noff;
           RETURN(mds);
       }
       noff += lmv_node->obj[mds].size;
    }
    *off = *off - noff
    RETURN(-ERANGE);
}
/* LMV readdir() itself */
static int
lmv_readdir(struct obd_export *exp, lnode_t *node,
            void *filp, unsigned long *off,
            void *buff, filldir_t filldir,
            lcontext_t *ct)
{
    struct lmv_node *lmv_node;
    unsigned long roff = *off;
    lnode_t *entity = NULL;
    int mds, rc = 0;
```

```
        ENTRY;

        ...

        lmv_node = get_lmv_node(node);
        /* check if directory splitted */
        if (is_splitted_node(node)) {
             mds = lmv_off2mds(exp, lmv_node, &roff);
             if (mds == -ERANGE) {
                 CERROR('needed offset %lu lies behind '
                        'the object size\n', *off);
                 RETURN(-ERANGE);
             }
             tgt_exp = lmv->tgts[mds];
             entity = lmv_node->obj[mds].entity;
        } else {
            /*
             * dir is not splitted, getting dir content
             * from MDS @node lies at.
             */
            mds = id_group(lmv_node->id);
            tgt_exp = lmv->tgts[mds];
            entity = lmv_node->entity;
        }

        /* requesting dir content from next MD layer */
        rc = md_readdir(tgt_exp, entity, filp,
                        &roff, buff, filldir, ct);
        if (rc) {
            CERROR('readdir() failed, err %d\n', rc);
            RETURN(rc);
        }
        *off = roff;
        RETURN(rc);
    }
```

3.3.3 readdir() implementation for MDC

As usually, MDC takes care of locks and RPCs. Here are readdir itself and few helpers used from it.

```
    /*
     * this function does actually reading directory content
     * from server and transforms it from lustre network or
     * server format to required by system using passed @filldir
     * conversion function.
```

```
 */
static int
mdc_readdir_rpc(struct obd_export *exp, lnode_t *node,
                void *filp, unsigned long *off,
                void *buff, filldir_t filldir,
                lcontext_t *ct)
{
    int rc = 0;
    ENTRY;

    ...
    /* making RPCs, filling @buff by data using @filldir. */

    ...
    RETURN(rc);
}

/* and this function takes page from page cache */
static int
mdc_readdir_cache(struct obd_export *exp, lnode_t *node,
                  void *filp, unsigned long *off,
                  void *buff, filldir_t filldir,
                  lcontext_t *ct)
{
    int rc = 0;
    ENTRY;

    ...
    /*
     * getting page from page cache and filling @buff
     * by data using @filldir.
     */

    ...
    RETURN(rc);
}
static int
mdc_readdir(struct obd_export *exp, lnode_t *node,
            void *filp, unsigned long *off,
            void *buff, filldir_t filldir,
            lcontext_t *ct)
{
    struct obd_device *obd = exp->exp_obd;
    struct lustre_handle lockh;
    unsigned long roff = *off;
    ldlm_policy_data_t policy;
```

```
        int mds, rc = 0;
        ENTRY;

        ...

        /*
         * looking if we still have page in local
         * page cache.
         */
        rc = mdc_find_lock(exp, &node->id, &lockh);

        if (rc == 0) {
            /* lock is not found, taking it */
            rc = md_enqueue(exp, &lockh, ...);
            if (rc) {
                CERROR('can't take read lock for '
                       'reading dir content\n');
                RETURN(rc);
            }

            /*
             * actual reading RPC sending, dir content
             * transforming using @filldir, etc.
             */
            rc = mdc_readdir_rpc(exp, node, entity,
                                 off, buff, filldir);
        } else {
            /* readding data from local page cache */
            rc = mdc_readdir_cache(exp, node, entity,
                                   off, buff, filldir);
        }
        RETURN(rc);
    }
```

3.4 Use cases for locks distraction

The main goal of this stuff is to notify all MD layers of locks distraction. Many actions should be performed in this time like cache invalidating, etc.

The main rule here is to deal only with data belonging to corresponding level of abstraction. Thus, LLITE cannot do something related to LMV, and so on.

Locks distraction (revocation) is done in such a way that only MDC as master of locks will have blocking ast callback call and have to call next upper MD layer from it. Next MD layer should handle it and make decision should it forward this lock revocation to its upper layer or not. This decision apparently should be made on understanding what kind of data should be revalidated on particular lock revocation. For instance, in some cases LMV should not forward

it to LLITE, as all is needed to be updated/invalidated is located on LMV layer.

As MDC layer takes care of lock cancelling on blocking, the rest of blocking ast callbacks will not have such stuff at all and will handle only LDLM_CB_CANCELING case.

3.4.1 locks distraction for LLITE

```
static int
ll_blocking_ast(struct ldlm_lock *lock,
                void *cbdata, int flags)
{
    struct lustre_handle lockh;
    int rc = 0;
    ENTRY;

    switch (flag) {
    case LDLM_CB_CANCELING:
        struct lnode *inode = (struct inode *)cbdata;
        ...

        /*
         * cleaning diff. things like open lock
         * related one, unhashing aliases, etc.
         */

        ...
        iput(inode);
        break;
    default:
        LBUG();
    }
    RETURN(rc);
}
```

3.4.2 locks distraction for LMV

The main goal here is to manage things such a way to release from page cache only those pages which belong to MDS which gets modified in the case of splitted dir.

The main idea how to implement it is to really onto per-layer lock distraction mechanism. Accordingly to it, only MDC takes and releases locks and each layer only has own lock distraction callback for each lustre object.

Thus each MD layer having own callback only clears things related to it and LMV layer as one which knows what MDS page at particular offset belongs to, takes care of this page and must throw it out from page cache on lock revocation. Thus only those pages will be released which belong to MDS that modified the

directory.

```
static int
lmv_blocking_ast(struct ldlm_lock *lock,
                 void *cbdata, int flags)
{
    struct lustre_handle lockh;
    int rc = 0;
    ENTRY;

    switch (flag) {
    case LDLM_CB_CANCELING:
        struct lmv_obj *lmv_obj = (struct lmv_obj *)cbdata;
        struct lmv_node *lmv_node = lmv_obj->master;
        /*
         * clearing diff. things related to LMV like
         * subobject itself, it size, data pages,
         * attributes, etc.
         */
        rc = lmv_invalidate_obj(lmv_obj);
        if (rc)
            RETURN(rc);

        ...

        /* calling upper layer lock revocation cb */
        rc = md_revoke_lnode(lmv_obj->exp, lock,
                             node->lrv_data, flags);
        break;
    default:
        LBUG();
    }
    RETURN(rc);
}
```

3.4.3 locks distraction for MDC

As MDC manages locks, it also manages all blocking ast callbacks. Thus, its purpose here is first - to clear MDC related stuff and second - call blocking ast for upper layer to let it know that lock is distracted.

```
static int
mdc_blocking_ast(struct ldlm_lock *lock,
                 void *cbdata, int flags)
{
    struct lustre_handle lockh;
    int rc = 0;
```

```
        ENTRY;

        switch (flag) {
        case LDLM_CB_BLOCKING:
            ldlm_lock2handle(lock, &lockh);
            rc = ldlm_cli_cancel(&lockh);
            break;
        case LDLM_CB_CANCELING:
            struct mdc_node *mdc_node;

            /*
             * clearing diff. things related to MDC like
             * some flags, etc.
             */
            mdc_node = (struct mdc_node *)cbdata;

            ...

            rc = md_revoke_lnode(mdc_node->exp, lock,
                                 cbdata, flags);
            break;
        default:
            LBUG();
        }
        RETURN(rc);
    }
```

4 State specification

4.1 State sharing

Seems there is only one thing which have state and needs to maintain it correctly. And namely it is inode and its representation on each MD layer.

As from state managing point of view, objects should be created, destroyed, updated and invalidated correctly, in right time to reflect the real object state (splitted or not for LMV, object size, etc). This is done from two directions: client and server. Client initiates object creation, updating and destroying. And server causes LDLM lock revocation, which should be used on clients for invalidating corresponding object.

The following kins of data should be taken care in relation to state managing:

- inode and inode related structures on all MD layers. Inode itself and its reflections on different MD layers should be updated on state changes. This is all inode attributes and attributes of all inode related structures on MD layers. For instance, inode->i_size on LLITE layer and lmv_node->size or lmv_node->objcount (for splitted nodes) on LMV layer.

- inode data (directory content). As splitted directory are possible and only LMV knows that an object is splitted directory, invalidating directory pages should be done on LMV layer.

As for a particular layer (in relation to private object managing), it (layer) takes care of correct object updating having all the data needed from system or API itself. It uses spin locks and semaphores and others synchronization primitives for protecting structures in own patch and maintaining their state.

4.2 Recovery

From recovery point of view, no changes to usual schema is needed as API is only the way of interaction.

5 Environment

5.1 Network API

No changes in network API or packets format is needed. This is client only API and all the rest of things will be implemented using existing network framework.

5.2 Format change, configuration and compatibility

No changes in disk format configuration is needed. As for compatibility, it should be held. That is clients running new API should work correctly in clusters which run old clients.

5.3 Documentation changes

Documentation changes are only those, it should reflect this API which is going to be in use. may be some parts of this DLD or DLD itself may be used as documentation.