Client Metadata API HLD

Alex Tomas, Yury Umanets

23rd February 2006

Contents

1 Requirements

Our goal is to get metadata API to build stackable, portable client. It should be compatible with the following requirements:

1.1 Common requirements

- client should run stack of "metadata" devices;
- each layer may have own private data for each object;
- modules should stay portable as much as possible;
- each layer should have ability to be notified about lock retraction;
- metadata stack should be able to pass received attributes to the data stack;
- metadata stack should be adopted to such invariants as:
  - networking in MDC;
  - DLM in MDC;
  - using FIDs in accordance with FIDs HLD.

1.2 Specific requirements

- striped dir handling should be implemented in the LMV module;
- Llite layer should be as small as possible: it's OS specific.

2 Functional Specifications

All metadata modules form metadata stack. The top module is Llite, it interacts with operating system. The bottom module is MDC, it interacts with MDT via network.

Each operation has an intent and intermediate results. An intent describes what should be done during an operation: attributes to be gotten for the file, file to be opened, etc. We'll use *lookup context* (or *lcontext)* as a name for the structure that stores an intent and intermediate results:

```
struct lcontext {
    intent;             /* intent of operation */
    credentials;        /* uid/gids */
    attrs;              /* attributes to be passed
                         * to the data stack */
    private;            /* data of lower module */
};
```

We can't pass struct inode down through the stack because of portability requirement. So, we need one more structure:

```
struct lnode {
    fid;                /* node fid */
    entity;             /* reference to next layer node */

    ...
}
```

All private data for a object must be linked into a chain. So, OS-specific structure (struct inode, for example) has address of object-related info for Llite. In turn, Llite's structure has address of object-related info for MDC and so on.

Also, to implement cache-consistency-via-LDLM we need callback mechanism. Similar to private info described above, each module installs callback during object initialization.

```
struct lcallback {
    int (*callback) (struct ldlm_lock *lock,
                     void *cbdata, int flags);
    void *cbdata;
};
```

All metadata modules (but top one, Llite) must provide set of methods (listed and described in Logic specifications section).

2.1 lookup() method

2.1.1 Prototype

```
int md_lookup(struct obd_device *obd,
              struct lnode *parent,
              struct qstr *name,
              struct lnode *child,
              struct lcontext *ct);
```

2.1.2 Description

The method is called when dcache has no valid dentry for (parent; name) couple or when revalidate method returned *False* for the dentry. The caller expects valid fid in child structure or error code as return value. The fid will be used to find and initialize OS-specific structure (inode in Linux, vnode in *BSD and so on). The intent structure should have valid credits at least. It also, can contain an intent that MDS will use to implement *do-ahead* logic. The method can leave some data in the context for future using. For example, MDS reply can contain attributes of the object being looked up. The method leaves these attributes in the context, subsequent md_read_inode() method will find them and won't issue additional RPC to fetch the attributes.

2.1.3 Parameters

- obd - next obd in the metadata stack;
- parent - parent directory;
- name - name to be resolved in the parent directory;
- child - the method should return fid in this structure;
- ct - context of the request.

2.1.4 Return value

The method returns 0 in successful case, error code otherwise.

2.2 read_lnode() method

2.2.1 Prototype

```
void *md_read_lnode(struct obd_device *obd,
                    struct lnode *node,
                    struct lcontext *ct,
                    struct lcallback *lcb);
```

2.2.2 Description

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

2.2.3 Parameters

- obd - next obd in the metadata stack;
- node - inode to be initialized;
- ct - context of the request;
- lcb - callback the module will call from own callback. The lcb must be stored to private structure of node.

2.2.4 Return value

The method returns an address of structure the method allocates for the inode.

2.3 revalidate() method

2.3.1 Prototype

```
int md_revalidate(struct obd_device *obd,
                  struct lnode *parent,
                  struct qstr *name,
                  struct lnode *child,
                  struct lcontext *ct);
```

2.3.2 Description

The method is called to check whether {parent; name} still resolves to child.

2.3.3 Parameters

- obd - next obd in the metadata stack;
- parent - parent directory;
- name - name of object;
- child - object this name was resolved to last time;
- ct - context of the request.

2.3.4 Return value

The method returns 1 if resolving is still valid and 0 if the name needs to be looked up again.

2.4 open() method

2.4.1 Prototype

```
void *md_open(struct obd_device *obd,
              struct lnode *node,
              struct lcontext *ct);
```

2.4.2 Description

The method is called to open an object.

2.4.3 Parameters

- obd - next obd in the metadata stack;
- node - object to be opened;
- ct - context of the request.

2.4.4 Return value

The method returns an address of the structure the method allocates for an instance of open object or error code in ERR_PTR form.

2.5 getattr() method

2.5.1 Prototype

```
int md_getattr(struct obd_device *obd,
               struct lnode *node,
               struct stat *stat,
               struct lcontext *ct,
               int refresh);
```

2.5.2 Description

The method is called to fetch attributes for given object. This method can be used during initialization of in-core object. As we need minimal set of attributes (type, for example), it would be enough to get non-fresh ones.

2.5.3 Parameters

- obd - next obd in the metadata stack;
- node - object the caller asks attributes for;
- stat - the method should store fetched attributes there;
- ct - context of the operation;
- refresh - flag: 1 - if attributes aren't fresh, fetch them from a server, 0 - return attributes from local storage.

2.5.4 Return value

The method returns 0 on success and error code otherwise.

2.6 create() method

2.6.1 Prototype

```
int md_create(struct obd_device *obd,
              struct lnode *parent,
              struct qstr *name,
              struct lcontext *ct);
```

2.6.2 Description

The method is called to create named object in directory parent.

2.6.3 Parameters

- obd - next obd in the metadata stack;
- parent - directory where caller wants to create object;
- name - name of object to be created;
- ct - context of the operation. it will contain credits of the caller and attributes for new object (type, access modes and so on).

2.6.4 Return value

The method returns 0 on success and error code otherwise.

2.7 link() method

2.7.1 Prototype

```
int md_link(struct obd_device *obd,
            struct lnode *parent,
            struct qstr *name,
            struct lnode *target,
            struct lcontext *ct);
```

2.7.2 Description

The method is called to create one more name for object target in the directory parent.

2.7.3 Parameters

- obd - next obd in the metadata stack;
- parent - directory where caller wants to create object;
- name - name of object to be created;
- target - object new name should point on;
- ct - context of the operation. it will contain credits of the caller and attributes for new object (type, access modes and so on).

2.7.4 Return value

The method returns 0 on success and error code otherwise.

2.8 unlink() method

2.8.1 Prototype

```
int md_unlink(struct obd_device *obd,
              struct lnode *parent,
              struct qstr *name,
              struct lcontext *ct);
```

2.8.2 Description

The method is called to unlink name from directory parent. Once last name to object is removed, the object itself is removed too.

2.8.3 Parameters

- obd - next obd in the metadata stack;
- parent - directory where caller wants to create object;
- name - name to be unlinked;
- ct - context of the operation. it will contain credits of the caller and attributes for new object (type, access modes and so on).

2.8.4 Return value

The method returns 0 on success and error code otherwise.

2.9 rename() method

2.9.1 Prototype

```
int md_rename(struct obd_device *obd,
              struct lnode *old_parent,
              struct qstr *old_name,
              struct lnode *new_parent,
              struct qstr *new_name,
              struct lcontext *ct);
```

2.9.2 Description

The method is called to rename old_name in old_parent directory to new_name in new_parent directory.

2.9.3 Parameters

- obd - next obd in the metadata stack;
- old_parent - directory with old_name;
- old_name - name to be renamed;
- new_parent - directory for new_name;
- new_name - new name of an object;
- ct - context of the operation. it will contain credits of the caller and attributes for new object (type, access modes and so on).

2.9.4 Return value

The method returns 0 on success and error code otherwise.

2.10 readdir() method

2.10.1 Prototype

```
int md_readdir(struct obd_device *obd,
               struct lnode *node,
               void *private,
               __u64 *offset,
               filldir_t filldir,
               void *buffer);
```

2.10.2 Description

The method is called to read directory content.

2.10.3 Parameters

- obd - next obd in the metadata stack;
- node - directory to be read;
- private - return value of open method;
- offset - where in directory to start reading. Also, the method should return offset of a next entry to be read;
- filldir - routine that the method will call to store directory entries;
- buffer - storage for directory entries.

2.10.4 Return value

The method returns 0 on success and error code otherwise.

2.11 close() method

2.11.1 Prototype

```
int md_close(struct obd_device *obd,
             struct lnode *node,
             void *private);
```

2.11.2 Description

The method is called to close open object, drop references and release memory the stack allocated for the object.

2.11.3 Parameters

- obd - next obd in the metadata stack;
- node - object to be closed;
- private - private data peer method md_open() returned.

2.11.4 Return value

The method returns 0 on success and error code otherwise.

2.12 init_lcontext() method

2.12.1 Prototype

```
int md_init_lcontext(struct obd_device *obd,
                     struct lcontext *ct);
```

2.12.2 Description

The method is called before any another method to prepare context.

2.12.3 Parameters

- obd - next obd in the metadata stack
- ct - context where temporal data is stored

2.12.4 Return value

The method returns 0 on success and error code otherwise.

2.13 release_lcontext() method

2.13.1 Prototype

```
void md_release_lcontext(struct obd_device *obd,
                         struct lcontext *ct);
```

2.13.2 Description

The method is called in the end of an operation to free all resources: requests, locks, etc.

2.13.3 Parameters

- obd - next obd in the metadata stack;
- ct - context where temporal data is stored.

2.13.4 Return value

None

2.14 clear_lnode() method

2.14.1 Prototype

```
void md_clear_lnode(struct obd_device *obd,
                    struct lnode *node);
```

2.14.2 Description

The method is a complement for read_lnode. It's called when OS purges object from cache. The purpose of the method is to release all structures and locks related to the object.

2.14.3 Parameters

- obd - next obd in the metadata stack;
- node - object to be released.

2.14.4 Return value

None

3 Use cases (in Linux)

3.1 single MDS configuration

3.1.1 stat /A/B/F1 with empty dcache

The objects A and B wont be found in the cache and VFS will call lookup method against them to them get in the cache. Intent for these methods will be LOOKUP because all we need is to get next fid to lookup in. For the last component real intent UPDATE will be used and lookup method will return both LOOKUP+UPDATE locks and attributes. After that, getattr will be called to fill stat structure. If UPDATE lock isn't granted it will refresh attributes via one more RPC.

3.1.2 stat /A/B/F2 with A and B in dcache

The difference from the previous case is that A and B will be found in dcache, VFS will revalidate them via revalidate method.

3.1.3 open /A/B/F3 with create flag

The special OPEN intent directs MDC to issue a RPC from revalidate or lookup methods depending on state of the dcache. MDS will try to look F3 up. In success case, MDS will open it and send a reply back to the client. If the object doesn't exists, MDS will create it.

3.2 multi MDS configuration

In this configuration LMV module hides from Llite that cluster has numerous MDS nodes. The key features of multi MDS configuration are:

- cross-node objects - a name is stored on one MDS and it points to an inode that is stored on another MDS;
- striped directories - a directory can be distributed among few MDS nodes (set of objects - subobjects).

3.2.1 stat /A/B/F1 with F1 pointing to another MDS

Obviously, lookup operation against such a file can't return attributes. Subsequent read_lnode method won't find needed attributes in the context and will issue RPC.

3.2.2 stat against striped directory

During object initialization read_lnode method of LMV module will observe that object is a striped directory. It will create additional private structure that holds list of MDC's objects for all involved subobjects. Then stat method will refresh all MDC's objects and return summarized info back to the caller.

3.2.3 readdir against striped directory

LMV module will choose subobject using offset parameter and set of subobject's sizes. Then it will forward READDIR request down to the proper MDC.

4 Logic specifications

4.1 Methods

All the modules should get listed below methods that perform requested functionality.

lookup() - the method translates {parent; name} couple to a fid. It can have a context and send an intent to MDS to do some operations ahead (attributes prefetching, object open);

revalidate() - the method checks whether given {parent; name} couple still resolves to a specified fid. It also can send a RPC with an intent to do some operations ahead using context;

read_lnode() - the method should initialize in-core structures for a given fid and fetch minimal set of attributes. It can have a context, use prefetched data and an intent from it;

clear_lnode() - the method is called to release all data and locks used by an object;

open() - the method is called to open an object. It can have a context and use prefetched data/lock from it;

close() - the method is used to close an object and release data for this opened instance;

getattr() - the method returns attributes fog an object. It can use data and locks from an operation context;

create() - the method is called to create an object in given directory with given name. Credits and permissions can be gotten from an operation context;

link() - the method is called to create one more name for a given object in a given directory. Credits can be gotten from an operation context;

unlink() - the method is called to unlink a given name from a given directory;

rename() - the method is called to rename an object;

readdir() - the method is called to fetch an directory content. The method can store fetched data in a local cache;

init_lcontext - the method is called before any another method gets called and can be used to initialize all needed structures;

release_lcontext() - the method is used to release prefetched data and locks client can get during operaion handling.

4.2 Llite module

The purpose of the module is to interact with OS-specific VFS, prepare all needed data and call proper metadata methods. It also has to pass attributes from the metadata stack to the data stack via a context. Llite's callbacks can be called to invalidate cached metadata OS can have as a result of previous requests.

4.3 MDC module

MDC module is portable and it shouldn't depend on OS much. MDC module implements lowest layer of the metadata stack. It prepares RPC's using intents from contexts. It stores attributes for objects. It registers callbacks in LDLM and calls upper layer's callback to notify state changes.

4.4 LMV module

LMV modules impelements multi MDS support on client side:

request forwarding - depending on parent fid (or on name, if directory is striped) LMV forwards request to proper MDS hiding this complexity from upper layers;

data aggregating - upper layers know nothing about striped directories, but they still want to have actual metadata for them: size, modification times, blocks. LMV maintains all subobjects of a striped directory exporting MDC's abilities to store object's attributes.

5 State management

6 Focus for inspections

- Mac OS X and Windows port;
- compatibility with new Linux intents patches by Oleg.