FIDs DLD

Yury Umanets

15th Apr, 2006

Contents

1 Introduction

This DLD only describes details of FIDs changes. Many aspects such as FLD, OI, Sequence management are left outside of it and matter of separate DLDs.

2 Functional Specification

Purpose of this work is to implement FIDs (as described in FIDs HLD).

2.1 FIDs abstract

FID is used for the following purposes:

- FID is a unique object identifier in Lustre for both stacks: metadata and data one. All functions which identify objects for any purpose should use FID except for OSD code;
- FID is used for DLM as base or making resource id and uniquely identify lustre objects in DLM service as well;
- FID is used as base for making client's inode numbers. This helps to avoid their duplication when inodes live on different MDS servers.

FID structure looks like the following:

```
struct lu_fid {
        __u64 f_seq;
        __u32 f_oid;
        __u32 f_ver;
};
```

Fields in the structure have the following meaning:

f_seq - objects sequence number, cluster wide, cluster unique value. May be changed quite quickly, contains predefined number of objects;

f_oid - objects identifier, unique in its sequence. Pair f_seq and f_oid form unique object identifier;

f_ver - objects version, needed for implementing objects of different versions in Lustre.

Few functions to access *struct lu_fid* fields:

fid_seq(fid) - get sequence from passed fid;

fid_oid(fid) - get fid object id from passed fid;

fid_ver(fid) - get fid version from passed fid;

fid_num(fid) - get compound fid id and version;

FID has the following properties:

- abstraction from the storage. That is, it does not contain any store related information like inode number and thus, it is handy in any kind of migrations because does not need additional conversion;
- FID is generated on a client. That is, client, before sending create RPC to server already knows object's identifier and thus, many nice features are becoming possible, for instance WB caching;
- sequence numbers are generated and managed on server. Server which generated some sequence is home server for that sequence. This means, that all objects created in this sequence live on same server - home server of the sequence;

FIDs are resolved into object into the following two stages:

- using FLD (FIDs Location Database) service, client or server may find home server of some FID;
- on home server, FID is translated into backing store inode using OI (Object Index) service.

This way, clients do not need to have server store cookies of some inode and only use FIDs to access it.

2.2 FIDs invariants

The following FIDs invariants should be supported:

- FID number (that is object number) cannot be zero. Zero value is used for uninitialized FID number and can be used for validness checks;
- FID number is started from scratch for each new sequence;
- sequence is collection of FIDs created in it and living on same server;
- sequence is denoted by cluster-unique sequence number;
- FID sequence number cannot be zero. Zero value means uninitialized sequence and may be used for sanity checks;

- FID sequence number is incremented by one each time as new connect is performed or current sequence is exhausted. Code should have corresponding checks that sequence is close to overflow;
- FID sequence number starts from some predefined value, say 0x400, thus first 1024 sequences are left for special purposes (for instance compatibility), see chapter 6 for details;
- each server should have a collection of sequences or at least one sequence, used for special purposes. For instance, server has special FS structures like object index directory, etc., which should be accessible by FID as well as others. It is logically to build them based on server number. Thus, MDT0 would have special sequences from 1 to 100 and MDT1 - from 100 to 200 and so on;
- FID sequences actually may be different number of FIDs allocated in them. This is because some clients may reconnect or recover, etc.

3 Use Cases

No new uses cases are foreseen different than in FIDs HLD. Consult it for details.

4 Logic Specification

In order to implement all FIDs functionality and properties, the following changes should be done.

4.1 Common changes

The following common changes will take place on server and client:

- all functions using *struct ll_fid*, or pair *ino* and *generation* should use *struct lu_fid* in both, server and client;
- all wire structures should use *struct lu_fid* instead of former *struct ll_fid*.

4.1.1 Using FID for DLM

To issue DLM locks or to check that some lock is cached, client and server use FID to form resource id out. This looks like the following:

```
void fid_build_res_name(struct lu_fid *fid,
                        struct ldlm_res_id *res_id)
{
    res_id->name[LUSTRE_RES_ID_SEQ_OFF] = fid_seq(fid);
    res_id->name[LUSTRE_RES_ID_OID_OFF] = fid_oid(fid);
    res_id->name[LUSTRE_RES_ID_VER_OFF] = fid_vef(fid);
}
```

As it is seen from code example, third component of res_id is FID version, what makes it possible to divide access to objects of different versions.

4.2 Client changes

Client will see the following changes.

4.2.1 Inode numbers based on FID

As FID is unique over the cluster, thus, it is good base for making client's inode numbers. Idea is to map *struct lu_fid* (which is in fact two dimensional array, having seq numberand fid number in seq) to one dimension inode numbers space which is 32bit wide.

Algorithm looks like the following:

```
/*
 * This is how may FIDs may be allocated in one sequence.
 * 16384 for now.
 */
#define LUSTRE_SEQ_MAX_WIDTH 0x0000000000004000ULL

static inline __u64 fid_flatten(const struct lu_fid *fid)
{
        return (fid_seq(fid) - 1) *
                LUSTRE_SEQ_MAX_WIDTH + fid_oid(fid);
}

ino_t ll_fid_build_ino(struct ll_sb_info *sbi,
                       struct lu_fid *fid)
{
        ino_t ino;
        ENTRY;
        ino = fid_flatten(fid);
        RETURN(ino & 0x7fffffff);
}
```

As it is seen from code above, each sequence has own range of object that may be created in it. Range size may be chosen as tunable or as predefined value, based on experience. This algorithm also produces inode numbers which do not depend on version component from FID.

This algorithm has the following downsides:

- in case of quick sequence changing, not completely filled ranges are possible due to fixed range width, what is wasting the 32bit limited inodes space;
- if LUSTRE_FID_SEQ_WIDTH is 10 000 objects, only ~70 000 000 sequences are possible what is not good enough, counting that cluster may

have more than 10 000 nodes. Thus, each node may perform only 7 000 sequence switches what is terribly small number. Making range smaller size does not help much, because sequences will be changed more often and thus, total time of one client run will be roughly the same before it runs into trouble.

Taking into account downsides above, algorithm is effectively not usable. To design effective algorithm we should understand that all clients will have different number of FIDs allocated in sequences and thus, new sequences are needed more often than if they function in regular mode. Apparently, we want to consider possible reasons of premature sequence retiring. They are the following:

- client re-connects, thus loses its current sequence, server allocates new one for it;
- in some cases, thnks to intents, we can't know that object is already allocated, so allocate new fid for it and this new fid is not used later. So, sequence gets exhausted more quickly. This is the case of *ll_lookup_it()* with *it->it_op & IT_OPEN* and *it_create_mode & O_CREAT*. This may be fixed by detecting such a situtions and include such a not used FIDs into spcial lists where they may be found later for allocation.

If clients could foresee everage number of FIDs in sequence and this way to make inode numbers allocation denser.

4.2.2 Client allocates FIDs

Each time as new inode needs to be allocated, client allocates new FID in its current sequence and makes inode - FID association.

```
static struct dentry *
ll_lookup_it(struct inode *parent,
             struct dentry *dentry,
             struct lookup_intent *it,
             int lookup_flags)
{
        ...

        if (it->it_op & IT_CREAT ||
            (it->it_op & IT_OPEN &&
             it->it_create_mode & O_CREAT))
        {
                struct lu_placement_hint hint =
                        { .ph_pname = NULL,
                          .ph_cname = &dentry->d_name,
                          .ph_opc = LUSTRE_OPC_CREATE
                        };
```

```
                rc = ll_fid_md_alloc(ll_i2sbi(parent),
                                     &op_data.fid2, &hint);
                if (rc) {
                        CERROR("can't allocate new fid,
                               rc %d\n", rc);
                        LBUG();
                }
        }

        ...
}
```

In the code above llite delegates inode fid allocation to lower layers: LMV and MDC. This is because LMV contains placement policy, which chooses what MDS new fid should be allocated on. And MDC holds last used FID in current sequence for its MDS.

In LMV this looks like the following:

```
static int
lmv_fid_alloc(struct obd_export *exp,
              struct lu_fid *fid,
              struct lu_placement_hint *hint)
{
        struct obd_device *obd = class_exp2obd(exp);
        struct lmv_obd *lmv = &obd->u.lmv;
        int rc = 0, mds;
        ENTRY;
        LASSERT(fid != NULL);
        LASSERT(hint != NULL);
        mds = lmv_placement_policy(obd, hint);
        if (mds < 0 || mds >= lmv->desc.ld_tgt_count) {
                CERROR("can't get target for allocating fid\n");
                RETURN(-EINVAL);
        }
        /* asking underlaying tgt layer to allocate new fid */
        rc = obd_fid_alloc(lmv->tgts[mds].ltd_exp, fid, hint);
        RETURN(rc);
}
```

The *struct lu_placement_hint* is used here to give LMV idea of what is the operation, what names of objects and other stuff about calling this method and allow it to make right decision as for where to forward this request and what MDS to use to create new object on it.

And in MDC module it looks like this:

```
static int
```

```
mdc_fid_alloc(struct obd_export *exp,
              struct lu_fid *fid,
              struct lu_placement_hint *hint)
{
        struct client_obd *cli = &exp->exp_obd->u.cli;
        int rc = 0;
        ENTRY;
        LASSERT(fid != NULL);
        LASSERT(hint != NULL);
        LASSERT(fid_seq(&cli->cl_fid));
        spin_lock(&cli->cl_fid_lock);
        if (fid_oid(&cli->cl_fid) < LUSTRE_FID_SEQ_WIDTH) {
                cli->cl_fid.f_oid += 1;
                *fid = cli->cl_fid;
        } else {
                CERROR("sequence is exhausted. Switching to "
                       "new one is not yet implemented\n");
                rc = -ERANGE;
        }
        spin_unlock(&cli->cl_fid_lock);
        RETURN(rc);
}
```

4.2.3 Client saves FID to inode

After FID is allocated by client in create time, it is sent to server to create object on it. When object is created and server answers with create RPC, client saves FID of successfully created object into its client inode.

For that, *ll_prep_inode()* function is used. It is also used for updating inode by data from MDS. Function *ll_prep_inode()* in turn uses *ll_iget()* for getting inode actually. It looks like the following:

```
int
ll_prep_inode(struct inode **inode,
              struct ptlrpc_request *req,
              int offset, struct super_block *sb)
{
        struct ll_sb_info *sbi;
        struct lustre_md md;
        int rc = 0;
        ENTRY;
        LASSERT(sb);
        sbi = ll_s2sbi(sb);
        prune_deathrow(sbi, 1);
        rc = md_get_lustre_md(sbi->ll_md_exp, req,
                              offset, sbi->ll_dt_exp, &md);
```

```
        if (rc)
                RETURN(rc);
        if (*inode) {
                ll_update_inode(*inode, &md);
        } else {
                LASSERT(sb != NULL);
                /* at this point server answers to client's
                 * RPC with same fid as client generated for
                 * creating some inode. So using ->fid1 is
                 * okay here. */
                LASSERT(fid_is_sane(&md.body->fid1));
                *inode = ll_iget(sb, ll_fid_build_ino(sbi,
                                 &md.body->fid1), &md);
                if (*inode == NULL || is_bad_inode(*inode)) {
                        md_free_lustre_md(sbi->ll_dt_exp, &md);
                        rc = -ENOMEM;
                        CERROR("new_inode -fatal: rc %d\n", rc);
                        GOTO(out, rc);
                }
        }

        LASSERT(fid_is_sane(&ll_i2info(*inode)->lli_fid));
        rc = obd_checkmd(sbi->ll_dt_exp, sbi->ll_md_exp,
                         ll_i2info(*inode)->lli_smd);
out:
        RETURN(rc);
}
```

As it seen from code above, client relies on FID returned by server. This is not really needed, because client already knows that FID for create case. But as this means substential client code reorganizing, we do not do that, because CMD3 is server side works mostly. This may be done later when time for client cleanups and new MDAPI comes.

And function *ll_iget()* for Linux 2.6.x looks like the following:

```
struct inode *
ll_iget(struct super_block *sb,
        ino_t hash, struct lustre_md *md)
{
        struct ll_inode_info *lli;
        struct inode *inode;
        LASSERT(hash != 0);
        inode = iget_locked(sb, hash);
        if (inode) {
                if (inode->i_state & I_NEW) {
                        lli = ll_i2info(inode);
```

```
                        ll_read_inode2(inode, md);
                        unlock_new_inode(inode);
                } else {
                        ll_update_inode(inode, md);
                }

                CDEBUG(D_VFSTRACE, "inode: %lu/%u(%p)\n",
                       inode->i_ino, inode->i_generation, inode);
        }
        return inode;
}
```

In code above, *ll_iget()* uses *ll_update_inode()* or *ll_read_inode2()* which in turn calls *ll_update_inode()* for updating inode attributes.

Function *ll_read_inode2()* looks like the following:

```
void
ll_read_inode2(struct inode *inode,
               void *opaque)
{
        struct lustre_md *md = opaque;
        struct ll_inode_info *lli = ll_i2info(inode);
        ENTRY;
        CDEBUG(D_VFSTRACE, "VFS Op:inode=%lu/%u(%p)\n",
               inode->i_ino, inode->i_generation, inode);
        ll_lli_init(lli);
        LASSERT(!lli->lli_smd);
        LTIME_S(inode->i_mtime) = 0;
        LTIME_S(inode->i_atime) = 0;
        LTIME_S(inode->i_ctime) = 0;
        inode->i_rdev = 0;
        ll_update_inode(inode, md);
        if (S_ISREG(inode->i_mode)) {
                struct ll_sb_info *sbi = ll_i2sbi(inode);
                inode->i_op = &ll_file_inode_operations;
                inode->i_fop = sbi->ll_fop;
                inode->i_mapping->a_ops = &ll_aops;
                EXIT;
        } else if (S_ISDIR(inode->i_mode)) {
                inode->i_op = &ll_dir_inode_operations;
                inode->i_fop = &ll_dir_operations;
                inode->i_mapping->a_ops = &ll_dir_aops;
                EXIT;
        } else if (S_ISLNK(inode->i_mode)) {
                inode->i_op = &ll_fast_symlink_inode_operations;
                EXIT;
```

```
        } else {
                inode->i_op = &ll_special_inode_operations;
#if (LINUX_VERSION_CODE > KERNEL_VERSION(2,5,0))
                init_special_inode(inode, inode->i_mode,
                                   kdev_t_to_nr(inode->i_rdev));
                inode->i_mapping->backing_dev_info =
                        &ll_backing_dev_info;
#else
                init_special_inode(inode, inode->i_mode,
                                   inode->i_rdev);
#endif
                EXIT;
        }
}
```

Function *ll_update_inode()* looks like the following:

```
void
ll_update_inode(struct inode *inode,
                struct lustre_md *md)
{
        struct ll_inode_info *lli = ll_i2info(inode);
        struct mdt_body *body = md->body;
        ...

        if (body->valid & OBD_MD_FLID) {
                spin_lock(&lli->lli_lock);
                lli->lli_fid = body->fid1;
                spin_unlock(&lli->lli_lock);
        }
        LASSERT(fid_is_sane(&lli->lli_fid));
}
```

Thus, having this code we have the following invariants:

- client creates inode and assigns FID to it atomically;
- client saves FID from server into inode inode, so that for each in-memory inode we have FID stored in its info;
- server is in charge to return the same fid what client sends to it with create RPC. This is not big deal and very handy, because no need to pass FID to all functions which already have lustre_md or ptlrpc_request with answer from server. Also all stacks, MDC and possibly LMV can access FID in all create related paths and check it, inspect, etc. When server returns FID, it puts special flag into *body->valid*;
- whereas FID is immutable after creating, server may send it all the time for some reasons.

4.2.4 LMV uses FLD

As FID does not contain any address information, LMV module uses FLD service for getting MDS number of where request should be forwarded to.

```
static int
lmv_getattr(struct obd_export *exp, struct lu_fid *fid,
            obd_valid valid, int ea_size,
            struct ptlrpc_request **request)
{
        struct obd_device *obd = exp->exp_obd;
        struct lmv_obd *lmv = &obd->u.lmv;
        struct lmv_obj *obj;
        int rc, mds;
        ENTRY;
        rc = lmv_check_connect(obd);
        if (rc)
                RETURN(rc);
        mds = lmv_fld_lookup(obd, fid);
        LASSERT(mds < lmv->desc.ld_tgt_count);
        LASSERT(mds >= 0);
        rc = md_getattr(lmv->tgts[mds].ltd_exp, fid,
                        valid, ea_size, request);
        if (rc)
                RETURN(rc);
        obj = lmv_obj_grab(obd, fid);

        ...
}
```

4.3 Server changes

Server will see the following changes.

4.3.1 Sequence management

There is need to manage sequences in cluster. Sequence number in any FID is allocated from one cluster wide set and thus, there should be a point in charge for that. See Sequence Management DLD for more details.

4.3.2 Object lookup

The following cases of lookup should be handled out by server:

- lookup object by FID. This is needed for any operation handled out on server, because what client sends to server as object identifier is FID;

- lookup object by name, return object attributes including FID to client. This is needed for lookup cases.

Lookup by FID

Object should be found by FID using OI (Object Index) service. OI is kind of FID->store cookie translation map which allows to find backing store FS inode and generation by FID. Later inode and generation are used for getting inode. This can be *iget(ino)* or something like this.

Lookup by name

Object should be found by name passed from client in lookup time. In this time also object's FID and other attributes should be obtained from backing store inode and passed to client. This is in fact the only case we do not have FID passed from client and have to find inode or FID some alternative way.

To do that, we can use FID stored in directory entry. This is needed for CMD cross-ref case as well. Client sends name to server and server finds dentry by name. Dentry contains *struct lu_fid* of the object it points to. Then we have the following two lookup cases:

- using FLD, server finds *home MDS* of the FID from directory entry, *home MDS* is same as current MDS. Then, MDS finds object by FID using OI like it is described in section 4.3.2;

- using FLD, server finds *home MDS* of the FID from directory entry, *home MDS* is not the same as current MDS. This is cross-ref case. In this case MDS return -ERESTART to client. Client then finds correct MDS and sends RPC to it. Such a technique avoid doing cascading RPCs. Cascading RPC is when client requests MDS0 and MDS0 requests MDS1. Such a chain may timeout and client should be smart enough to foresee this. Much easier to not produce cascades. One of condition which makes this scenario true is that, lookup RPC is sent to the MDS which holds name of an object. In CMD, this is always that MDS where parent directory inode located and thus, it may hold directory names. But inode itself may be stored on another MDS.

4.3.3 MDS lookup

Module CMM, running on server, like module LMV running in client should be able to find home MDS for any FID. They should use FLD (Fids Location Database) service. CMM uses FLD like it is shown in section 4.2.4 in LMV.

5 State Specification

5.1 Resources Involved and Their State

The following resources are considered to be managed.

5.1.1 Client inode with FID

Client inode with assigned FID. This resource is shared between client threads. It also has two states:

- just created inode has no FID assigned to it;
- inode has FID assigned to it.

Inode with no FID assigned should not be accessible by any thread. That is, inode creating and FID assigning to it should be atomic. After inode is created and FID is assigned, FID is immutable.

5.1.2 Server inode with OIE (Object Index Entry)

When server inode is created, it should also have corresponding OIE created in same transaction, to let server find it later by FID. Though here also two states, inode created and OIE created, no need to make state transition atomic as no one will find server inode without OIE created for it. Client to server protocol implies only using FIDs for pointing out what object should be used for some operation, and only one why to access object on server by FID is to use OI (Object Index) to find it.

No need to protect server side object in this case. Until OIE is created, nobody will access new created object anyway.

5.1.3 Server name with FID

Server should contain name of object being created along with FID of the object. Name and FID are created atomically as FID is located in directory entry. See section 4.3.2 for details of why we need to store FID in directory entry.

5.1.4 Last used meta-sequence

There is always some last used cluster wide *meta sequence* on each MDS. All threads on MDS can access it in any time, hence, it should be changed atomically.

Sequence is managed by sequence manager code which is running on all MDSes. It takes care of atomic change of last used sequence. See Sequence Management DLD for details.

5.2 Locking

No more legacy locking in resource id order is going to be used. Instead, parent first order should be introduced. This means at least getting rid of different checks in server.

5.3 Recovery

The following aspects of recovery needs to be described:

- server recovery, preserving meta-sequence and OI;
- client recovery, replaying RPCs.

5.3.1 Server recovery

Server recovery should take following invariants into account:

- meta-sequence should be saved to backing store each time as it changes. This should be done in same transaction as last created object with this sequence. This guarantees, that no objects with sequence more than last known is possible, thus, objects on backing store are always coherent with last sequence server knows about. See Sequence Management DLD for more details about sequence management;
- meta-sequence should be saved to backing store with immediate commit each time as it changes. This is not going to happen often, because sequences have some range of objects created in them. Then, in recovery time, when server is back from failure it may read last saved sequence and continue allocate new ones for new clients;
- objects on MDS should be created in same transaction as their OIE (Object Index Entries). This ensures that no lost objects is created on server FS and no empty (pointing to nothing) OIE in Object Index;
- objects created on MDS should have their FID EA (if needed) created in the same transaction;
- in case of server failure, no new sequence is given to old clients when they reconnect, because they may want to replay some RPCs;
- upon client's replay, server re-creates objects and makes sure that correct OIEs and EAs are created.

5.3.2 Client recovery

Client will see the following changes in recovery:

- in former, pre-fids solution, object's FID along with object's store cookies were returned to client and not only saved in inode for further use, but also saved in create/open RPCs which were preserved for the case of recovery. In recovery, these RPCs with filled in server side inode FID and store cookies were re-sent to server to re-construct server inodes with same attributes as they were created before failure. Currently, no need to save answered FIDs to preserved create/open RPCs. FIDs is already there because they are allocated on client side before sending RPCs to server and thus, RPCs have them when they go to server first time even;

- after client fails and connection is restored (after reboot), client obtains new sequences from all servers just like it is done for regular connects. This is because client loses its last FIDs and cannot continue using its sequences. Servers start new sequences for it. One more reason is that, client needs no old sequence, as nothing to replay because client loses all its data when rebooting.

5.4 Operation Ordering

Operation order is changed in meaning that FID of object being created is known before create RPC is sent to server even.

6 Environment

6.1 Disk Format Changes

In order to support new FIDs and CMD features, server should have the following disk format changes:

- no EA is needed to store object's FID with inode, see details in section 4.3.2;

- directory entries of ldiskfs filesystem should contain *struct lu_fid* to handle cross-ref dentries and usual lookups like it is shown in section 4.3.2;

- stripping info in EA on MDS is going to change format, in order to have *struct lu_fid* instead of object id and object group for each stripe data located on.

6.1.1 Directory entries

Directory entry in ldiskfs should look like the following:

```
struct ldiskfs_dir_entry_2 {
        /* usual entry fields */
        __le32 inode;
        __le16 rec_len;
```

```
        __u8 name_len;
        __u8 file_type;

        /* FIDs related fields */
        __u64 seq;
        __u32 oid;
        __u32 ver;

        /* former entry name */
        char name[LDISKFS_NAME_LEN];
};
```

In structure above, we add 3 fields:

seq - FIDs sequence number;

oid - object identifier in its sequence;

ver - object version.

These new fields are used to form *struct lu_fid* from them and pass it to client in lookup time for both, cross-ref and regular lookup cases as shown in section 4.3.2.

Adding new fields implies adjusting *rec_len* by sum of sizes new fields. This also means that usual ext3 fsck will not be confused and rather ignore new field. Having new fields in *struct ldiskfs_dir_entry* means, that lustre fsck should know them and fix in checking time if needed.

6.1.2 Stripping info

The stripping info structure, which is saved on MDS in inode EA and holds file's stripping info looks like the following:

```
struct lov_mds_md {
        __u32 lmm_magic;
        __u32 lmm_pattern;
        struct lu_fid lmm_fid;
        __u32 lmm_stripe_size;
        __u32 lmm_stripe_count;
        struct lov_ost_data lmm_objects[0];
};
```

And *struct lov_ost_data* which holds per-stripe info looks like the following:

```
struct lov_ost_data {
        struct lu_fid l_fid;
        __u32 l_ost_gen;
        __u32 l_ost_idx;
};
```

Changes in disk structures should be taken into account by fsck. All this is part of "FIDs Interoperability" documents.

6.2 Wire Format Changes

Wire format is going to change in the following aspects:

- all fields in wire structures use *struct lu_fid* instead of former struct ll_fid and/or oid and ogr for object identification. This also changes size of RPCs;
- valid flags may contain OBD_MD_FLID also for identifying that RPC contains *struct lu_fid*.

6.3 Network Protocol Compatibility, Interoperability

This is matter of separate documents, because it is big part, needs to be worked out carefully.