# FLD DLD

Yury Umanets

22nd June, 2006

## Contents

# 1 Introduction

This document describes FLD (FIDs Location Database) which is used in new lustre to provide location information as for what node specified FID lives on. This is the consequence of new FIDs design, which says that FID has no locality or addressing information in it to make the migration easy. This is why FLD is needed. In general, its purpose is to resolves passed FIDs into MDT server number. I said in general, because in details, it does translate sequence number (FID belong to) into MDT number, using the fact, that whole sequence belongs to same MDT.

# 2 Functional Specification

## 2.1 Abstract

FLD is distributed database, scattered among all server nodes in cluster and each node stores part of it. On lower layers, server uses special object index API provided by OSD to save FLD database entries. In context of this document "client" and "servers" always refer to FLD client and FLD server if not specified especially.

FLD module has two parts:

- server side FLD;
- client side FLD.

Server side FLD starts special service which purpose is to handle FLD RPCs. And client side provides API for communicating with server side FLD and caching.

In details, FLD functions like the following:

- all servers start server part FLD service. Its purpose is to handle FLD RPCs and communicate with backing store write/write FLD data;
- each server stores own part of distributes FLD index using backing store API, provided by lower layers on server;
- each client connects to all servers storing FLD at startup. As FLD for MD stack is only needed in multi-MDT configurations, client FLD is started in CMM and LMV modules which need to know where to forward some operation and what node does particular FID live on. Thus, FLD may use already connected exports passed to it from CMM and LMV modules which manage them;
- when client FLD is requested to perform SEQ->MDT resolution, it checks cache first of all. If cache does not contain needed FLD entry, client sends FLD RPC to server;
- when client FLD has to modify FLD index, it also modifies cache to add recently used/created entry to it and thus, allow subsequent lookups find it. In delete case, FLD client also deletes FLD entry from cache;
- in all cases that client FLD has to send RPC to some server, it should choose correct target export from targets list (see below) so that, FLD index is distributes among all FLD servers. To do that, client uses special hash function which is chosen in client FLD init time. All clients should use the same hash function;
- currently there are two hash functions supported: Round Robin and DHT (Distributed Hash Table). Using DHT is preferable option, because it allows to not change hash values after number of MDTs is changed. Thus, it allows to add new MDTs into alive cluster on fly.

## 2.2 Structures

The following are server side and client side FLD structures.

```
struct lu_server_fld {
        struct proc_dir_entry   *fld_proc_entry;
        struct ptlrpc_service   *fld_service;
        struct dt_device        *fld_dt;
        struct dt_object        *fld_obj;
        struct lu_fid            fld_fid;
};
```

**fld_proc_entry** - FLD service procfs entry, which purpose is that userspace programs and test scripts may access various tunables.

**fld_service** - pointer to FLD service started in MDT startup time.

**fld_dt** - service side device needed to access object index API.

**fld_obj** - service side object needed to access object index API.

**fld_fid** - /fld index file/directory FID, it is needed in initialization time.

This is client side FLD structures.

Types of hashes client side may use for choosing MDT to store new FLD entry on it.

```
enum {
        LUSTRE_CLI_FLD_HASH_DHT = 0,
        LUSTRE_CLI_FLD_HASH_RRB,
        LUSTRE_CLI_FLD_HASH_LAST
};

typedef int (*fld_hash_func_t) (struct lu_client_fld *, __u64);
```

This is FLD hash function structure. It contains name of hash and pointer to hashing function. What hash to use it chosen in client FLD startup time. These days only two functions are supported:

- DHT (Distributed Hash Table);
- RRB (Round Robin).

```
struct lu_fld_hash {
        const char              *fh_name;
        fld_hash_func_t          fh_func;
};
```

**fh_name** - hash function name.

**fh_func** - pointer to hash function.

```
struct lu_client_fld {
        struct proc_dir_entry   *fld_proc_entry;
        struct list_head         fld_exports;
        struct lu_fld_hash      *fld_hash;
        int                      fld_count;
        spinlock_t               fld_lock;
};
```

**fld_proc_entry** - client side procfs entry, needed to access various tunables from userspace.

**fld_exports** - list of all MDT exports that FLD should be distributed among.

**fld_hash** - hash function chosen by passed hash type in startup time.

**fld_count** - number of exports FLD knows about.

**fld_lock** - lock protecting fld_exports list.

## 2.3 Server side API

On server side, each MDT is supposed to start FLD service on startup and stop it when server is stopped.

The following tasks are main for server side FLD:

- start service and receive and handle FLD RPCs from FLD clients;
- maintain FLD index, using indexing API from lower layers.

The following functions are server side API functions which main purpose is to start and stop FLD service.

### 2.3.1 fld_server_init()

**Parameters**

```
int
fld_server_init(struct lu_server_fld *fld,
                const struct lu_context *ctx,
                const struct dt_device *dt,
                const char *uuid)
```

**fld** - pointer to allocated *struct lu_server_fld*.

**ctx** - server side thread context.

**dt** - server side device to access indexing API.

**uuid** - uuid of node FLD service is running on.

**Return value**

Function returns zero on success and error code otherwise.

**Description**

This function does two main tasks:

- start the FLD server side service for handling FLD RPCs;
- initialize passed @fld, that is assign some fields and zero out others.

### 2.3.2 fld_server_fini()

**Parameters**

```
void
fld_server_fini(struct lu_server_fld *fld,
                const struct lu_context *ctx);
```

**fld** - pointer to initialized *struct lu_server_fld*.

**ctx** - server side thread context.

**Return values**

No return values.

**Description**

This function does two main tasks:

- stop the FLD server side service;
- finalize all fields.

## 2.4 Client side API

On client side, FLD client performs the following main tasks:

- maintain list of servers which holds parts of distributed FLD index, choose correct target using hash function. DHT hash is preferred one, because allows to add new MDTs into alive cluster. But currently also Round Robin hash is used for testing purposes.
- maintain client side FLD cache, to not send RPCs each time as client needs to resolve FID into MDT number where the FIDs lives (FIDs home MDT). In fact FLD uses one of FIDs invariants, which says that all FIDs belong to the same sequence live on the same MDT. That means that FLD as lookup key uses not whole FID, but rather its sequence number. See section 6.1 for details if FLD index format;
- perform RPCs to MDT to get indexing information or to modify FLD index when new sequence is started on client.

The following is client side API functions.

### 2.4.1 fld_client_init()

**Parameters**

```
int
fld_client_init(struct lu_client_fld *fld,
                int hash, const char *uuid);
```

**fld** - allocate *struct lu_client_fld*.

**hash** - type of hash function to be used in client FLD to distribute FLD index among MDTs.

**uuid** - uuid of node client FLD is running on.

**Return values**

Function returns zero on success and error code otherwise.

**Description**

Function initializes client side FLD. It assigns used hash function and initializes all the other fields. After this is done, @fld may be used with other client side FLD API functions.

### 2.4.2 fld_client_fini()

**Parameters**

```
void
fld_client_fini(struct lu_client_fld *fld);
```

**fld** - initialized client side FLD struct.

**Return values**

No return values.

**Description**

This function turn off client side FLD and free all the memory assigned with it.

### 2.4.3 fld_client_create()

**Parameters**

```
int
fld_client_create(struct lu_client_fld *fld,
                  seqno_t seq, mdsno_t mds);
```

**fld** - initialized *struct lu_client_fid*.

**seq** - sequence number which should be added to FLD index and which may be looked up later.

**mds** - mds number which should be added to FLD index and where @seq will live. Pair *seq* and *mds* form FLD entry where *seq* is entry key and *mds* is entry record. This mds number will be looked later using *seq* as the key.

#### Return values

The function returns zero on success and error code otherwise.

#### Description

This function performs two main tasks:

- send FLD create RPC to correct MDT to create FLD index entry on it and thus let passed mds number be found by passed sequence number;
- add *seq/mds* entry into local cache upon successful adding it on MDT.

In the case such mapping already exists, server will return error and this error will be translated to caller. New entry will not be added to index and local cache.

### 2.4.4 fld_client_delete()

#### Parameters

```
int
fld_client_delete(struct lu_client_fld *fld,
                  seqno_t seq);
```

**fld** - initialized *struct lu_client_fld*.

**seq** - sequence number to be removed from FLD index.

#### Return values

Function returns zero on success and error code otherwise.

#### Description

This function removes FLD index entry using @seq as key from index on corresponding MDT and after that removes it from local cache.

This is not yet clear when to use this function. Issue is in tracking that some sequences contain no more objects and may be removed from index. One of possible solution could be introducing refcounting, but this is still idea from top of the head and needs to be reworked carefully in separate design document.

### 2.4.5 fld_client_lookup()

**Parameters**

```
int
fld_client_lookup(struct lu_client_fld *fld,
                  seqno_t seq, mdsno_t *mds);
```

**fld** - initialized *struct lu_client_fld*.

**seq** - sequence number which is looked for the MDT number.

**mds** - pointer to __u64 where we should store found MDT number (home MDT for passed *seq*).

**Return values**

Returns zero on success and error code otherwise. On error code *mds* parameter may contain anything and should not be trusted.

### 2.4.6 fld_client_add_target()

**Parameters**

```
int
fld_client_add_target(struct lu_client_fld *fld,
                      struct obd_export *exp);
```

**fld** - initialized *struct lu_client_fld*.

**exp** - OBD export pointing to MDT which we want to add distributed FLD index storage.

**Return values**

Returns zero on success and error code otherwise.

**Description**

This function adds passed export to local exports list which is used for the following purposes:

- choose correct export to send lookup RPCs.
- choose correct export to send create/delete RPCs.

  Correct export is chosen by hash function selected in client FLD initialization time.

### 2.4.7 fld_client_del_target()

**Parameters**

```
int
fld_client_del_target(struct lu_client_fld *fld,
                      struct obd_export *exp);
```

**fld** - initialized *struct lu_client_fld*.

**exp** - export to be removed from list of FLD exports.

**Return values**

Returns zero on success and error code otherwise.

**Description**

This function does opposite actions to *fld_client_add_target()* one. That is removing passed export from FLD client exports list.

## 2.5 Cache API

FLD client contains FLD cache, which purpose is to hold recently used resolved sequence/mdsnum pairs to avoid requesting them again from the server. Client FLD works with cache using the following API functions.

### 2.5.1 fld_cache_init()

**Parameters**

```
struct fld_cache_info *fld_cache_init(int size);
```

**size** - cache hash table size, default value 256 is used.

**Return values**

Returns pointer to allocated *struct fld_cache_info* on success and error otherwise. Error code may be extracted using macro PTR_ERR().

**Description**

This function initializes FLD cache. It allocates *struct fld_cache_info* and initializes its fields. Used once in client FLD initialization process.

### 2.5.2 fld_cache_fini()

**Parameters**

```
void fld_cache_fini(struct fld_cache_info *cache);
```

**cache** - pointer to allocated by *fld_cache_init()* instance of *struct fld_cache_info*.

**Return values**

No return values.

**Description**

This function finalizes using of FLD cache. Should be called at the end of using it.

### 2.5.3 fld_cache_insert()

**Parameters**

```
int
fld_cache_insert(struct fld_cache_info *cache,
                 seqno_t seq, mdsno_t mds);
```

**cache** - pointer to initialized *struct fld_cache_info*.

**seq** - sequence number to be used as key in new created FLD entry.

**mds** - MDT number to be used as value in new created FLD entry.

**Return values**

Returns zero on success and error code otherwise.

**Description**

This function inserts new FLD entry into FLD cache. For key it uses passed @seq number and for value which can be found by key mds number if used (stored in @mds).

### 2.5.4 fld_cache_delete()

**Parameters**

```
void
fld_cache_delete(struct fld_cache_info *cache,
                 seqno_t seq);
```

**cache** - pointer to initialized *struct fld_cache_info*.

**seq** - sequence number to be used as key to delete FLD entry.

**Return values**

No return values.

**Description**

This function deletes FLD entry (seq/mds pair) from FLD cache.

#### 2.5.5 fld_cache_lookup()

**Parameters**

```
struct fld_cache_entry *
fld_cache_lookup(struct fld_cache_info *cache,
                 seqno_t seq);
```

**cache** - pointer to initialized *struct fld_ cache_ info.*

**seq** - sequence number to be used as key in in lookup.

**Return values**

Returns found cache FLD entry on success and NULL otherwise.

**Description**

This function is used to lookup for cached FLD entries by key (sequence number).

# 3 Use Cases

This chapter describes use cases for server and client side FLD API.

## 3.1 Client side

Here is client side FLD API use cases.

### 3.1.1 FLD client init

On client side, FLD is initialized in LMV module. This looks like the following:

```
static int
lmv_setup(struct obd_device *obd,
          struct lustre_cfg *lcfg)
{
        int rc;
        ENTRY;

        ...

        rc = fld_client_init(&lmv->lmv_fld,
                             "LMV_UUID",
```

```
                                LUSTRE_CLI_FLD_HASH_RRB);
        if (rc) {
                CERROR("can't init FLD, err %d\n",
                       rc);
                GOTO(out_free_datas, rc);
        }

        ...
}
```

### 3.1.2 Create FLD entry

Each time as client needs to allocate new FID, it requests client sequence manager for sequence number in which this FID may be allocated. In the case sequence number has changed and new one is allocated, FLD wants to create new FLD entry for it. 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 underlying tgt layer to allocate new fid */
        rc = obd_fid_alloc(lmv->tgts[mds].ltd_exp, fid, hint);
        /* client switches to new sequence, setup fld */
        if (rc == -ERESTART) {
                rc = fld_client_create(&lmv->lmv_fld,
                                       fid_seq(fid),
                                       mds);
                if (rc) {
                        CERROR("can't create fld entry, "
                               "rc %d\n", rc);
                }
```

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

### 3.1.3 Lookup FLD entry

Each time as client's LMV module has to make decision as for what node particular operation RPC should be forwarded, it requests FLD client to resolve passed FID (more precisely its sequence number) into MDS number. This looks like the following:

```
int
lmv_fld_lookup(struct obd_device *obd,
               const struct lu_fid *fid)
{
        struct lmv_obd *lmv = &obd->u.lmv;
        mdsno_t mds;
        int rc;
        ENTRY;

        LASSERT(fid_is_sane(fid));
        rc = fld_client_lookup(&lmv->lmv_fld,
                               fid_seq(fid), &mds);
        if (rc) {
                CERROR("can't find mds by seq "LPU64", rc %d\n",
                       fid_seq(fid), rc);
                RETURN(rc);
        }
        CDEBUG(D_INFO, "LMV: got MDS "LPU64
               " for sequence: "LPU64"\n",
               mds, fid_seq(fid));
        if (mds >= lmv->desc.ld_tgt_count) {
                CERROR("Got invalid mdsno: "LPU64" (max: %d)\n",
                       mds, lmv->desc.ld_tgt_count);
                mds = (__u64)-EINVAL;
        }
        RETURN((int)mds);
}

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, i;
        ENTRY;
        rc = lmv_check_connect(obd);
        if (rc)
                RETURN(rc);
        i = lmv_fld_lookup(obd, fid);
        if (i < 0)
                RETURN(i);
        LASSERT(i < lmv->desc.ld_tgt_count);
        rc = md_getattr(lmv->tgts[i].ltd_exp, fid,
                        valid, ea_size, request);
        if (rc)
                RETURN(rc);
        obj = lmv_obj_grab(obd, fid);
        CDEBUG(D_OTHER, "GETATTR for "DFID3" %s\n",
               PFID3(fid), obj ? "(splitted)" : "");

        ...
}
```

### 3.1.4 Adding MDT target to FLD

FLD client initialization does not mean that FLD is fully functional after that. It should be given also set of MDT targets to distribute FLD among them and use them for lookups.

FLD uses the same exports as LMV does, and thus, LMV lets FLD know about them in the time when they are added to LMV and connected. This looks like the following:

```
int
lmv_connect_mdc(struct obd_device *obd,
                struct lmv_tgt_desc *tgt)
{
        struct lmv_obd *lmv = &obd->u.lmv;
        struct obd_uuid *cluuid = &lmv->cluuid;
        struct obd_connect_data *mdc_data = NULL;
        struct obd_uuid lmv_mdc_uuid = { "LMV_MDC_UUID" };
        struct lustre_handle conn = {0, };
        struct obd_device *mdc_obd;
        struct obd_export *mdc_exp;
        int rc;
#ifdef __KERNEL__
        struct proc_dir_entry *lmv_proc_dir;
#endif
        ENTRY;
```

```
        if (obd_uuid_equals(&tgt->uuid, cluuid)) {
                CDEBUG(D_CONFIG, "don't connect back to %s\n",
                       cluuid->uuid);
                RETURN(0);
        }

        mdc_obd = class_find_client_obd(&tgt->uuid,
                                        LUSTRE_MDC_NAME,
                                        &obd->obd_uuid);
        if (!mdc_obd) {
                CERROR("target %s not attached\n",
                       tgt->uuid.uuid);
                RETURN(-EINVAL);
        }

        if (!mdc_obd->obd_set_up) {
                CERROR("target %s not set up\n",
                       tgt->uuid.uuid);
                RETURN(-EINVAL);
        }
        rc = obd_connect(&conn, mdc_obd, &lmv_mdc_uuid,
                         &lmv->conn_data);
        if (rc) {
                CERROR("target %s connect error %d\n",
                       tgt->uuid.uuid, rc);
                RETURN(rc);
        }

        mdc_exp = class_conn2export(&conn);

        fld_client_add_target(&lmv->lmv_fld, mdc_exp);

        ...
}
```

## 3.2 Server side

On server side there are almost the same needs about using FLD. First of all server should initialize it and then use some API methods.

### 3.2.1 FLD server init

```
static int
mdt_fld_init(const struct lu_context *ctx,
             const char *uuid, struct mdt_device *m)
```

```
{
        struct lu_site *ls;
        int rc;
        ENTRY;

        ls = m->mdt_md_dev.md_lu_dev.ld_site;

        OBD_ALLOC_PTR(ls->ls_fld);

        if (ls->ls_fld != NULL) {
                rc = fld_server_init(ls->ls_fld, ctx,
                                     uuid, m->mdt_bottom);
                if (rc) {
                        OBD_FREE_PTR(ls->ls_fld);
                        ls->ls_fld = NULL;
                }
        } else {
                rc = -ENOMEM;
        }
        RETURN(rc);
}

static int
mdt_init0(struct mdt_device *m,
          struct lu_device_type *t,
          struct lustre_cfg *cfg)
{
        int rc;
        ...
        rc = mdt_fld_init(&ctx, obd->obd_name, m);
        if (rc)
                GOTO(err_fini_stack, rc);

        ...
}
```

### 3.2.2 Lookup FLD entry

On server side there is CMM component, which needs FLD client service. It uses it like LMV for resolving FID (its sequence in fact) into MDT number where to forward operation RPCs. The using pattern is the same as in LMV case. Look at section 3.1.3 for details.

### 3.2.3 Adding MDT target to FLD

On server side, like on client, in setup time, CMM also has added all MDT targets in cluster that it has to work with. In same fashion like it is done for client nodes, CMM adds all targets to its FLD client. See section 3.1.4 for details.

# 4 Logic Specification

As was said above, FLD consists of two parts: client and server. This chapter describes how main important functions of both of them are implemented.

## 4.1 Server side

There are three important functions on server side which do all the work. They are the following:

- fld_server_init();
- fld_server_fini();
- fld_server_handle().

The following are details of implementation.

FLD service init function.

```
int
fld_server_init(struct lu_server_fld *fld,
                const struct lu_context *ctx,
                struct dt_device *dt)
{
        int rc;
        struct ptlrpc_service_conf fld_conf = {
                .psc_nbufs            = MDS_NBUFS,
                .psc_bufsize          = MDS_BUFSIZE,
                .psc_max_req_size     = FLD_MAXREQSIZE,
                .psc_max_reply_size   = FLD_MAXREPSIZE,
                .psc_req_portal       = FLD_REQUEST_PORTAL,
                .psc_rep_portal       = MDC_REPLY_PORTAL,
                .psc_watchdog_timeout = FLD_SERVICE_WATCHDOG_TIMEOUT,
                .psc_num_threads      = FLD_NUM_THREADS
        };
        ENTRY;
        fld->fld_dt = dt;
        lu_device_get(&dt->dd_lu_dev);
```

```
        rc = fld_index_init(fld, ctx);
        if (rc == 0) {
                fld->fld_service =
                        ptlrpc_init_svc_conf(&fld_conf,
                                             fld_req_handle,
                                             LUSTRE_FLD0_NAME,
                                             fld->fld_proc_entry,
                                             NULL);
                if (fld->fld_service != NULL)
                        rc = ptlrpc_start_threads(NULL,
                                fld->fld_service,
                                LUSTRE_FLD0_NAME);
                else
                        rc = -ENOMEM;
        }
        if (rc != 0)
                fld_server_fini(fld, ctx);
        else
                CDEBUG(D_INFO|D_WARNING, "Server FLD\n");
        RETURN(rc);
}
EXPORT_SYMBOL(fld_server_init);
```

FLD service fini function.

```
void
fld_server_fini(struct lu_server_fld *fld,
                const struct lu_context *ctx)
{
        ENTRY;
        if (fld->fld_service != NULL) {
                ptlrpc_unregister_service(fld->fld_service);
                fld->fld_service = NULL;
        }
        if (fld->fld_dt != NULL) {
                lu_device_put(&fld->fld_dt->dd_lu_dev);
                fld_index_fini(fld, ctx);
                fld->fld_dt = NULL;
        }
        CDEBUG(D_INFO|D_WARNING, "Server FLD\n");
        EXIT;
}
EXPORT_SYMBOL(fld_server_fini);
```

FLD RPCs handler function.

```
static int
fld_server_handle(struct lu_server_fld *fld,
                  const struct lu_context *ctx,
                  __u32 opts, struct md_fld *mf)
{
        int rc;
        ENTRY;
        switch (opts) {
        case FLD_CREATE:
                rc = fld_index_insert(fld, ctx,
                                      mf->mf_seq,
                                      mf->mf_mds);
                break;
        case FLD_DELETE:
                rc = fld_index_delete(fld, ctx,
                                      mf->mf_seq);
                break;
        case FLD_LOOKUP:
                rc = fld_index_lookup(fld, ctx,
                                      mf->mf_seq,
                                      &mf->mf_mds);
                break;
        default:
                rc = -EINVAL;
                break;
        }
        RETURN(rc);
}
```

As it is seen from function above, FLD handles 3 operations:

- CREATE new FLD index entry;
- DELETE existing FLD index entry;
- LOOKUP existing FLD index entry.

### 4.1.1 Modifying FLD index

The following code performs FLD index setup and modification.
This function initializes /fld index to store FLD entries in it.

```
int
fld_index_init(struct lu_server_fld *fld,
               const struct lu_context *ctx)
{
        struct dt_device *dt = fld->fld_dt;
```

```
        struct dt_object *dt_obj;
        int rc;
        ENTRY;
        if (fld_key_registered == 0) {
                rc = lu_context_key_register(&fld_thread_key);
                if (rc != 0)
                        RETURN(rc);
        }
        fld_key_registered++;

        LASSERT(fld->fld_service == NULL);
        dt_obj = dt_store_open(ctx, dt, "fld", &fld->fld_fid);
        if (!IS_ERR(dt_obj)) {
                fld->fld_obj = dt_obj;
                rc = dt_obj->do_ops->do_object_index_try(ctx, dt_obj,
                        &fld_index_features);
                if (rc == 0)
                        LASSERT(dt_obj->do_index_ops != NULL);
                else
                        CERROR("fld is not an index!\n");
        } else {
                CERROR("Cannot find fld obj %lu \n",
                       PTR_ERR(dt_obj));
                rc = PTR_ERR(dt_obj);
        }
        RETURN(rc);
}
```

Finalize FLD index. This is called when FLD service is shutting down.

```
void
fld_index_fini(struct lu_server_fld *fld,
               const struct lu_context *ctx)
{
        ENTRY;
        if (fld->fld_obj != NULL) {
                lu_object_put(ctx, &fld->fld_obj->do_lu);
                fld->fld_obj = NULL;
        }
        if (fld_key_registered > 0) {
                if (--fld_key_registered == 0)
                        lu_context_key_degister(&fld_thread_key);
        }
        EXIT;
}
```

This function inserts new FLD entry into FLD index.

```
int
fld_index_insert(struct lu_server_fld *fld,
                 const struct lu_context *ctx,
                 fidseq_t seq, mdsno_t mds)
{
        struct dt_device *dt = fld->fld_dt;
        struct dt_object *dt_obj = fld->fld_obj;
        struct txn_param txn;
        struct thandle *th;
        int rc;
        ENTRY;
        txn.tp_credits = FLD_TXN_INDEX_INSERT_CREDITS;
        th = dt->dd_ops->dt_trans_start(ctx, dt, &txn);
        rc = dt_obj->do_index_ops->dio_insert(ctx, dt_obj,
                                              fld_rec(ctx, mds),
                                              fld_key(ctx, seq), th);
        dt->dd_ops->dt_trans_stop(ctx, th);
        RETURN(rc);
}
```

This function deletes FLD entry from FLD index.

```
int
fld_index_delete(struct lu_server_fld *fld,
                 const struct lu_context *ctx,
                 fidseq_t seq)
{
        struct dt_device *dt = fld->fld_dt;
        struct dt_object *dt_obj = fld->fld_obj;
        struct txn_param txn;
        struct thandle *th;
        int rc;
        ENTRY;
        txn.tp_credits = FLD_TXN_INDEX_DELETE_CREDITS;
        th = dt->dd_ops->dt_trans_start(ctx, dt, &txn);
        rc = dt_obj->do_index_ops->dio_delete(ctx, dt_obj,
                                              fld_key(ctx, seq), th);
        dt->dd_ops->dt_trans_stop(ctx, th);
        RETURN(rc);
}
```

This function performs lookup of FLD entry in FLD index.

```
int
fld_index_lookup(struct lu_server_fld *fld,
```

```
                const struct lu_context *ctx,
                fidseq_t seq, mdsno_t *mds)
{
        struct dt_object *dt_obj = fld->fld_obj;
        struct dt_rec    *rec = fld_rec(ctx, 0);
        int rc;
        ENTRY;
        rc = dt_obj->do_index_ops->dio_lookup(ctx, dt_obj, rec,
                                              fld_key(ctx, seq));
        if (rc == 0)
                *mds = be64_to_cpu(*(__u64 *)rec);
        RETURN(rc);
}
```

## 4.2 Client side

On client side, as was said above, FLD should perform three main tasks. See chapter 2.4 for details.

Client FLD init and fini functions.

```
int
fld_client_init(struct lu_client_fld *fld,
                int hash)
{
        int rc = 0;
        ENTRY;
        LASSERT(fld != NULL);
        if (!hash_is_sane(hash)) {
                CERROR("wrong hash function 0x%x\n", hash);
                RETURN(-EINVAL);
        }
        INIT_LIST_HEAD(&fld->fld_exports);
        spin_lock_init(&fld->fld_lock);
        fld->fld_hash = &fld_hash[hash];
        fld->fld_count = 0;
        CDEBUG(D_INFO|D_WARNING,
               "Client FLD initialized, using \"%s\" hash\n",
               fld->fld_hash->fh_name);
        RETURN(rc);
}
EXPORT_SYMBOL(fld_client_init);

void
fld_client_fini(struct lu_client_fld *fld)
{
        struct fld_target *target, *tmp;
```

```
        ENTRY;
#ifdef LPROCFS
        fld_client_proc_fini(fld);
#endif
        spin_lock(&fld->fld_lock);
        list_for_each_entry_safe(target, tmp,
                                 &fld->fld_targets, fldt_chain) {
                fld->fld_count--;
                list_del(&target->fldt_chain);
                class_export_put(target->fldt_exp);
                OBD_FREE_PTR(target);
        }
        spin_unlock(&fld->fld_lock);
        if (fld->fld_cache != NULL) {
                fld_cache_fini(fld->fld_cache);
                fld->fld_cache = NULL;
        }
        CDEBUG(D_INFO|D_WARNING, "Client FLD finalized\n");
        EXIT;
}
EXPORT_SYMBOL(fld_client_fini);
```

### 4.2.1 MDT exports list

FLD maintains export list and has few functions for adding, deleting and looking up the export by sequence number. The reason is that, first of all, when FLD client starts, there is no yet information about MDT exports. Adding exports is done after all the stuff is initialized. This is the nature of mountconf based configurations. Also, new MDT may be added to alive cluster and thus, all components should be prepared for that.

The following struct is MDT target which unites MDT export and its number to be used in all target related functions. This struct is linked into FLD targets list and used later in traversals.

```
struct fld_target {
        struct list_head   fldt_chain;
        struct obd_export *fldt_exp;
        __u64              fldt_idx;
};
```

Following function is used to find the export corresponding to passed sequence number. First of all it, it uses hash function to get MDT number where the operation should be forwarded to. This guarantees that FLD will be distributed among all the registered MDTs.

```
static struct fld_target *
fld_client_get_target(struct lu_client_fld *fld,
```

```
                        seqno_t seq)
{
        struct fld_target *target;
        int hash;
        ENTRY;
        LASSERT(fld->fld_hash != NULL);
        spin_lock(&fld->fld_lock);
        hash = fld->fld_hash->fh_func(fld, seq);
        list_for_each_entry(target,
                            &fld->fld_targets, fldt_chain) {
                if (target->fldt_idx == hash) {
                        spin_unlock(&fld->fld_lock);
                        RETURN(target);
                }
        }
        spin_unlock(&fld->fld_lock);
        RETURN(NULL);
}
```

This function simply adds new export to the list.

```
int
fld_client_add_target(struct lu_client_fld *fld,
                      struct obd_export *exp)
{
        struct client_obd *cli = &exp->exp_obd->u.cli;
        struct fld_target *target, *tmp;
        ENTRY;
        LASSERT(exp != NULL);
        CDEBUG(D_INFO|D_WARNING, "FLD(cli): adding export %s\n",
              cli->cl_target_uuid.uuid);
        OBD_ALLOC_PTR(target);
        if (target == NULL)
                RETURN(-ENOMEM);
        spin_lock(&fld->fld_lock);
        list_for_each_entry(tmp, &fld->fld_targets, fldt_chain) {
                if (obd_uuid_equals(&tmp->fldt_exp->exp_client_uuid,
                                    &exp->exp_client_uuid))
                {
                        spin_unlock(&fld->fld_lock);
                        OBD_FREE_PTR(target);
                        RETURN(-EEXIST);
                }
        }
        target->fldt_exp = class_export_get(exp);
```

```
        /* XXX: should this me MDT number? */
        target->fldt_idx = fld->fld_count;
        list_add_tail(&target->fldt_chain,
                      &fld->fld_targets);
        fld->fld_count++;
        spin_unlock(&fld->fld_lock);
        RETURN(0);
}
EXPORT_SYMBOL(fld_client_add_target);
```

And this one removes passed export from the list.

```
int
fld_client_del_target(struct lu_client_fld *fld,
                      struct obd_export *exp)
{
        struct fld_target *target, *tmp;
        ENTRY;
        spin_lock(&fld->fld_lock);
        list_for_each_entry_safe(target, tmp,
                                 &fld->fld_targets,
                                 fldt_chain)
        {
                if (obd_uuid_equals(&target->fldt_exp->exp_client_uuid,
                                    &exp->exp_client_uuid))
                {
                        fld->fld_count--;
                        list_del(&target->fldt_chain);
                        class_export_put(target->fldt_exp);
                        spin_unlock(&fld->fld_lock);
                        OBD_FREE_PTR(target);
                        RETURN(0);
                }
        }
        spin_unlock(&fld->fld_lock);
        RETURN(-ENOENT);
}
EXPORT_SYMBOL(fld_client_del_target);
```

### 4.2.2 FLD client cache

Client FLD maintains last requests cache in order to not request last used location information from server, because sending RPCs is expensive.

Cache related structs look like the following:

```
struct fld_cache_entry {
        struct hlist_node  fce_list;
```

```
        mdsno_t             fce_mds;
        fidseq_t            fce_seq;
};
```

**fce_list** - list node to attach *struct fld_cache_entry* into *struct hlist_head*.

**fce_mds** - mds number cache entry describes.

**fce_seq** - sequence number living on *fce_mds* that cache entry describes.

```
struct fld_cache_info {
        struct hlist_head *fci_hash;
        spinlock_t         fci_lock;
        int                fci_hash_mask;
};
```

**fci_hash** - in-memory array of cached FLD entries (*struct fld_cache_entry*).

**fci_lock** - lock protecting *fci_hash* while lookup and modifications.

**fci_hash_mask** - mask which should be applied to all hashes used to chose correct bucket.

Cache API has three functions for working with it. They are the following:

- *fld_cache_insert()* - insert pair known *seq/mds* into the cache;
- *fld_cache_delete()* - delete pair *seq/mds* from cache by passed *seq*;
- *fld_cache_lookup()* - lookup for pair *seq/mds* by *seq*.

Implementation looks the following:

```
static int
fld_cache_insert(struct fld_cache_info *fld_cache,
                 seqno_t seq, __u64 mds)
{
        struct fld_cache_entry *flde, *fldt;
        struct hlist_head *bucket;
        struct hlist_node *scan;
        int rc = 0;
        ENTRY;
        OBD_ALLOC_PTR(flde);
        if (!flde)
                RETURN(-ENOMEM);
        bucket = fld_cache->fci_hash + (fld_cache_hash(seq) &
                                        fld_cache->fci_hash_mask);
        spin_lock(&fld_cache->fci_lock);
        hlist_for_each_entry(fldt, scan, bucket, fce_list) {
```

```
                if (fldt->fce_seq == seq)
                        GOTO(exit_unlock, rc = -EEXIST);
        }
        INIT_HLIST_NODE(&flde->fce_list);
        flde->fce_mds = mds;
        flde->fce_seq = seq;

        hlist_add_head(&flde->fce_list, bucket);
        EXIT;
exit_unlock:
        spin_unlock(&fld_cache->fci_lock);
        if (rc != 0)
                OBD_FREE_PTR(flde);
        return rc;
}
```

This function removes FLD index entry from the cache.

```
static void
fld_cache_delete(struct fld_cache_info *fld_cache,
                 seqno_t seq)
{
        struct fld_cache_entry *flde;
        struct hlist_head *bucket;
        struct hlist_node *scan;
        ENTRY;
        bucket = fld_cache->fci_hash + (fld_cache_hash(seq) &
                                        fld_cache->fci_hash_mask);
        spin_lock(&fld_cache->fci_lock);
        hlist_for_each_entry(flde, scan, bucket, fce_list) {
                if (flde->fce_seq == seq) {
                        hlist_del_init(&flde->fce_list);
                        OBD_FREE_PTR(flde);
                        GOTO(out_unlock, 0);
                }
        }
        EXIT;
out_unlock:
        spin_unlock(&fld_cache->fci_lock);
}
```

This function performs lookup for cached FLD entry by sequence number.

```
static struct fld_cache_entry *
fld_cache_lookup(struct fld_cache_info *fld_cache, seqno_t seq)
{
```

```
        struct fld_cache_entry *flde;
        struct hlist_head *bucket;
        struct hlist_node *scan;
        ENTRY;
        bucket = fld_cache->fci_hash + (fld_cache_hash(seq) &
                                        fld_cache->fci_hash_mask);
        spin_lock(&fld_cache->fci_lock);
        hlist_for_each_entry(flde, scan, bucket, fce_list) {
                if (flde->fce_seq == seq) {
                        spin_unlock(&fld_cache->fci_lock);
                        RETURN(flde);
                }
        }
        spin_unlock(&fld_cache->fci_lock);
        RETURN(NULL);
}
```

### 4.2.3 RPCs sending

There is main function which performs RPC sending over the network. It is used by three client side API functions to send CREATE, DELETE and LOOKUP RPCs to the server.

API functions are described in "Functional specification" and they are the following:

- *fld_client_create()* - create FLD index entry;
- *fld_client_delete()* - delete FLD index entry;
- *fld_client_lookup()* - lookup for FLD index entry using sequence number.

Implementation looks like the following:

```
static int
fld_client_rpc(struct obd_export *exp,
               struct md_fld *mf, __u32 fld_op)
{
        int size[2] = {sizeof(__u32), sizeof(struct md_fld)}, rc;
        int mf_size = sizeof(struct md_fld);
        struct ptlrpc_request *req;
        struct md_fld *pmf;
        __u32 *op;
        ENTRY;
        LASSERT(exp != NULL);
        req = ptlrpc_prep_req(class_exp2cliimp(exp),
                              LUSTRE_MDS_VERSION, FLD_QUERY,
                              2, size, NULL);
```

```
        if (req == NULL)
                RETURN(-ENOMEM);
        op = lustre_msg_buf(req->rq_reqmsg, 0, sizeof (*op));
        *op = fld_op;
        pmf = lustre_msg_buf(req->rq_reqmsg, 1, sizeof (*pmf));
        memcpy(pmf, mf, sizeof(*mf));
        req->rq_replen = lustre_msg_size(1, &mf_size);
        req->rq_request_portal = MDS_FLD_PORTAL;
        rc = ptlrpc_queue_wait(req);
        if (rc)
                GOTO(out_req, rc);
        pmf = lustre_swab_repbuf(req, 0, sizeof(*pmf),
                                 lustre_swab_md_fld);
        *mf = *pmf;
 out_req:
        ptlrpc_req_finished(req);
        RETURN(rc);
}
```

Create FLD index entry. After it is created on server - create it in local cache.

```
int
fld_client_create(struct lu_client_fld *fld,
                  seqno_t seq, mdsno_t mds)
{
        struct fld_target *target;
        struct md_fld      md_fld;
        __u32 rc;
        ENTRY;
        target = fld_client_get_target(fld, seq);
        if (!target)
                RETURN(-EINVAL);
        md_fld.mf_seq = seq;
        md_fld.mf_mds = mds;
        rc = fld_client_rpc(target->fldt_exp, &md_fld, FLD_CREATE);
#ifdef __KERNEL__
        if (rc  == 0)
                rc = fld_cache_insert(fld_cache, seq, mds);
#endif
        RETURN(rc);
}
EXPORT_SYMBOL(fld_client_create);
```

Delete FLD index entry.

```
int
fld_client_delete(struct lu_client_fld *fld,
                  seqno_t seq)
{
        struct fld_target *target;
        struct md_fld      md_fld;
        __u32 rc;
#ifdef __KERNEL__
        fld_cache_delete(fld_cache, seq);
#endif
        target = fld_client_get_target(fld, seq);
        if (!target)
                RETURN(-EINVAL);
        md_fld.mf_seq = seq;
        md_fld.mf_mds = 0;
        rc = fld_client_rpc(target->fldt_exp, &md_fld, FLD_DELETE);
        RETURN(rc);
}
EXPORT_SYMBOL(fld_client_delete);
```

Lookup FLD index entry.

```
static int
fld_client_get(struct lu_client_fld *fld,
               seqno_t seq, mdsno_t *mds)
{
        struct fld_target *target;
        struct md_fld md_fld;
        int rc;
        ENTRY;
        target = fld_client_get_target(fld, seq);
        if (!target)
                RETURN(-EINVAL);
        md_fld.mf_seq = seq;
        rc = fld_client_rpc(target->fldt_exp,
                            &md_fld, FLD_LOOKUP);
        if (rc == 0)
                *mds = md_fld.mf_mds;
        RETURN(rc);
}
int
fld_client_lookup(struct lu_client_fld *fld,
                  seqno_t seq, mdsno_t *mds)
{
#ifdef __KERNEL__
        struct fld_cache *fld_entry;
```

```
#endif
        int rc;
        ENTRY;
#ifdef __KERNEL__
        /* lookup it in the cache */
        fld_entry = fld_cache_lookup(fld_cache, seq);
        if (fld_entry != NULL) {
                *mds = fld_entry->fld_mds;
                RETURN(0);
        }
#endif
        /* can not find it in the cache */
        rc = fld_client_get(fld, seq, mds);
        if (rc)
                RETURN(rc);
#ifdef __KERNEL__
        if (rc == 0)
                rc = fld_cache_insert(fld_cache, seq, *mds);
#endif
        RETURN(rc);
}
EXPORT_SYMBOL(fld_client_lookup);
```

# 5 State Specification

## 5.1 Resources Involved and Their State

There are two types of resources:

- /fld file on server side, which contains FLD index;
- FLD cache on client side.

### 5.1.1 /fld file

This resource changes each time as new entry is inserted into index. Thus, this change is state transition and should be protected. Protection is done by object index API and fortunately FLD as a user of the API should do nothing.

### 5.1.2 FLD cache

On client, FLD cache is resource which should be also protected while lookup, insert and/or delete entries in it. This is done by special *->fci_lock* in *struct fld_cache_info*.

### 5.2 Locking

No special locking schema needed except what is said in sections above.

### 5.3 Recovery

Recovery is based on few assumptions. They are the following:

- all MDTs do any modifications to FLD index in transactions based style. That is open transaction before modifying the index and close (commit) transaction after that is done.
- all requests which did not fit into last committed transaction should be replied by client in the case of recovery.

Then, in case of MDT crash, and having situation that client send an FLD RPC and MDT did not commit it, client will re-send it again and MDT creates FLD entry.

## 6 Environment

### 6.1 Disk Format Changes

This chapter describes disk changes due to FLD service.

### 6.2 Basics

Backing store filesystem should have created /fld file before first mount. This file also should contain some default FLD entries for ROOT and may be other objects. File for storing FLD index may be created and set up using the following commands after creating FS itself.

These commands create /fld index file with key size 8 bytes (sequence number) and record size also 8 bytes (mds number).

```
echo "making /mnt/mdt/fld"
mount -text3 $MDS_DEV /mnt/mdt
$CREATE_IAM -f lfix -k 8 -r 8 > /mnt/mdt/fld
umount /mnt/mdt
```

And these commands setup /fld index with default values for ROOT directory. Command orders to create index entry with key 2 and record 0. This means that sequence number 2 (ROOT object sequence) lives on MDT 0.

```
echo "setup /mnt/mdt/fld"
mount -tldiskfs $MDS_DEV /mnt/mdt
$IAM_UT -i -K 0000000000000002 -R 0000000000000000 < /mnt/mdt/fld
umount /mnt/mdt
```

Thus, /fld file format is very simple. It contains number of records which may be found by key. Key in this case is sequence number and record is MDT number this sequence lives.

## 6.3 Issues

Thus, number of record should be number of sequences living on particular MDT. As sequences may be various of size and nothing to enforce them to be exactly predefined size (say 10 000 objects), /fld index may become too big in big installations. This issue may be even worse, because it is not clear when some record may be deleted.

As a possible solution may be idea to store into /fld index not sequence number as a key, but rather range of sequences. This idea is inspired by the fact, that Sequence Manager allocates to clients so called meta-sequences (see Sequence Management DLD). And meta-sequence, which is in fact range of sequences live on same MDT. Thus, we are sure, that whole meta-sequence (say 10000 sequences) live on same MDT and may be indexed only by having one record in /fld index file.