FLD (FIDs Location Database) HLD

Yury Umanets, Nikita Danilov

3rd February 2006

Contents

1 Introduction[1]

With the introduction of FIDs as the sole high-level identifier for Lustre objects, clients and metadata servers need a way to find out at what server object with given FID is stored (or, actually, to what server requests for the operations on this object are to be sent). To this end FIDs Location Database (FLD) is established. Few preliminary definitions and considerations:

1. FID is a unique file identifier stable and unique across cluster and during whole cluster life-time. Fid structure is: *fid = (f-sequence, f-number)*, where f-sequence is *fid sequence*, and *f-number* is a *fid number* within sequence;

2. every cluster node capable of issuing meta-data related requests to other nodes (that is: every client node and every metadata server in CMD configuration), receives new cluster-widely unique *fid sequence* every time it joins the cluster (that is: during normal boot/mount, on server reboot after failure and when re-joining cluster after eviction). Each node that may request objects creation needs sequence. It has separate sequence for each MDT and each OST in cluster;

3. node uses received sequences to generate fids for all new objects whose creation it requests;

4. files created within same sequence should be located on the same MDT referred to as "home MDT" of this sequence;

5. FID may be a part of larger structure called lustre_id that, in addition to the former, contains data sufficient to locate file in the cluster. lustre_id may contain cached MDT number, etc., to not resolve it each time new operation is performed;

6. file, identified by FID, is stored on metadata server, called "home server" of this FID (and this file).

[1] This section is not mandatory. It may contain some introduction points, like scope of HLD, etc.

2 Requirements[2]

Having FIDs as file identifier in cluster, and the fact that FID does not contain address or location information, we faced with need to find correct MDT where inode with designated FID lives. Thus, requirements are the following:

- establish distributed FIDs Location Database (FLD);
- make API which uses FLD to resolve FID into MDT;
- after migration, files should be able to find in new location.

3 Functional specification[3]

To meet requirements the following should be done:

- distributed FID->MDT mapping index (FIDs Location Database - FLD) should be established;
- there should be API containing at least three methods:
  - create index entry;
  - delete index entry;
  - lookup entry by FID.
- there should be locking mechanism sequence in migration via *create* and *delete* API methods;
- each node in cluster should be able to issue create, delete and lookup RPCs to perform FID to MDT resolution and index setup in create and delete time;
- there should be caching mechanism to prevent sending RPCs each time as FID->MDT resolution is needed;
- service/component which performs data migration should re-setup FLD and clients caches to allow them find migrated data in new location.

[2] This is mandatory section. It should contain "what management wants" we do in this work.

[3] This is mandatory section, it should contain "what should we do (and may be why) to meet requirements"

4 Use cases[4]

4.1 API use cases

New functionality may be used the following way:

1. in connect time, server uses *fld_ create()* to make sure that FLD is set up for new sequence which is going to be given to client;
2. client switches to new sequence, uses *fld_ create()* to make sure that FLD is set up for it;
3. client or server use *fld_ lookup()* to find "home MDT" of some FID.

4.2 Unit tests

FLD is component which is used by any operation in Lustre. Any existing test is also test for FLD in this case.

Additional recovery related test. See recovery changes in section 6 for details.

1. Not committed FLD, failed MDT, client should be able to find correct MDT after recovery is finished:
   (a) setup Lustre;
   (b) make client switch sequence and creates file "foo" in it;
   (c) fail MDT, last *fld_ create()* not committed;
   (d) client makes sure that cached FLD entry for new created sequence is invalidated;
   (e) client tries to stat "foo" and its home MDT should be found.

5 Logical specification[5]

To meet new functionality needs in section 3 FLD should work out the following fields.

First of all, importantant to state, that FLD API methods operate on *fid sequence* rather than on whole FID. So, even if they have whole FID passed as one of arguments, *fid sequence* is mostly needed for their work. And other fields may be used for some aside functionality, like sanity checks, debug info, etc.

5.1 FLD API

There is three methods used with FLD. See section 5.2 to see how MDT number is calculated to send FLD RPCs. API methods are the following:

[4] This is mandatory section, it should contain "how to use or how to check new functionality". It is naturally to use it as design for unit or sanity tests.

[5] Mandatory section, moreover, it is very important. It should contain "how to implement new functionality to meet requirements"

5.1.1 fld_create()

Creates new index entry in FLD. Its purpose is to make index entry with passed FID so that this entry may be found later using the FID. This is quite rare operation as it is only performed when client switches to/obtains new sequence. That is possible in connect time and when current sequence is close to be exhausted (say 10000 objects created). As this is rare operation, performance will not suffer from having this one more RPC.

May be used in the following cases:

- in new sequence startup;
- migrator sets up FLD for new data location.

5.1.2 fld_delete()

Deletes index entry to make passed FID impossible to find. In fact deleting index entry may be optional operation. Index does not grow so fast to cause some issues. This is because of quite long sequences.

Delete may be used in the following cases:

- in migration time, migrator deletes old fid->mds mapping and establishes new one to allow to find data in new location;
- server can delete entry from FLD for some sequence if no more files in this sequence are stored on it. However, this should be tracked somehow and seems this is not matter of this HLD.

5.1.3 fld_lookup()

Finds index entry by passed FID (in fact its sequence). Most commonly used method. It may be issued by any node. After first lookup is done, cached value is used for all consequent operations. See section 5.3 for details.

The result of lookup may be some complex structure which contains not only raw MDT number, but rather full addressing or location specification of requested resource (FID).

5.2 FLD structure

All MDTs in cluster store own part of FLD mapping index. Index is distributed across all MDTs using some DHT (Distributed Hash Table). So that, all FLD RPCs (see section 5.1 for details) are sent to MDT server which may be found by this function:

```
idx-mds-num = dht_hash_func(seq-number)
```

In this function, *seq number* is taken from FID assigned to inode in inode init time on client, or seq number obtained from server in client's connect time.

Each MDT maintains htree based index (see Index API for details) that maps FID to MDT which is a home for that FID. In general case, MDT that knows some FID's home MDT is not same as home MDS of the FID being looked up. This is why all methods from FLD API, including *fld_ create()* and *fld_ delete()* should be exported to network and cannot be used only locally from MDT.

5.3 FLD caching

There is need to cache FLD lookup results on clients to avoid doing it on each operation again and again. Reasons are the following:

- to avoid doing bunch of FLD lookup RPCs in big clusters needlessly loading MDT servers and network;
- FLD is not going to change quickly. Once created objects live on same MDT until migration and thus, this is very rare operation;
- to avoid doing RPC in operation time due to the following:
  - this makes operation tied to some MDT server and some operations are asynchronous or do not need server's attention;
  - this makes all operations in Lustre slow as they need to wait for lookup answer first.

We could have lockless cache in the following case:

- sequences are never reused;
- sequences never migrate.

However this is not our case, because of the following:

- sequences may migrate and migrator should cause all clients to invalidate their caches;
- client may switch to new sequence in some cases (exhausting, some smart logic related to placement policies, etc).

5.3.1 Using DLM for managing caches

One of ways to keep cache consistent is to use DLM for that.

FLD locks namespace

For doing that we need separate locking namespace and here we face with need to define this namespace. How DLM is going to manage locks, what is used for resource id, etc. As the solution FLD may use the following approach:

- locking namespace is sequences based. That is sequence number is subject of namespace;
- resource id is built on sole sequence number. This is possible because of properties of sequences:
  - they never reused;
  - they are argument to function of making decision as for what MDT for FLD lookup should be chosen. That is, taking not compatible locks will cause cache invalidation on all clients. See section 5.3.1 for details;

Using FLD locks & cache invalidation

To maintain cache consistency, FLD DLM locks should be taken the following way:

- each node, performing lookup, should check cache first;
- if there is no cached result - perform FLD lookup RPC;
- if there is cached result, node checks first its validness, that is, if there is valid read lock for interesting sequence. If there is no lock - cache is invalidated and lookup RPC should be performed;
- after lookup is finished, server returns read lock for the result;
- each node starting to migrate sequence, takes write lock. This makes sure that:
  - all clients have their caches invalidated, because for using cached results they take read lock which is not compatible with write one;
  - all possible readers (some clients may try to access data being moved) will wait until write lock is released, that is, migration is finished.
- each node, performing delete, should take write lock first on the sequence being deleted. This guarantees that all cached FLD entries on all nodes will be invalidated;
- blocking ast function on all nodes using FLD lookup invalidates local cached lookup result for particular sequence.

Having this schema guarantees that cache is consistent on all nodes using it. However, there are possible issues about atomicity of connecting and setting up cache. See section 6 for details.

5.4 Adding MDT

There is possibility to add MDT in live cluster. And this means that new MDT should start manage its part of distributed index. Here we have the following possible issue:

- adding one more MDT may cause all clients request wrong MDT with FLD API methods RPCs as hash function used for that may give another result taking into account that number of MDTs has changed. Thus, used hash function should take this into account and do not form its yield on number of MDTs. See section 5.5 for details;

5.5 Hash function

Hash function used for calculating MDT for sending RPC to should be based on DHT to be able to work fine in the case of adding new MDT t oalive cluster. It's details are matter of DLD.

5.6 Data migration

In migration time, migrator should do the following:

- make sure that client's caches containing migrated sequence are invalidated. This is done by canceling client's read locks taking write lock on server. Write lock is also needed to protect FLD from reading while it is modifying. This may be done by CMOBD running along with cache MDT. Clients wait on taking read lock on migrating sequence and do nothing until migration is finished;
- modify FLD to remove old routing information and add new one to allow clients to find migrated objects in new location;
- after this is done, clients are able to find data in new location.

6 State management[6]

6.1 State invariants

There are the following state invariants introduced by FLD:

- sequence setup should be atomic. That is, connecting to MDT, getting new sequence from it, setting up FLD entry on MDT for this sequence and setting up local cache on client should be atomic. This means, that client should not perfrom any FLD related operations before this is done as whole;

[6]Mandatory and very important section, it should contain recovery changes, scalability, formats, protocols, state invariants, state sharing, etc.

- in migration time, client's should wait until migration is finished and both data and sequence are migrated;
- cached entries should be removed from memory on lock canceling;

6.2 Scalability & Performance

There are the following changes to scalability and performance:

6.2.1 Scalability

Scalability should not be affected. Clients create objects in separate sequences, so even if they all create 10000 files in parallel that does not cause sequence change to any of them.

In the case of invalidating local lookup caches, clients will not be busy long time due to the following:

- local cached entry is of simple structure, does not need any iterations, just remove it from memory is enough;
- cache invalidation is rare operation. We may use existing DLM infrastructure for finding cached FLD entry from lock.

However, there is theoretical possibility that one hostile client affects other clients in cluster.

This is possible when the client starts quickly switch to new sequences and create objects on them. And if other clients in cluster use objects created by first client, they will have their caches grow. And in case of invalidation, clients theoretically may be busy enough to cause some inconsistencies to applications running on them.

Thus, cache should have a mechanism of managing cache in terms of size. Using Linux built-in mechanism is preferred way for doing that.

6.2.2 Performance

Performance should not be affected. FLD RPCs are not sent so often to hurt it due to the following:

- local caches;
- sequences are quite long to.

6.3 Recovery changes

The following recovery changes are expected:

6.3.1 Failed MDT

In the case when MDT failed, we have the following recovery scenario:

- MDT failed, recovery is started;
- after MDT is up, all clients should resend their *fld_ create()* and *fld_ delete()* RPCs (replay) which did not fit into last committed transaction. Thus, FLD modification RPCs on client should be preserved if not committed yet like that is done about regular Lustre RPCs;
- after clients sent their FLD modification RPCs and set up own caches, cluster continue function;
- until recovery finished and all FLD recovered, no client may send new FLD RPCs to MDT being recovered.

6.3.2 Failed client

In the case of client failure, nothing special is happening. Client loses its FLD cache and after mounting again will have to gather it again. FLD on MDT is consistent.

6.3.3 Client eviction

In the case of client's eviction all locks are getting canceled and thus, client's cache will be invalidated. After client re-connects, it will gather its cache again.

6.4 Disk format changes

FLD introduces disk format changes due to storing htree on each MDT. But this is described in Index API HLD and will not be described here again.

6.5 Wire format changes

In connect time, MDT should send new sequence to clients and to another MDTs connecting to it.

6.6 Protocol changes

FLD introduces three new RPCs due to need to have API methods described in section 5.1.

6.7 API changes

FLD adds three new API methods. They may be part of *md_* API or separate into stand alone *fld_* API like it is done about separating *obd_* methods and *md_* ones.

6.8 RPCs order changes

In fact no RPC order changes expected. The only thing which should be mentioned is that, all RPCs should be sent after sequence is set up, *fld_create()* RPC is sent and completed and cache is set up.

7 Alternatives[7]

The following alternatives are possible in this area:

- we may do not use cache consistency managing in cases when no migration is expected and cache may function as lockless one;
- hash function may be different;
- mapping index structure may be organized another way;
- FLD could store mapping for range of sequences rather than for one. That would make its size smaller and possibly improbe performance.

8 Focus for inspections[8]

Main focus in inspection should be given to these aspects:

- cache management;
- mapping index structure, API methods.

[7] Not mandatory section. May contain alternative points of view on some aspects of HLD.
[8] Not mandatory section. It may contain references to some important parts of HLD. Some complicated algorithms, disputable or opaque solutions, etc.