# FID HLD

Yury Umanets

20th January 2006

## Contents

# 1 Introduction[1]

In this work we're introducing FIDs to make cleanups of object identification protocol possible for regular Lustre activities and for write back caches and proxy nodes.

This is maintenance HLD for FIDs. It describes only basic things about FIDs, like management, recovery, etc. It does not cover gaps handling simplifications, FLD (FIDs lookup database), etc. which are matter of separated HLDs. This HLD uses some definitions, check them before reading.

## 1.1 Definitions

Here are some definitions used in this HLD:

***fid number*** - part of fully specified FID, contains object id within its sequence;

***seq number*** - part of fully specified FID, contains sequence in which object was created;

**object version** - part of fully specified FID, contains object version number. It is matter of separate HLD;

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

***FID*** *or* ***fid*** - fully specified object identification structure, *FID = (f-sequence, f-number, f-version)*;

***home MDT*** - MDT, that contains inode of some object or master inode for split directories, is called *home MDT* for that object;

***name MDT*** - MDT, that contains a given name within a given directory. In the non-CMD configurations content of directory are always stored on a single server. In the CMD configuration parts of split directory are distributed across multiple nodes according to a "split algorithm".

***cross-ref*** - in CMD configuration, names and their inodes may be stored on different MDTs. Such a case is called cross-ref;

**migrator** - node, which performs data and FIDs migration. This may be cache/proxy node in cache flush time or any client node driven by special utility used by system administrator;

**slave** - when talking about WB caches, means node which flushes/moves data to new location;

**master** - when talking about WB caches, means node which is new location for data after migration or cache flush;

**meta-sequence** - sequence of sequences, range of sequences or grant of sequences given by server to client for allocation fids in all sequences available from the range.

# 2 Requirements[2]

FID is an universal object identifier in Lustre, stable across cluster and during whole cluster life-time.

## 2.1 Fid structure

Fid *should* contain the following components:

- object identity. This should be one of few fields making possible to uniquely identify objects in Lustre;
- object version. This component should provide support for storing and accessing different versions of objects in Lustre.

Fid *should not* contain the following components:

- object store related information, that is, inode number and inode generation;
- object location information like MDT number.

[2]This is mandatory section. It should contain “what management wants” us to do in this work.

## 2.2 FID using in clients

There are few aspects of using fids on clients. They are the following:

- client should not have inode number collisions for inodes living on different MDTs in CMD configurations. So, it should be built on some cluster unique object identifier. Look at section 3.2 for details;
- client inode number does not depend on version component from FID;
- client uses FIDs for object identification across whole cluster.

## 2.3 Fids using in DLM

- fids should be used for identifying Lustre objects for DLM;
- DLM resource id should be built on FIDs;
- DLM resource id built on fids is unique across its namespace;
- fids can be used for sanity checks in DLM related stuff. This mostly means that FID<->resource id conversion should be simple to extract each of them having only one of them;

## 2.4 Fids management

There are few requirements related to fids management:

- fids should be issued by clients;
- each MDC connect to MDT should start new fid sequence;
- each OSC connect to OST should start new fid sequence;
- each sequence exhausting (accordingly to chosen limit) starts new fid sequence;
- each client re-connect (recovered client) continues to use the same sequence;
- each client has sequences for all MDT and OST servers;
- all objects created within same sequence should live on same MDT;
- FIDs are unique, they may be used for sanity checks, asserts, etc.

## 2.5 Consistency & compatibility

There are few general requirements about fids related to consistency. They are the following:

- an object store may create at most one object with designated FID;
- objects can be efficiently found by FID;
- servers should provide a mechanism to store objects with FIDs. They will be used for reconstruction, locking, etc;
- FIDs should be backward compatible with at least one 1.6.x release which does not contain fids;

## 2.6 Fids location & migration

There are few migration and location requirements:

- FIDs (stored in EA or another structure in store) and designated objects should be able to migrate easily, that is, without any additional conversion operations, etc;
- migration unit is whole sequence, rather than individual objects/fids in it;
- for each operation against some inode, client, using FID components, should be able to find *home MDT* to send RPCs. See section 1.1 for detailed definition of "home MDT";
- proxies, caches, are responsible for finding FIDs after migration.

## 2.7 Benefits

Achieving requirements above, we will have the following benefits:

- enable cache/proxy nodes;
- unconnected operations, asynchronous creates, unconnected caches/proxies;
- recovery simplicity and improvements (simplifying recovery in MDT and gaps handling);
- add support for object versions;
- get rid of iopen patch (servers provide lookup by FID);
- metadata protocol cleanups;
- object identification cleanups;
- performance improvements;
- client independent on server side object store;
- get rid of exporting object store inodes to user space on clients.

# 3 Functional specification[3]

## 3.1 Fid structure

To meet requirements in section 2.1and provide planned features and benefits, FID contains minimal set of fields. One of important things here is the notion that FID should not contain any address information like MDT num. This is need to migrate easily and avoid converting FIDs in MDT EAs, in cache/proxy nodes, etc. This is why it has such a minimal set of fields, which are the following:

- **fid number**. This is in fact object identifier. It is unique in its sequence;
- **seq number**. This is number of sequence. Fid numbers are unique in one sequence;
- **fid version**. This is object version and needed to support objects with different versions;
- pair, *fid number and seq number* form unique file id (or FID) in this look: FID = *(f-sequence, f-fid). It* may be used for DLM, sanity checks, etc

FID may be a part of more complicated structures living in memory.

## 3.2 FIDs using in client

Client should have the following done about fids:

- llite inode numbers should be built on FID rather than on store cookies from MDT. This helps to prevent possible inode collisions in CMD configurations, in compare to former solution, when client's inode number and generation were taken from MDTs and thus, might happen to be the same for different inodes on the same client. See section 5.5 for details;
- at inode initialization time, each client should link corresponding FID to the inode. This FID will be used later for communicating with servers and perform operations against object associated with FID. Client should not use anything but FID to talk to servers for object related operations. Client should not send any store cookies, client has no idea about store at all;
- also, at inode initialization time, client saves object striping info which possibly contains FIDs for each stripe;
- client inode numbers do not depend on version component from FID;
- all client functions should be converted to use FIDs, as it is only the universal object identifier in Lustre.

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

## 3.3 Fid using for DLM

FID uniqueness makes it naturally suitable to use in locking stuff. It may be used for the following:

- FIDs are used for building resource id for DLM locks. This makes resource id unique what completely implements requirements in this area;
- FIDs are used for sanity checks. For instance, when client requests lock from MDT and server sends reply, client may check if requested lock coherent with FID of object lock should be taken on.

## 3.4 Fids management

Fids are managed the following way:

- FIDs should be issued by clients using corresponding sequence. This means, that for creating new object, client does not have to talk to server. This allows to implement unconnected operations, create bunch of objects in one shot, etc. This also enables cache/proxy nodes to create objects independently of master server;
- each connect from node which may issue create RPCs starts new fid sequence. See section 5.1 for details. As each connect starts new sequence, this allows to not save last fid on MDT (former approach) for each create operation. This makes recovery stuff on MDT simpler and also gives performance benefits as we do not do one more I/O in create time;
- each sequence has limited number of objects which may be created in it. That is, when sequence is exhausted, new one should be started;
- each re-connect (recovered client) continues to use the same sequence. See recovery stuff in section 6.7 for details;
- sequences are cluster-wide. This guarantees that FIDs are unique across cluster;

## 3.5 Consistency

MDD and OSD servers should constantly take care, that it may create at most one object with designated FID. At inode create time, they should save object FIDs with inode. See section 5.3 for more details.

MDD and OSD use lookup mechanism based on Index API, to find inodes by means of using associated FID, obtained from client.

### 3.6 Fids location & migration

Fid does not contain location and addressing information. This is needed to migrate easily. However, this makes also some problems about objects location. They are the following:

- both, client and server should use FIDs location database (FLD) to find object *home MDT* to perform operation against it;
- migrator (cache, proxy, etc.) should take personal responsibility that migrated FIDs will be found in new location.

Read FLD HLD for details of FIDs lookup, clients cache management and migration details.

## 4 Use cases[4]

This section describes two types of use cases. First one is related to unit tests to check new functionality. And second - to using fids in Lustre itself, regular lustre operations, some possible complicated cases which should be described especially.

### 4.1 Unit tests

The following use cases may form basis for unit tests:

1. Connect changes sequence number in MDT:
   (a) setup Lustre;
   (b) get last sequence number from MDT, using lproc;
   (c) disconnect client;
   (d) connect client again;
   (e) get new value of last sequence number from MDT, using lproc;
   (f) check if new sequence number larger than old one.
2. Re-connect does not change sequence number:
   (a) setup Lustre;
   (b) get last sequence number from MDT, using lproc;
   (c) fail MDT and wait for recovery to finish;
   (d) get new value of last sequence number from MDT, using lproc;
   (e) check if new sequence number the same as old one.

[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.

3. Client maintains fids:
   (a) setup Lustre;
   (b) disable client's sequence switch;
   (c) get last fids for all MDSes from client, using lproc;
   (d) create object;
   (e) find what MDS it lives at;
   (f) get new value last fid for that MDS from client, using lproc;
   (g) check if new fid value is larger then old one for correct MDS.
4. Client does not reuse fids:
   (a) setup Lustre;
   (b) create object;
   (c) get last fid from client, using lproc;
   (d) delete object;
   (e) get new value of last fid from client, using lproc;
   (f) check if new value did not change after object is deleted.
5. Client forms its inode numbers using fids:
   (a) setup Lustre;
   (b) create few objects;
   (c) get their inode numbers;
   (d) check if they correlate with chosen fid->inum mapping algorithm on client.
6. Client inode may be found on MDT:
   (a) setup Lustre;
   (b) create object;
   (c) change its ownership;
   (d) umount client;
   (e) mount client again;
   (f) get object attributes;
   (g) check if attributes after mount the same as before umount.
7. Client's inode fid stored in inode info and MDT' one stored in EA the same:
   (a) setup Lustre;

(b) create object;

(c) get object fid from client's inode info, using lproc;

(d) get object fid from MDT inode EA, using lproc;

(e) check if client's and MDT' fids the same.

8. Sequences are cluster-wide:

(a) setup Lustre using few clients;

(b) get sequence numbers from all MDTs;

(c) check that all MDTs know the same set of sequences. There may be different checks. One of them would be ask *sequence controller* (see below) node as for how many sequences allocated after all clients and servers connected cluster.

## 4.2 Using FIDs

Here are some examples of using FIDs for regular Lustre operations. Some definitions used in this section, see section 1.1 for details.

1. Regular create

(a) using split policy, client obtains MDT number where name should live;

(b) in this case *name MDT* and *home MDT* are the same;

(c) client generates new FID from sequence obtained from object *home MDT,* generates new FID from sequence obtained from OST and sends RPC with both FIDs supplied to the MDT;

(d) *home MDT* creates object, saves its MD FID into EA and sends create RPC to OST (this is going to change in CROW);

(e) OST creates object, saves both FIDs (MD and DT ones) into EA and returns status to MDT. FIDs in EA are used later for reconstruction;

(f) *home MDT* returns status to client;

(g) client saves both FIDs in new created inode (inode info).

2. Cross-ref create

(a) using placement policy, client obtains MDT number where object should live (*home MDT*);

(b) using split policy, client obtains MDT number where name should live (*name MDT*);

(c) *home MDT* and *name MDT* are different, client detects cross-ref create;

(d) client generates new FID from sequence obtained from object *home MDT* (not same as *name MDT* in this case) and sends RPC to *name MDT*;

(e) *name MDT* creates name and sends create RPC to object *home MDT* using the FID generated by client from *home MDT* sequence;

(f) *home MDT* creates object, saves MD FID into EA, sends create RPC to OST (like in point 1) and returns status to *name MDT*;

(g) *name MDT* returns operation status to client;

(h) client saves both FIDs in new created inode (inode info).

3. Regular lookup case

(a) using placement policy, client find *name MDT* and sends lookup RPC to it;

(b) *name MDT* finds name and if this is not cross-ref case, it returns objects attributes to client; Along with object attributes, object FID is also returned to client;

(c) client saves FID from MDT in inode (inode info).

4. Cross-ref lookup case

(a) using split policy, client finds *name MDT* and sends lookup RPC to it;

(b) *name MDT* finds name and detects cross-ref case;

(c) *name MDT* extracts FID from dentry, finds FID's *home MDT* (using FLD) and sends RPC to object *home MDT;*

(d) *home MDT* returns objects attributes to *name MDT*;

(e) *name MDT* returns attributes to client;

(f) client saves attributes from *name MDT* in inode.

# 5 Logic specification[5]

To accomplish requirements, both, client and servers have to be changed a lot. Note, this chapter only describes regular activities, for recovery details see section 6.7.

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

## 5.1 Seq number management

General rules as for *seq number* management are the following:

- there is such a cluster-wide 96-bit digital value - *sequence*. It is compound value and consists of the following entities:
  - 64-bit part is used for storing ranges which server allocates to clients. Thus, server may allocate $2^{64}$of sequence ranges to clients;
  - 32-bit part stores number of sequence number in range. Thus, each range that may be given to client may be $2^{32}$. So that, client may perform $2^{32}$ connections to server before it needs to ask for new range.
- such a compound meta-sequence is stored on MDT on store and used in recovery and in allocation time;
- each node, which may request creating, should have separate sequence for that. Currently only clients may have sequences. Though MDTs request object creating on OST (CROW will change that), they have no sequences for OSTs. See section 1 for details of creating objects on OST and why MDT has no sequences;
- in connect time, client should have range of allowed for allocation sequences; Read section 5.1.1 for details;
- seq number for using (generate new FIDs in it) is incremented by one each time as new sequence is needed (new connection to server or sequence switch);
- *seq number* may be incremented in the case of exhausting (sequence switch). That is, when fid number in current sequence is close to reach chosen limit;
- in the case of sequence switch client uses new sequence from given range and does not need to ask server for that;
- sequences are not re-useable. Also having separate sequences for each MDT and OST, guarantees, that objects that belong to the same sequence live on the same server rather than scattered across all server nodes;
- each client should have two arrays with *active sequences* (currently used ones), which contain sequences from one cluster-wide set. They are the following:
  - array with sequences for all MDTs to generate FIDs for metadata objects;
  - array with sequences for all OSTs to generate FIDs for data objects.
- client performs operations against some server (MDT or OST) using corresponding sequence;

- that is possible for client to perform operations on different sequences in multi-threaded fashion, if operations do not need to synchronize;
- each sequence should not contain more than defined number of objects. This number will be chosen in init time and definitely will depend on cluster size and other parameters;
- each recovered connection continues to use same sequence number. This is needed because all replies should run on the same sequence as original RPCs. See recovery details in section 6.7.

### 5.1.1 Seq number synchronizing

Sticking with approach, that each connection has separate *seq number* (described in section 5.1), raises the question of synchronizing sequences across all involved nodes (sequences management).

In few words, issue looks like the following. If some connection in cluster is performed between two nodes, only these nodes know this fact and may bump local presentations of cluster-wide sequence. Whereas, all other involved nodes also should know it, because they may have connection in next moment or in parallel.

There are few possible solutions of this issue:

1. request sequences with special RPC from some *sequence controller* node. See section 5.1.2 for details;
2. use *meta-sequence* (sequence of sequences) approach, that is, each client has grant from server of allowed for use sequences. See section 5.1 for details of sequence management. Also see section 5.1.3 for details of meta-sequence approach;
3. combine 1 and 2 approaches using good bits of both of them.

### 5.1.2 Requesting seq number with special RPC

This solution implies the following being done:

- first MDT is called *sequence controller*, because it is connected from all clients and all OST servers. Its purpose is to always know correct sequence and tell it to another nodes in cluster;
- only *sequence controller* node (first MDT) may bump sequences and all other nodes just save it locally for own purposes;
- each time as a node wants to switch sequence (connect, limit is reached), it gets correct *seq number* from *sequence controller* node;
- in principle, each MDT node may become *sequence controller* in process of work if needed, because all they have up-to-date *sequence counter* and question is only in that, which one should bump it.

This approach has the folowing advantages:

- simplicity of implementation, no need to maintain complicated protocol of synchronizing sequences;

However there are also following downsides:

- *sequence controller* may be quite busy with sequences allocation requests in connect time in big clusters. Also it is becoming the bottleneck;
- in the case sequence controller fails, nodes which need new sequence get blocked until sequence controller up and runs again;
- no possibility to have parallel connections, all nodes synchronized via sequence controller.

### 5.1.3 Meta-sequence approach

There is alternative way to manage sequences. It is the following:

- in connect time, each client, obtains from each server, special grant to use some range of sequences, called meta-sequence;
- sequence ranges do not cross, so that, each client has own pool;
- client starts use first sequence from range right after connection;
- in the case of switching the sequence, client takes new one from the grant and does not have to ask server.

This approach has the following additional advantages:

- clients do not ask server each time new sequence is needed;
- better scalability;
- no need in special RPC;

Downsides of approach:

- sequence range may be exhausted, it should be chosen with special care and may depend on cluster size or configuration;
- servers need special algorithm of range allocations. However it may be simple one and based on MDT number or something like that, so that servers do not need to synchronize ranges;
- basing range allocation on MDT number may not be appropriate due to possibility to add new MDTs into alive cluster.

### 5.1.4 Combined approach

Apparently if we want to have meta-sequence approach, it should have also some server which controls allocation of meta-sequences. In this is essence of combined approach:

- we have one MDT which is *sequence controller*;
- sequence controller allocates meta-sequences to all clients. This is needed to avoid ranges crossing, etc., as each client should have range which will not cross with ranges given to others;
- in MDT-MDT connect time, they exchange meta-sequence with controller node and store it locally.

Using such approach we avoid many downsides of both. For instance, we do not use special RPC for requesting new sequence.

## 5.2 Fid number management

- each client maintains 16-bit value - *fid number*, for each sequence it has. Fid number for new sequence starts from predefined value - for instance 1, so that 0 is used for sanity checks;
- client increments *fid number* by one (in its sequence) each time as new object is created.
- fid number is not reusable, that is, when some object is deleted, corresponding *fid number* in its sequence does not change;
- each sequence has limit as for how many objects may be created in it. Sequence is supposed to switch to new one if limit is reached;
- *fid number* is not saved to persistent storage, it starts from 1 each time new sequence is started.

## 5.3 FID management

- each client uses own array of sequences to allocate new FIDs for objects on different MDTs and OSTs;
- in client, for new created object, fully specified FID *(f-sequence, f-number, f-version)* for MDT object presentation, is stored in client inode info;
- in client, for new created object, array of fully specified FIDs (one FID per stripe) for OST object presentation, is stored in client inode info;
- both, MD FID and array of OST FIDs in client inode used for the following:

  - communicating with MDT or OST, requesting getattr, etc;
  - issuing DLM locks;
  - sanity checks.

- on MDT, fully specified FID *(f-sequence, f-number, f-version)* for new created inode is stored in inode EA and used later for the following:
  - reconstruction;
  - getting FID by MDT inode for DLM purposes (issue DLM lock);
  - sanity checks.

- on OST server, in object create time, both, OST and MDT FIDs should be stored into inode EA to be used later for the following purposes:
  - reconstruction;
  - getting FID from inode for DLM purposes;
  - sanity checks.

- each node should be able to find correct server handling particular FID, to send operation RPCs to it;

- each MDT and OST provide a way to resolve FIDs into local inodes. This should use Index API with fid->store cookie mapping;

- there should be a way to iterate over all fids or over fids from some sequence. See Index API HLD for details.

## 5.4 Using FIDs for DLM purposes

All nodes in cluster should use FIDs to make resource id to use it for DLM purposes (issuing locks, matching, etc.). This may be done by using such a rule: *resource_id = (f-sequence, f-number)*. This rule defines locks namespace as space of objects which are visible for DLM as resources.

Using above rule guarantees that resource id is unique across whole cluster.

## 5.5 Client inode numbers

To meet requirements in section 2.2, client inode numbers should be built on FID components linked to inode. This is matter of mapping algorithm used on client. It should has the following properties:

- inode numbers built using some mapping algorithm. It should map *fid number* and *fid sequence* into client inode number. It should have the following properties:

  - for any two different pairs (*fid number* and *seq number*) there should be different inode numbers. That is possibility of inode number collisions should be very low;
  - algorithm should take into account quite big range of possible seq numbers (many clients, many connects);
  - algorithm should take into account, that *seq number* is changed quickly. It should produce unique numbers;
  - using zero fid number, algorithm should yield zero inode number, to use it in sanity checks;
  - clients inode numbers should be unique on all client nodes.

- inode number does not depend on version component from inode FID;
- inode generation is chosen by client's VFS in inode init time and may be left as is.

## 5.6 Using FIDs with WB cache, proxies

There are few aspects of using FIDs for WB caching and proxy nodes. They are the following:

- sequence allocation changes to regular schema;
- migration.

### 5.6.1 Sequence allocation changes

There are the following changes:

- WB cache obtains its sequence from master node in connect time. This connection to master may be performed:
  - in WB cache setup time;
  - or in cache flush time.
- thus, WB cache function as a client (allocates FIDs, etc.) for *master* and as regular MDT or OST for client node (maintain sequences, etc);
- migrator is part of cluster, this means the following:
  - it has sequences from one cluster wide set;
  - after migration, when data is moved into new location, they should be found by original client in new location. That is, cache node may remove the data from cache. This is especially important for non persistent caches, which hold their data in memory (for instance using tmpfs).

### 5.6.2 Migration

In migration time, *migrator* should perform the following actions:

- move data to new location (*master* node). All creates should be performed using cache node sequence (obtained from *master*) rather than client's one, so that:
  - it is always known that data on some *master* are created as result of cache flush from some WB node;
  - client could find data in new location later.
- re-setup FLD so that data could be found in new location. Thus, when client requests data which already removed from cache, request should be forwarded to the node data live after migration;
- make sure that client's FLD caches are flushed for moved sequences. See FLD HLD for more details.

# 6 State management[6]

## 6.1 State invariants

The following state invariants should be kept true:

- object, created on server and known to client (inode with assigned FID), can be found by client's FID;
- last *seq number* or meta-sequence saved to "SEQUENCE" file on server, should be coherent with existing objects and their fids. That is, there should not be objects on server with seq number larger that what is saved in "SEQUENCE".

## 6.2 Protocol changes

The following protocol changes will take place:

- as client always creates inode on its *home MDT* (in accordance with *home MDS* definition) and knows full FID before sending RPC, server does not have to return FID to client on create operation. However, operations like getattr still need FID returned to client, as they may be requested by clients from different sequences including those, which do not have FID in their local inode yet;

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

- server does not have to return store cookie to client anymore;
- client may do not wait for MDT create inode completion, as client already knows full FID of the object;
- each node that joins cluster sends *seq number* request RPC to sequence controller node (first MDT);
- using sequence grants (meta-sequence approach) implies changes to connect protocol;
- FLD implies need to issue some RPCs. Read FLD HLD for details.

## 6.3 API

API is changed in the following aspects:

- all methods using former objects presentation need changes to use fids instead;
- some functions need their names changed to be coherent with functionality.

## 6.4 Disk format

Disk format is changed in the following aspects:

- each MDT object creates additional EA which stores object's fid component. Back store FS should be formatted appropriately (EA in inode for ext3);
- each MDT saves object stripe info which contains FIDs instaed of of former objids;
- each OST saves own FID and FID from MDT into inode EA on object create;
- there will be changes related to FLD. See FLD for details;
- both MDT and OST has new file "SEQUENCE" which holds last known sequence.

## 6.5 Wire format

Wire format is changed in the following aspects:

- all RPCs should have fid instead of former structures;
- many MD related structures changed their size.

## 6.6 Scalability and Performance

The following changes to performance may take place:

- performance may be increased due to allocating fids on clients, thus clients may perform asynchronous creates, create bunch objects into one shot, etc;
- OSD on both MDT and OST servers should update fid->inode map on each create. This is additional work and may cause some slowdown;
- FLD will have some overhead. See FLD HLD for details;
- MDT and OST does not do additional I/O on each create, that is, it does not save last fid into LAST_FID file what should also have additional performance boost.

The following scalability changes may take place:

- clients do not depend on server object store (inode number, generation and their allocation policies). This means that server object store may be changed (another FS, etc.) with smaller impact to client.

## 6.7 Recovery

Recovery has many changes, both on clients and servers.

### 6.7.1 Seq number allocation

Though we said that seq number is not changed in recovery, here is some clarification on this matter. The following rules used in recovery time for maintain seq numbers:

- *seq number* **does not** change in case of server failure, because of the following:
  - alive clients should reply their requests (especially create ones) in the same sequence. This is important, because there possibly another clients which have inodes in memory with sequence number used before failure. And say re-creating inodes with new sequence will cause a lot of problems to those clients;
  - alive clients have correct fid number for their sequences and may continue using them.
- *seq number* **does** change in case of client failure and in reconnect time client should obtain new sequences from all servers. That is because of the following:
  - failed client loses fid number counter (which is managed in client's memory) for all sequences and cannot continue using them;
  - failed client does not have anything to replay, and does not need to keep its old sequences.

### 6.7.2 MDT & OST (regular sequences allocation)

The following changes to MDT and OST recovery should take place:

- each time as new client connects to MDT or OST (server), they allocate new sequence, return it to the client and save it to "SEQUENCE" file with consequent commit. Server is supposed to create "SEQUENCE" file if it does not exist yet;
- in recovery time, server reads "SEQUENCE" file. It uses it for reporting sequence to clients restored connection to MDT in recovery time;
- server re-creates objects in recovery time and takes care that they have correct FID stored in EA;
- server does not care of *fid number* reconstruction and thus, does not have to store it. Client is replaying requests in recovery time, sends them with already initialized FID and all server has to do is to recreate object and save FID into EA. After recovery is finished, client has its FIDs synchronized with server.

### 6.7.3 MDT & OST (meta-sequence based allocation)

- in connect time, each server gives a sequence grant (meta-sequence) to each client. Sequences from that grant may be used without servers confirmation. In this time, server should save allocated meta-sequence on local store ("SEQUENCE" file);
- in recovery time, server uses grant from store to return it to client after reconnect, so that client continue to use it;
- server re-creates objects and takes care that correct FIDs are saved to object's EA.

### 6.7.4 Client

The following changes to clients should take place:

- on each create completion, client does not need to save FID from server to create request, as it is already known before sending RPC to server;
- in client recovery time, current *seq numbers* do not change and all request replies are performed in their former sequences;
- in case of eviction, current *seq numbers* do not change as well.

# 7 Alternatives[7]

There are the following alternatives possible:

- client inode numbers allocation algorithm in section ?? may state that inode numbers should start from same value on all clients, say 1 or they may be different, based on sequence for instance;
- seq number synchronization algorithm in section 5.1.1.

# 8 Focus for inspections[8]

The following should be inspected carefully:

- seq number synchronization algorithm and its connection to FLD;
- WB caches and proxies.

[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.