Sequence Management DLD

Yury Umanets

25th May, 2006

Contents

1 Introduction

This DLD only describes details of FIDs sequence management. Main focus is that, sequences are cluster wide and unique and thus, need to be managed in a special way. This DLD is addition to FIDs DLD. Read it first for getting idea of what FID is, why is it needed and so on.

There are some definitions:

sequence - sequence of FIDs, may contain limited number of FIDs, for instance 10 000. Clients use sequences to allocate FIDs in it.

meta-sequence - range of sequences, may contain limited number of sequences. Clients use meta-sequence to allocate new sequences without asking server;

super-sequence - range of sequences from what meta-sequences are allocated, may contain limited number of sequences. Servers use super-sequences to allocate new meta-sequences to clients without asking sequence controller;

sequence-controller - server which allocates super-sequences to other servers.

2 Functional Specification

2.1 Sequence invariants

Sequences introduce the following invariants:

- sequences are cluster wide space, all nodes should know sequences belonging to the same cluster-wide set. Though clients only work/access theose sequences belonging to meta-sequence obtained from sequence manager and do not know those sequences obtained by other clients, all these sequences all clients work with belong to same sequences space. This means that different clients work sequences ranges which do not overlap;
- sequence number is cluster unique, this is base for unique FIDs and all the FIDs stuff is built on notion that they are unique;
- sequence numbers are not reusable;
- each node in cluster, which may issue FIDs, should obtain own sequence in connect time. Clients issue FIDs for creating metadata objects on MDT and MDT may issue FIDs to create data objects on OST;
- sequence is always assigned to some server (home server of this sequence), and client obtains sequence during connect to this server and later uses it to generate FIDs for objects stored on this server, until sequence is exhausted;
- it is preferable to minimize sequence related traffic and minimize number of RPCs needed for management sequences.

2.2 Sequence management

To support all invariants above, the following sequence management schema is proposed:

- each client should be given meta-sequence for each server it connects to. Thus, client is allowed to switch to new sequence without asking MDT. When meta-sequence is exhausted, client requests new one from corresponding MDT. This is all clients know and they do not need to know that super-sequences, sequence-controller exist at all;
- there is one MDT which is called sequence controller. Its purpose is to manage sequences space and allocate super-sequences to other MDTs. In same time, sequence controller is also regular MDT which should allocate meta-sequences to clients. Thus, it should request itself to allocate super-sequence for itself just like for all other MDTs;
- first MDT is considered to be a sequence controller, because it starts first and all others may connect to it;
- after all MDTs have got super-sequences in connect time, they are ready for allocating meta-sequences to clients using obtained super-sequence;
- sequence controller, after allocating new super-sequence (that is after changing sequences space range), should save modified space range to persistent store for recovery. See section 5.3 for details;
- all other MDTs should save their super-sequences to backing store just after getting them from sequence controller MDT and/or allocating new meta-sequence. This is also needed for recovery purposes;
- when an MDT experiences super-sequence exhausting, it is supposed to ask new one from sequence controller.

2.3 Sequence manager API

The following API functions are used to work with sequence manager.

2.3.1 Server Sequence Manager

Server side sequence manager has the following methods:

seq_server_init() - initialize sequence manager on server side.

seq_server_fini() - finalize sequence manager on server side.

seq_server_controller() - assign client of sequence controller. It is used to talk to sequence controller and ask it to allocate new super-sequences for regular MDTs.

2.3.2 Client Sequence Manager

Client side sequence manager has the following methods:

seq_client_init() - initialize sequence manager on client side.

seq_client_fini() - finalize sequence manager on client side.

seq_client_alloc_super() - allocate new super-sequence, that is, ask sequence controller to allocate super-sequence. This function is used by regular MDTs to talk to sequence-controller when new super-sequence is needed.

seq_client_alloc_meta() - allocate new meta-sequence. That is, ask regular MDT to allocate meta-sequence for the client. This function is used by client nodes only to talk to regular MDTs when new meta-sequence is needed.

seq_client_alloc_seq() - allocate new sequence number. This function is only used by clients. If client's meta-sequence is already allocated and has free sequences, new sequence is taken from it. If meta-sequence is exhausted, server is asked to allocate new one using *seq_client_alloc_meta()* function, and then client uses it to allocate new sequence number.

seq_client_alloc_fid() - allocate new FIDs using current sequence. If sequence is exhausted, new one is allocate using *seq_client_alloc_seq()* function.

Using this API both client and server may work with sequence manager.

2.4 Using Sequence Manager

There are two parts of using sequence manager: server side and client side.

2.4.1 Server side

Server is on duty to start sequence management service, so that clients may connect to server and send sequence requests. Server starts sequence-manager using *seq_server_init()* method. One more aspect of using sequence manager on server is that, server also has client which should talk to sequence-controller node. The method *seq_server_controller()* assigns client sequence manager to server in initialization time and thus, allows server talk to controller. See section 2.4.2 for details.

Though there are two types of servers (controller node and regular one), server side sequence manager does not distinguish them by special flags and only considers that sequence controller is that MDT which is started first. Thus, all other MDTs know that first MDT is controller and connect to it in startup time.

Thus, sequence controller in same time also regular MDT and connects to itself to request super-sequences using RPCs just like other MDTs. In this time, very important to remember about possible deadlocks on *seq->seq_sem* semaphore, because server takes sem while doing any allocations. This means, that server should release sem before talking to other servers, because it never knows what server does it request and it may be same server as it is.

2.4.2 Client side

On client side (client nodes or MDT), client sequence manager should be initialized to send sequence RPCs to server sequence manager and manage meta-sequence obtained from corresponding MDT. This is done using *seq_client_init()* method.

Few methods are used to obtain sequence number and/or allocate new FID using last sequence. They in fact consume such a resource as meta-sequence space and may cause the situation when client should request new one.

3 Use Cases

The following use cases can be checked:

1. Controller allocates super-sequence for generic MDT
   (a) an MDT, on startup connects to sequence controller and asks it to allocate new super-sequence;
   (b) controller allocates new super-sequence and sends it to MDT;
   (c) MDT uses own super-sequence for allocating meta-sequences to its clients.
2. Generic MDT exhausted its super-sequence
   (a) when MDT exhausts its super-sequence, it may ask to allocate new one using client sequence manager and request to sequence controller node;
   (b) sequence controller gets the RPC and allocates new super-sequence;
   (c) both servers save changed ranges to backing store.
3. Client gets meta-sequence from all servers
   (a) in connect time, client using client interface of sequence management module requests meta-sequence from all servers it works with;
   (b) when meta-sequence is exhausted, client ask for new one.
4. Controller recovery

(a) each time as controller allocates new super-sequence, it saves it to backing store;

(b) in recovery time, super-sequence is read from backing store and used for allocating new super-sequences.

5. Generic MDT recovery

   (a) each time as MDT asks controller to allocate new super-sequence or when MDT allocates meta-sequence for a client, it saves it to backing store;

   (b) in recovery time, super-sequence is read from backing store and used for allocating new meta-sequences.

6. Controller requests itself to allocate super-sequence

   (a) in startup time, first started MDT is considered to be the sequence controller;

   (b) in same time it is MDT which should allocate meta-sequences to clients and thus, it needs super-sequence to be allocate by controller;

   (c) thus, sequence controller requests itself to allocate super-sequence in startup time;

   (d) sequence controller allocate new super-sequence to itself using RPC mechanism and does not fail and does not hang up due to some deadlock.

4 Logic Specification

4.1 Abstract

This is struct which describes the range of sequences.It is suitable for holding both super-sequence and meta-sequence.

```
struct lu_range {
        __u64 lr_start;
        __u64 lr_width;
};
```

lr_start - start of range which may be allocated by server.

lr_width - size of range which may be allocated.

4.1.1 Client sequence manager

Client side sequence manager structure looks like the following:

```
struct lu_client_seq {
        struct obd_export      *seq_exp;
        struct semaphore        seq_sem;
        struct lu_range         seq_range;
        struct lu_fid           seq_fid;
};
```

seq_exp - export to the MDT, client uses it to talk to MDT and ask it for new meta-sequence or new super-sequence if client runs on server.

seq_sem - semaphore used for serializing all sequence and FID allocations.

seq_range - last obtained range: meta-sequence on client nodes and super-sequence on servers. This used to allocate new sequences or new meta-sequences.

seq_fid - last used FID in current sequence. This is used to allocate new FIDs.

4.1.2 Server sequence manager

This is the structure of server sequence manager.

```
struct lu_server_seq {
        struct lu_range         seq_space;
        struct lu_range         seq_super;
        struct dt_device       *seq_dev;
        struct proc_dir_entry  *seq_proc_entry;
        struct ptlrpc_service  *seq_service;
        struct lu_client_seq   *seq_cli;
        struct semaphore        seq_sem;
};
```

seq_space - sequences space from which super-sequences are allocated on sequence controller node. On regular MDTs, this field is initialized by LUSTRE_SEQ_SPACE_RANGE (see below).

seq_super - super-sequence range from which meta-sequences are allocated.

seq_dev - device to store sequence manager state (on controller *seq->seq_space* and *seq->seq_super* and on regular MDT only *seq->seq_super*).

seq_proc_entry - manager proc entry to access various tunables from user-space.

seq_service - server side service accepting clients requests.

seq_cli - client sequence-manager to talk to sequence controller.

seq_sem - semaphore for serializing all allocations and writes to backing store. It also used to make allocation and write to back store the atomic operation.

4.2 Server allocation stuff

Server has handlers for two allocation requests. They are the following:

- *seq_server_alloc_super()* - allocate new super-sequence to MDTs;
- *seq_server_alloc_meta()* - allocate new meta-sequence to clients;

The following is whole sequences space range. It is initial value for ->seq_space and later the field for allocating new super-sequences. See below for details.

```
const struct lu_range LUSTRE_SEQ_SPACE_RANGE = {
        (0x400),
        ((__u64)~0ULL)
};
EXPORT_SYMBOL(LUSTRE_SEQ_SPACE_RANGE);

const struct lu_range LUSTRE_SEQ_ZERO_RANGE = {
        0,
        0
};
EXPORT_SYMBOL(LUSTRE_SEQ_ZERO_RANGE);

static int
seq_server_alloc_super(struct lu_server_seq *seq,
                       struct lu_range *range)
{
        struct lu_range *space = &seq->seq_space;
        int rc;
        ENTRY;
        LASSERT(range_is_sane(space));
        if (range_space(space) < LUSTRE_SEQ_SUPER_WIDTH) {
                CWARN("sequences space is going to exhaust soon. "
                      "Only can allocate "LPU64" sequences\n",
                      space->lr_end - space->lr_start);
                *range = *space;
                space->lr_start = space->lr_end;
                rc = 0;
        } else if (range_is_exhausted(space)) {
                CERROR("sequences space is exhausted\n");
                rc = -ENOSPC;
```

```
        } else {
                range_alloc(range, space, LUSTRE_SEQ_SUPER_WIDTH);
                rc = 0;
        }
        if (rc == 0) {
                CDEBUG(D_INFO|D_WARNING,
                       "SEQ-MGR(srv): allocated super-sequence "
                       "["LPX64"-"LPX64"]\n", range->lr_start,
                       range->lr_end);
        }
        RETURN(rc);
}

static int
seq_server_alloc_meta(struct lu_server_seq *seq,
                      struct lu_range *range)
{
        struct lu_range *super = &seq->seq_super;
        int rc = 0;
        ENTRY;
        LASSERT(range_is_sane(super));
        if (range_is_exhausted(super)) {
                if (!seq->seq_cli) {
                        CERROR("no seq-controller client is setup\n");
                        RETURN(-EOPNOTSUPP);
                }

                up(&seq->seq_sem);
                rc = seq_client_alloc_super(seq->seq_cli);
                down(&seq->seq_sem);
                if (rc) {
                        CERROR("can't allocate new super-sequence, "
                               "rc %d\n", rc);
                        RETURN(rc);
                }

                if (seq->seq_cli->seq_range.lr_start > super->lr_start) {
                        *super = seq->seq_cli->seq_range;
                        LASSERT(range_is_sane(super));
                } else {
                        /* race is catched */
                        RETURN(0);
                }
        }
        range_alloc(range, super, LUSTRE_SEQ_META_WIDTH);
        if (rc == 0) {
```

```
                CDEBUG(D_INFO|D_WARNING,
                       "SEQ-MGR(srv): allocated meta-sequence "
                       "["LPX64"-"LPX64"]\n", range->lr_start,
                       range->lr_end);
        }
        RETURN(rc);
}
```

This is handler for sequence allocation related RPCs.

```
static int
seq_server_handle(struct lu_server_seq *seq,
                  const struct lu_context *ctx,
                  struct lu_range *range,
                  __u32 opc)
{
        int rc;
        ENTRY;
        down(&seq->seq_sem);
        switch (opc) {
        case SEQ_ALLOC_SUPER:
                rc = seq_server_alloc_super(seq, range);
                break;
        case SEQ_ALLOC_META:
                rc = seq_server_alloc_meta(seq, range);
                break;
        default:
                rc = -EINVAL;
                break;
        }
        if (rc)
                GOTO(out, rc);
        rc = seq_server_write_state(seq, ctx);
        if (rc) {
                CERROR("can't save state, rc = %d\n",
                       rc);
        }
        EXIT;
out:
        up(&seq->seq_sem);
        return rc;
}
```

4.3 Client allocation stuff

Client has few functions which perform various fid and sequence related allocations. They are the following:

- *seq_client_alloc_super()* - ask server to allocate new super-sequence. This is only used by clients which run on regular MDTs and supported to talk to controller node in super-sequence allocation time;
- *seq_client_alloc_meta()* - ask server to allocate new meta-sequence. This function is only used on client nodes;
- *seq_client_alloc_seq()* - allocate new sequence number from last obtained meta-sequence. If meta-sequence is exhausted - ask for new one. This function is used on client nodes only in FIDs allocation time;
- *seq_client_alloc_fid()* - allocate new FID in current sequence. This is only used on client nodes in the time when new FID is going to be allocate (create files or directories, etc).

This is function sending sequence allocation RPCs to server.

```
static int
seq_client_rpc(struct lu_client_seq *seq,
               struct lu_range *range,
               __u32 opc)
{
        struct obd_export *exp = seq->seq_exp;
        int repsize = sizeof(struct lu_range);
        int rc, reqsize = sizeof(__u32);
        struct ptlrpc_request *req;
        struct lu_range *ran;
        __u32 *op;
        ENTRY;
        req = ptlrpc_prep_req(class_exp2cliimp(exp),
                              LUSTRE_MDS_VERSION,
                              SEQ_QUERY, 1, &reqsize,
                              NULL);
        if (req == NULL)
                RETURN(-ENOMEM);
        op = lustre_msg_buf(req->rq_reqmsg, 0, sizeof(*op));
        *op = opc;
        req->rq_replen = lustre_msg_size(1, &repsize);
        req->rq_request_portal = MDS_SEQ_PORTAL;
        rc = ptlrpc_queue_wait(req);
        if (rc)
                GOTO(out_req, rc);
        ran = lustre_swab_repbuf(req, 0, sizeof(*ran),
```

```
                                    lustre_swab_lu_range);
        if (ran == NULL) {
                CERROR("invalid range is returned\n");
                GOTO(out_req, rc = -EPROTO);
        }
        *range = *ran;
        EXIT;
out_req:
        ptlrpc_req_finished(req);
        return rc;
}
```

Allocate new super-sequence.

```
static int
__seq_client_alloc_super(struct lu_client_seq *seq)
{
        int rc;
        ENTRY;
        rc = seq_client_rpc(seq, &seq->seq_range, SEQ_ALLOC_SUPER);
        if (rc == 0) {
                CDEBUG(D_INFO|D_WARNING,
                       "SEQ-MGR(cli): allocated super-sequence "
                       "["LPX64"-"LPX64"]\n", seq->seq_range.lr_start,
                       seq->seq_range.lr_end);
        }
        RETURN(rc);
}
int
seq_client_alloc_super(struct lu_client_seq *seq)
{
        int rc;
        ENTRY;
        down(&seq->seq_sem);
        rc = __seq_client_alloc_super(seq);
        up(&seq->seq_sem);
        RETURN(rc);
}
EXPORT_SYMBOL(seq_client_alloc_super);
```

Allocate new meta-sequence.

```
static int
__seq_client_alloc_meta(struct lu_client_seq *seq)
{
        int rc;
```

```
        ENTRY;
        rc = seq_client_rpc(seq, &seq->seq_range, SEQ_ALLOC_META);
        if (rc == 0) {
                CDEBUG(D_INFO|D_WARNING,
                       "SEQ-MGR(cli): allocated meta-sequence "
                       "["LPX64"-"LPX64"]\n", seq->seq_range.lr_start,
                       seq->seq_range.lr_end);
        }
        RETURN(rc);
}
int
seq_client_alloc_meta(struct lu_client_seq *seq)
{
        int rc;
        ENTRY;
        down(&seq->seq_sem);
        rc = __seq_client_alloc_meta(seq);
        up(&seq->seq_sem);
        RETURN(rc);
}
EXPORT_SYMBOL(seq_client_alloc_meta);
```

Allocate new sequence number.

```
static int
__seq_client_alloc_seq(struct lu_client_seq *seq,
                       __u64 *seqnr)
{
        int rc = 0;
        ENTRY;
        LASSERT(range_is_sane(&seq->seq_range));
        if (range_space(&seq->seq_range) == 0) {
                rc = __seq_client_alloc_meta(seq);
                if (rc) {
                        CERROR("can't allocate new meta-sequence, "
                               "rc %d\n", rc);
                }
        }
        *seqnr = seq->seq_range.lr_start;
        seq->seq_range.lr_start += 1;
        if (rc == 0) {
                CDEBUG(D_INFO|D_WARNING, "SEQ-MGR(cli): allocated "
                       "sequence ["LPX64"]\n", *seqnr);
        }
        RETURN(rc);
```

```
}
int
seq_client_alloc_seq(struct lu_client_seq *seq,
                     __u64 *seqnr)
{
        int rc = 0;
        ENTRY;
        down(&seq->seq_sem);
        rc = __seq_client_alloc_seq(seq, seqnr);
        up(&seq->seq_sem);
        RETURN(rc);
}
EXPORT_SYMBOL(seq_client_alloc_seq);
```

Allocate new FID.

```
int
seq_client_alloc_fid(struct lu_client_seq *seq,
                     struct lu_fid *fid)
{
        __u64 seqnr = 0;
        int rc;
        ENTRY;
        LASSERT(fid != NULL);
        down(&seq->seq_sem);
        if (!fid_is_sane(&seq->seq_fid) ||
            fid_oid(&seq->seq_fid) >= LUSTRE_SEQ_WIDTH)
        {
                rc = __seq_client_alloc_seq(seq, &seqnr);
                if (rc) {
                        CERROR("can't allocate new sequence, "
                               "rc %d\n", rc);
                        GOTO(out, rc);
                }
                seq->seq_fid.f_oid = LUSTRE_FID_INIT_OID;
                seq->seq_fid.f_seq = seqnr;
                seq->seq_fid.f_ver = 0;
                rc = -ERESTART;
        } else {
                seq->seq_fid.f_oid += 1;
                rc = 0;
        }
        *fid = seq->seq_fid;
        LASSERT(fid_is_sane(fid));
        CDEBUG(D_INFO, "SEQ-MGR(cli): allocated FID "DFID3"\n",
```

```
                PFID3(fid));
        EXIT;
out:
        up(&seq->seq_sem);
        return rc;
}
EXPORT_SYMBOL(seq_client_alloc_fid);
```

As it is seen from last function above, in the case client switches to new sequence, it informs caller with -ERESTART error code, to let it perform additional actions like FLD setup, etc.

5 State Specification

5.1 Resources Involved and Their State

There are three types of resources which change the state. They are the following:

- sequences space range on controller node, from which all super-sequences are allocated;
- super-sequence range on regular MDT, from which all meta-sequences are allocated;
- meta-sequence range on client node, from which all sequences are allocated.

All they are kind of resource which is allocated (consumed) while cluster is running and new objects in new sequences are allocated. These allocations are state transitions and they should be protected. Also, allocating new range on servers means also saving changed range to backing store. And this saving and allocation should be atomic.

5.2 Locking

No special locking schema is foreseen for any kind of resources involved. The only one semaphore (*seq->seq_ sem*) is used on servers to protect ranges allocation and make allocation and storing them for recovery atomic. Also on clients semaphore is used to serialize new fid allocation and meta-sequence allocation.

5.3 Recovery

Recovery is built on the following considerations.

5.3.1 Controller node

Each time as new super-sequence is allocated from sequences space, controller node should save reduced sequences space range to backing store. And each time as controller allocates new meta-sequence to its clients, it also saves super-sequence from which meta-sequences were allocate o backing store. So that, controller node wants to save two ranges:

- sequences space range, that is *seq->seq_space*;
- super-sequence range, that is *seq->seq_super*;

5.3.2 Regular MDT

Each time as regular MDT allocates new meta-sequence to its clients, it saves its super-sequence to backing store. So that, regular MDT only wants to save super-sequence, that is, *seq->seq_super*.

[1]In both cases, it is very important, that servers save changed ranges from which super-sequence and meta-sequence are allocate to backing store *before* sending the reply to client.

Then in recovery time, when MDT is down and got back, sequences space range and super-sequence range on controller node may be read from backing store and used to continue the work. And on regular MDT only last obtained super-sequence will be read from backing store what allows it to continue to allocate new meta-sequences to clients.

6 Environment

6.1 Scalability

The only possible issue in this field as for scalability may be about big number of MDT-MDT sequence synchronizing RPCs. Though servers exchange super-sequences in connect time or when they exhaust, this should not make any problems in big clusters as well, because this is quite rare operation.

6.2 Disk Format Changes

On servers, there is need to save super-sequence to backing store.

[1]