CMD MDS Recovery DLD

Mike Pershin

February 6, 2008

1 Introduction

This document describes recovery changes in CMD.

2 Requirements

The CMD environment requires the reviewed recovery due to major changes in functionality. The new recovery design covers the following issues:

- recovery cases occurred due to cross-ref situations (MDS-MDS recovery);
- land all recovery fixes from CMD2 project;
- changes in recovery logic due to FID introducing;
- changes due to moving network things to the MDT level;
- changes in recovery API due to layered nature of the new MDS.

3 Functional specification

3.1 Recovery process in the new MDS

The recovery starts as usual - while setup process, the MDT reads last_rcvd file. If it's size is not zero then recovery starts by setting `mdt->mdt_recovering = 1`. After that connecting clients will send the replay request.

The new MDS consist of several layers and almost all of them have own part in recovery process. There are several stages in recovery process:

1. Network recovery. MDT receives the replay requests and handles them as usual

   (a) the `transno` value is taken from request. This value is used for operation and is returned back;
   (b) `mdt_handle()` receives the request and processes them in order of `transno`.

2. Clients finish recovery and target_finish_recovery() calls `mds_postrecov()` on MDS OBD. The same method is defined for `lu_device`, so it invoke post-recovery process in all MDS layers;

3. MDD does LOV synchronize and orphan handling in `mdd_postrecov()`

3.2 Transaction handling in new MDS

The layering nature of the new MDS implies that transactions can be done in MDD layer only. But other layers like MDT and CMM need to write persistent data in several cases. The transaction callbacks were designed to provide such capability. There are three callbacks that every layer can register and use - start, stop and commit one. During start callback it is possible to increase the number of blocks involved in transaction. The stop callback is need to write any additional persistent data in the current transaction. The commit callback is used in cases where some data should be updated when transaction is committed.

The recovery in CMD uses all of these callbacks.

Each transaction in new MDS has a context - the set of values which are pre-allocated by every layers if needed. This context is initialized when transaction starts and finished when it is committed. The transaction context is used to store any values which exists while transaction exists.

3.2.1 `Transno` handling

The `transno` variable is used to track transactions, every new transaction has own `transno` number which is returned to the client. In new MDT layer the `transno` handling is encapsulated in transaction callbacks.

1. In the stop callback:
   (a) if `mdt_thread_info` contain the `transno != NULL` already then this is replay case and that `transno` will be used;
   (b) otherwise the `mdt->last_transno` value is taken as `transno` and increased;

2. the taken `transno` is stored in `struct mdt_thread_info` for further usage and is stored also in `struct mdt_txn_info` for commit callback;

3. when the request handling is finished in MDT, the reply is prepared and `transno` is taken from `mdt_thread_info()` back and written to the reply;

4. when transaction is committed the MDT commit callback takes `transno` value from `mdt_txn_info()` and uses it to update `mdt->last_committed` value if needed.

3.2.2 LAST_RCVD update

The LAST_RCVD file should be updated for recovery. This should be done at the end of any new transaction and the stop callback is natural place for this. The MDT stop callback reads and updates the LAST_RCVD file in the same manner as before. Also, the start callback is used to reserve enough space in transaction handle to do `LAST_RCVD` update.

The layered MDS introduces one problem with LAST_RCVD update. The result code should be written into record but there is no last result code at the time of calling the stop_callback and also that result code should be passed from thread to the callback somehow.

The result code will be passed through `txn_key` in the `mdd_trans_stop()` method. The stop callback will use this value while updating the LAST_RCVD. This implies that the 'rc' code from MDD operation shouldn't be changed after writing to the LAST_RCVD.

Note: In some cases, e.g. open() the result code after transaction is stopped could be modified by consequent operations like mdt_mfd_open(), etc. This is ok, while reconstruct we will repeat these steps again and result code will be updated in the same manner. The important thing with storing 'rc' is that it is result of operation which change persistent data and cannot be repeated once done, that 'rc' is needed to know persistent state. All other operations can be just repeated.

3.2.3 Last committed transaction update

In normal operation the last_committed value is managed in the MDT commit callback. If committed transno is higher than mdt->last_committed value then last_committed = transno. The last committed value is returned to the client after that, so client can purge the committed requests from replay queue.

3.3 Reconstruction

The MDT contains the methods for reconstruction of the requests. During reconstruction in the all cases the `LAST_RCVD` is read and result code is got from it. This is enough for most reconstruction cases, but some needs more work to be done:

create, setattr - get attributes. If object is remote one - return `-EREMOTE` to the client, so it will request the attributes from remote server.

open - most complex case for reconstruction. Intent disposition value is saved also in `LAST_RCVD` and is used with result code while reconstruction. If file was created while open, the its attributes and lov data are returned. The open() recovery is a matter of separated DLD.

unlink, link, rename, close - return only result of operation from `LAST_RCVD`

3.4 Replay changes

While replaying the stop callback will use `transno` from the request. This is the only difference with normal case.

3.4.1 FID and replay

Replay functionality on the MDS looks similar to the usual operations in the most cases and reuse the usual methods greatly because of FID design. The open/create replay becomes simpler due to the fact that client always knows FID for open/create operation so there is no separate code-path for replay operations.

4 Use cases

All use cases are intended to 11/17 tests. The cluster is started up and clients runs multiple applications. One of the nodes is failed and other nodes/clients shouldn't be affected. Due to requirements of test 17 the recovery time should be not more than 5 minutes in each case.

4.1 Client failure

Client is failed and normal operations should continue. Other clients and servers should receive no errors. The major areas in MDS recovery in that case are closing all opened files from that client and reading LAST_RCVD records upon reconnection.

4.2 Singe MDS failure in CMD

One MDS is failed, recovered and normal operations continue. All aspects of recovery can happens here - resent/reconstruct, replay. This is the major use case for MDS recovery.

4.3 OST failure

OST is failed and recovery is started, clients should get no errors.

5 Logic specification

5.1 Recovery process

5.1.1 Starting the recovery

```
static int mdt_init_clients_data(const struct lu_context *ctxt,
                                 struct mdt_device *mdt,
                                 unsigned long last_rcvd_size)
{
        struct mdt_server_data *msd = &mdt->mdt_msd;
        struct mdt_client_data *mcd;
        struct obd_device *obd = mdt->mdt_md_dev.md_lu_dev.ld_obd;
        loff_t off = 0;
        int cl_idx;
        int rc = 0;
        ENTRY;
        /* When we do a clean MDS shutdown, we save the last_transno into
         * the header.  If we find clients with higher last_transno values
         * then those clients may need recovery done. */
        OBD_ALLOC_PTR(mcd);
        if (!mcd)
                RETURN(rc = -ENOMEM);
        for (cl_idx = 0, off = le32_to_cpu(msd->msd_client_start);
             off < last_rcvd_size; cl_idx++) {
                __u64 last_transno;
                struct obd_export *exp;
                struct mdt_export_data *med;
                off = le32_to_cpu(msd->msd_client_start) +
                        cl_idx * le16_to_cpu(msd->msd_client_size);
                rc = mdt->mdt_last_rcvd->do_body_ops->dbo_read(ctxt,
                                                          mdt->mdt_last_rcvd,
                                                          mcd,
                                                          sizeof(*mcd), &off);
                if (rc == sizeof(*mcd))
                        rc = 0;
                else if (rc >= 0)
                        rc = -EFAULT;
                if (rc) {
                        CERROR("error reading MDS %s idx %d, off %llu: rc %d\n",
                               LAST_RCVD, cl_idx, off, rc);
                        break; /* read error shouldn't cause startup to fail */
                }
                if (mcd->mcd_uuid[0] == '\0') {
                        CDEBUG(D_INFO, "skipping zeroed client at offset %d\n",
                                cl_idx);
                        continue;
                }
                last_transno = le64_to_cpu(mcd->mcd_last_transno);
                /* These exports are cleaned up by mdt_obd_disconnect(), so
                 * they need to be set up like real exports as
                 * mdt_obd_connect() does.
                 */
                CDEBUG(D_HA, "RCVRNG CLIENT uuid: %s idx: %d lr: "LPU64
                       " srv lr: "LPU64" lx: "LPU64"\n", mcd->mcd_uuid, cl_idx,
                       last_transno, le64_to_cpu(msd->msd_last_transno),
                       le64_to_cpu(mcd->mcd_last_xid));
                exp = class_new_export(obd, (struct obd_uuid *)mcd->mcd_uuid);
                if (IS_ERR(exp))
                        GOTO(err_client, rc = PTR_ERR(exp));
                med = &exp->exp_mdt_data;
                med->med_mcd = mcd;
                rc = mdt_client_add(ctxt, mdt, med, cl_idx);
                LASSERTF(rc == 0, "rc = %d\n", rc); /* can't fail existing */
                exp->exp_replay_needed = 1;
                exp->exp_connecting = 0;
                obd->obd_recoverable_clients++;
                obd->obd_max_recoverable_clients++;
                class_export_put(exp);
                CDEBUG(D_OTHER, "client at idx %d has last_transno = "LPU64"\n",
                       cl_idx, last_transno);
                spin_lock(&mdt->mdt_transno_lock);
                if (last_transno > mdt->mdt_last_transno)
                        mdt->mdt_last_transno = last_transno;
                spin_unlock(&mdt->mdt_transno_lock);
        }
err_client:
        OBD_FREE_PTR(mcd);
        RETURN(rc);
}
static int mdt_init_server_data(const struct lu_context *ctxt,
                                struct mdt_device *mdt)
{
        struct mdt_server_data *msd = &mdt->mdt_msd;
        struct mdt_client_data *mcd = NULL;
        struct obd_device      *obd = mdt->mdt_md_dev.md_lu_dev.ld_obd;
        loff_t                  off = 0;
        unsigned long           last_rcvd_size = 0;
        __u64                   mount_count;
        int                     cl_idx;
        int                     rc;
        struct mdt_thread_info *info;
        struct dt_object       *last = mdt->mdt_last_rcvd;
        struct lu_attr         *la;
        ENTRY;

        ...

        last_rcvd_size = la->la_size;
        if (last_rcvd_size == 0) {
                LCONSOLE_WARN("%s: new disk, initializing\n", obd->obd_name);
                memcpy(msd->msd_uuid, obd->obd_uuid.uuid,sizeof(msd->msd_uuid));
                msd->msd_last_transno = 0;
                mount_count = msd->msd_mount_count = 0;
                msd->msd_server_size = cpu_to_le32(LR_SERVER_SIZE);
                msd->msd_client_start = cpu_to_le32(LR_CLIENT_START);
                msd->msd_client_size = cpu_to_le16(LR_CLIENT_SIZE);
                msd->msd_feature_rocompat = cpu_to_le32(OBD_ROCOMPAT_LOVOBJID);
                msd->msd_feature_incompat = cpu_to_le32(OBD_INCOMPAT_MDT |
                                                        OBD_INCOMPAT_COMMON_LR);
        } else {
                rc = last->do_body_ops->dbo_read(ctxt, last, msd,
                                                 sizeof(*msd), &off);
                if (rc == sizeof(*msd))
                        rc = 0;
                else if (rc >= 0)
                        rc = -EFAULT;
                if (rc) {
                        CERROR("error reading MDS %s: rc %d\n", LAST_RCVD, rc);
                        GOTO(out, rc);
                }
                if (strcmp(msd->msd_uuid, obd->obd_uuid.uuid) != 0) {
                        LCONSOLE_ERROR("Trying to start OBD %s using the wrong"
                                       " disk %s. Were the /dev/ assignments "
                                       "rearranged?\n",
                                       obd->obd_uuid.uuid, msd->msd_uuid);
                        GOTO(out, rc = -EINVAL);
                }
                mount_count = le64_to_cpu(msd->msd_mount_count);
        }
        ...

        mdt_init_clients_data(ctxt, mdt, last_rcvd_size);
        obd->obd_last_committed = mdt->mdt_last_transno;
        if (obd->obd_recoverable_clients) {
                CWARN("RECOVERY: service %s, %d recoverable clients, "
                      "last_transno "LPU64"\n", obd->obd_name,
                                obd->obd_recoverable_clients, mdt->mdt_last_transno);
                        obd->obd_next_recovery_transno = obd->obd_last_committed + 1;
                        obd->obd_recovering = 1;
                        obd->obd_recovery_start = CURRENT_SECONDS;
                        /* Only used for lprocfs_status */
                        obd->obd_recovery_end = obd->obd_recovery_start +
                                OBD_RECOVERY_TIMEOUT;
                }
```

Recovery starts if the `obd->obd_recoverable_clients > 0`. The `obd->obd_recovering` is set in 1 until recovery will finish and `obd_postrecov()` will be invoked.

5.1.2 Post-recovery

New method for `lu_device` is defined - `lu_post_recovery()`.

```
struct lu_device_operations {
        struct lu_object *(*ldo_object_alloc)(const struct lu_context *ctx,
                                              const struct lu_object_header *h,
                                              struct lu_device *d);
        int  (*ldo_process_config)(const struct lu_context *ctx,
                                   struct lu_device *, struct lustre_cfg *);
        int  (*ldo_recovery_complete)(const struct lu_context *ctx,
                                      struct lu_device *)
};
```

The MDD will use this to finish recovery process by doing lov synchronize and destroying the unlinked objects on OST.

```
int  mdd_recovery_complete(const struct lu_context *ctx, struct lu_device *ld)
{
        struct mdd_device *mdd = lu2mdd_dev(ld);
        struct obd_device *obd = mdd2_obd(mdd);
        rc = mdd_lov_set_nextid(ctx, mdd);
        if (rc) {
                CERROR("%s: mdd_lov_set_nextid failed %d\n",
                       obd->obd_name, rc);
                GOTO(out, rc);
        }
        rc = mdd_cleanup_unlink_llog(ctx, mdd);

        obd_notify(obd->u.mds.mds_osc_obd, NULL,
                   obd->obd_async_recov ? OBD_NOTIFY_SYNC_NONBLOCK :
                      OBD_NOTIFY_SYNC, NULL);
        RETURN(rc);
}
```

OSD will use that method to invoke orphans cleanup on bottom filesystem.

```
int  osd_recovery_complete(const struct lu_context *ctx, struct lu_device *ld)
{
        struct osd_device *osd = lu2osd_dev(ld);
        int rc;
        /* recovery is done, so all re-opens are done,
           opened orphans are pinned
           and only non-opened orphans will be deleted */
        rc = ldiskfs_orphans_cleanup(...);
        return rc;
}
```

5.2 Handling the `transno` value

```
static int mdt_txn_stop_cb(const struct lu_context *ctx,
                           struct dt_device *dev,
                           struct thandle *txn, void *cookie)
{
        struct mdt_device *mdt = cookie;
        struct mdt_txn_info *txni;
        struct mdt_thread_info *mti;
        int rc;
        /* transno is in two contexts - for commit_cb and for thread */
        txni = lu_context_key_get(&txn->th_ctx, &mdt_txn_key);
        mti = lu_context_key_get(ctx, &mdt_thread_key);

        spin_lock(&mdt->mdt_transno_lock);
        if (mti->mti_transno == 0) {
                mti->mti_transno = ++ mdt->mdt_last_transno;
        } else {
                /* replay case */
                if (mti->mti_transno > mdt->mdt_last_transno)
                        mdt->mdt_last_transno = mti->mti_transno;
        }
        spin_unlock(&mdt->mdt_transno_lock);
        /* save transno for the commit callback */
        txni->txi_transno = mti->mti_transno;
        /* Update last_rcvd records with latest transaction data */
        rc = mdt_update_last_rcvd(mti, dev, thandle);
        return rc;
}
/* commit callback is used to update last_commited value */
static int mdt_txn_commit_cb(const struct lu_context *ctx,
                             struct dt_device *dev,
                             struct thandle *txn, void *cookie)
{
        struct mdt_device *mdt = cookie;
        struct obd_device *obd = md2lu_dev(&mdt->mdt_md_dev)->ld_obd;
        struct mdt_txn_info *txi;
        txi = lu_context_key_get(&txn->th_ctx, &mdt_txn_key);
        spin_lock(&mdt->mdt_last_committed_lock);
        if (txi->txi_transno > mdt->mdt_last_committed) {
                mdt->mdt_last_committed = txi->txi_transno;
                spin_unlock(&mdt->mdt_last_committed_lock);
                ptlrpc_commit_replies(obd);
        } else
                spin_unlock(&mdt->mdt_last_committed_lock);
        CDEBUG(D_HA, "%s: transno "LPD64" committed\n",
               obd->obd_name, txi->txi_transno);
        return 0;
}
```

5.3 Updating the `LAST_RCVD`

```
enum {
        MDT_TXN_LAST_RCVD_CREDITS = 3
};
/* add credits for last_rcvd update */
static int mdt_txn_start_cb(const struct lu_context *ctx,
                            struct dt_device *dev,
                            struct txn_param *param, void *cookie)
{
        param->tp_credits += MDT_TXN_LAST_RCVD_CREDITS;
        return 0;
}

static int mdt_read_last_rcvd(struct mdt_thread_info *info,
                              struct mdt_client_data *mcd, loff_t *off)
{
        struct mdt_device *mdt = info->mti_mdt;
        int rc;
        rc = mdt->mdt_last_rcvd->do_body_ops->dbo_read(info->mti_ctxt,
                                                       mdt->mdt_last_rcvd,
                                                       mcd, sizeof(*mcd),
                                                           off);
        if (rc == sizeof(*mcd))
                rc = 0;
        else if (rc >= 0)
                rc = -EFAULT;
        return rc;
}
static int mdt_write_last_rcvd(struct mdt_thread_info *info,
                               struct mdt_client_data *mcd,
                               loff_t *off, struct thandle *th)
{
        struct mdt_device *mdt = info->mti_mdt;
        int rc;
        rc = mdt->mdt_last_rcvd->do_body_ops->dbo_write(info->mti_ctxt,
                                                        mdt->mdt_last_rcvd,
                                                        mcd, sizeof(*mcd),
                                                        off, th);
        if (rc == sizeof(*mcd))
                rc = 0;
        else if (rc >= 0)
                rc = -EFAULT;
        return rc;
}
int mdt_update_last_rcvd(struct mdt_thread_info *info, struct dt_device *dt,
                         struct thandle *th)
{
        struct mdt_device *mdt = info->mti_mdt;
        struct ptlrpc_request *req = mdt_info_req(info);
        struct mdt_export_data *med = &req->rq_export->exp_mdt_data;
        struct mdt_client_data *mcd = med->med_mcd;
        loff_t off;
        int err;
        __s32 rc = th->th_result;

        ENTRY;
        /* if the export has already been failed, we have no last_rcvd slot */
        if (req->rq_export->exp_failed) {
                CWARN("commit transaction for disconnected client %s: rc %d\n",
                      req->rq_export->exp_client_uuid.uuid, rc);
                if (rc == 0)
                        rc = -ENOTCONN;
                RETURN(rc);
        }
        off = med->med_lr_off;
        down(&mdt->mdt_mcd_lock);
        mcd->mcd_last_transno = cpu_to_le64(info->mti_transno);
        mcd->mcd_last_xid = cpu_to_le64(req->rq_xid);
        mcd->mcd_last_result = cpu_to_le32(rc);
        mcd->mcd_last_data = cpu_to_le32(op_data);

        if (off <= 0) {
                CERROR("client idx %d has offset %lld\n", med->med_lr_idx, off);
                err = -EINVAL;
        } else {
                err = mdt_write_last_rcvd(info, mcd, &med->med_lr_off, th);
        }
        up(&mdt->mdt_mcd_lock);
        RETURN(err);
}
```

5.4 Resent and reconstruction

Reconstruction methods are the same as in old MDS but they are using new API.

5.4.1 Generic reconstruct

Most of the operations, e.g. link(), unlink(), rename() need only generic reconstruct:

```
void mdt_reconstruct_generic(struct lu_context *ctxt, struct ptlrpc_request *req)
{
        struct mdt_export_data *med = &req->rq_export->exp_mdt_data;
        mdt_req_from_mcd(req, med->med_mcd);
}
void mds_req_from_mcd(struct ptlrpc_request *req, struct mdt_client_data *mcd)
{
        DEBUG_REQ(D_HA, req, "restoring transno "LPD64"/status %d",
                  mcd->mcd_last_transno, mcd->mcd_last_result);
        req->rq_repmsg->transno = req->rq_transno = mcd->mcd_last_transno;
        req->rq_repmsg->status = req->rq_status = mcd->mcd_last_result;
        mds_steal_ack_locks(req);
}
```

5.4.2 reconstruct_create(), reconstruct_setattr()

```
static void reconstruct_reint_create(struct mdt_thread_info *info)
{
        struct ptlrpc_request  *req = mdt_info_req(info);
        struct mdt_export_data *med = &req->rq_export->exp_mdt_data;
        struct mdt_device *mdt = info->mti_mdt;
        struct mdt_object *child;
        struct mdt_body *body;
        mdt_req_from_mcd(req, med->med_mcd);
        if (req->rq_status)
                return;
        /* if no error, so child was created with requested fid */
        child = mdt_object_find(info->mti_ctxt, mdt, info->mti_rr.rr_fid2);
        LASSERT(!IS_ERR(child));
        body = req_capsule_server_get(&info->mti_pill, &RMF_MDT_BODY);
        rc = mo_attr_get(ctxt, mdt_object_child(child), &info->mti_attr);
        if (rc == -EREMOTE) {
                /* object was created on remote server */
                body->valid |= OBD_MD_MDS;
        }
        mdt_pack_attr2body(body, &info->mti_attr.ma_attr,
                           info->mti_rr.rr_fid2);
        mdt_object_put(info->mti_ctxt, child);
}
static void reconstruct_reint_setattr(struct mdt_thread_info *info)
{
        struct ptlrpc_request  *req = mdt_info_req(info);
        struct mdt_export_data *med = &req->rq_export->exp_mdt_data;
        struct mdt_device *mdt = info->mti_mdt;
        struct mdt_object *obj;
        struct mdt_body *body;

        mds_req_from_mcd(req, med->med_mcd);
        if (req->rq_status)
                return;

        body = req_capsule_server_get(&info->mti_pill, &RMF_MDT_BODY);
        obj = mdt_object_find(info->mti_ctxt, mdt, info->mti_rr.rr_fid1);
        LASSERT(!IS_ERR(obj));
        mo_attr_get(ctxt, mdt_object_child(obj), &info->mti_attr);
        mdt_pack_attr2body(body, &info->mti_attr.ma_attr,
                           info->mti_rr.rr_fid1);
        /* Don't return OST-specific attributes if we didn't just set them */
        if (rec->ur_iattr.ia_valid & ATTR_SIZE)
                body->valid |= OBD_MD_FLSIZE | OBD_MD_FLBLOCKS;
        if (rec->ur_iattr.ia_valid & (ATTR_MTIME | ATTR_MTIME_SET))
                body->valid |= OBD_MD_FLMTIME;
        if (rec->ur_iattr.ia_valid & (ATTR_ATIME | ATTR_ATIME_SET))
                body->valid |= OBD_MD_FLATIME;
        mdt_object_put(info->mti_ctxt, obj);
}
```

6 State management

6.1 State invariants

FID is invariant so recovery become simpler because uses the same FID as ordinary operations did.