DLD for OSD Transaction API in CMD

Huang Hua

Jul 25, 2006

Contents

1 Introduction

This DLD document describes the details of OSD transaction API design, implementation and usage in CMD. The basic theory keeps the same as before. But due to new layering of CMD, there are some differences.

2 Functional Specification

2.1 Abstract

In CMD, we now have the following algorithm for transaction:

1. OSD exports transaction start and stop API to others in its data device operation.
2. MDD is the main user of these transaction APIs. Some other layers or modules may also use these transaction APIs if they want to directly access OSD interfaces.
3. Users of transaction APIs also can register callbacks to OSD: including transaction start and stop hooks, and commit callback. Before a transaction is really started, registered start hooks will be called, and users can do something to this transaction if necessary. Before a transaction is really stopped, registered stop hooks will be called, and users can also do something if necessary. After a transaction has committed, all registered commit callbacks will be called.
4. OSD registers its own callback for every transaction into JBD, and JBD will do callback when transaction committed. This is done by patching Linux JBD subsystem.
5. Every transaction has its own lu_context attached with it. So, users of transaction hooks and callbacks can add something they need to this context, and set and retrieve these information if necessary in hooks and callbacks.

6. Some layer can register this kind of callbacks to OSD, e.g MDT, CMM. The following example is based on MDT. When transaction start hook is executed, MDT add its some transaction block credit to this transaction for update of LAST_RCVD file. When transaction stop hook is executed, MDT set its own transaction number into transaction context; This transaction number is returned to client for recovery usage. When transaction is stopped, the transaction return value is also returned to MDT in the handle. So, MDT can update LAST_RCVD file for recovery issue. When transaction commit callback is executed, MDT retrieves this transaction number, and tells ptlrpc that this transaction has committed.

2.2 Data structures

2.2.1 general purpose transaction handle

This is the general purpose transaction handle, returned by OSD to users. Users should use this handle to do any other operations.

```
struct thandle {
        struct dt_device *th_dev; /* this OSD */
        struct lu_context th_ctx; /* transaction context */
};
```

Every transaction handle contains its own transaction context: th_ctx. When a transaction is started, its context is allocated by OSD; When the transaction is committed, its context is deallocated by OSD. Users can put its data in this context, such as transaction number from MDT, and others.

2.2.2 OSD transaction handle

This is the OSD related transaction handle, derived from the general purpose transaction handle, containing some OSD specific and JBD related data structures. If we use other underlying file system instead of ext3/JBD, we may have different data structures. But we always have the same format for general purpose transaction handle.

```
struct osd_thandle {
        struct thandle          ot_super;
        handle_t               *ot_handle; /* pointer to jbd handle */
        struct journal_callback ot_jcb;    /* jbd callback */
};
```

2.2.3 general purpose transaction parameter

```
struct txn_param {
        /* number of blocks this transaction will modify */
        unsigned int tp_credits;
};
```

2.2.4 transaction callbacks

```
/*
 * Transaction call-backs.
 *
 * These are invoked by osd (or underlying transaction engine) when
 * transaction changes state.
 *
 * Call-backs are used by upper layers to modify transaction parameters and
 * perform some actions on for each transaction state transition.
 */
struct dt_txn_callback {
        int (*dtc_txn_start)(const struct lu_context *ctx,
                             struct dt_device *dev,
                             struct txn_param *param, void *cookie);
        int (*dtc_txn_stop)(const struct lu_context *ctx,
                            struct dt_device *dev,
                            struct thandle *txn, void *cookie);
        int (*dtc_txn_commit)(const struct lu_context *ctx,
                              struct dt_device *dev,
                              struct thandle *txn, void *cookie);
        /* cookie is set by user, and will be filled as an argument
         * for hooks and callbacks */
        void            *dtc_cookie;
        /* link this callback to global list */
        struct list_head dtc_linkage;
};
```

2.3 Interfaces

2.3.1 start a transaction

```
struct thandle *
osd_trans_start(const struct lu_context *ctx,
                struct dt_device *d,
                struct txn_param *p);
```

Parameters: const struct lu_context *ctx - thread execution context;

struct dt_device *d - this OSD data device;
struct txn_param *p - required transaction parameter.

Return Value: valid transaction handle if succeed; error code if fail.

Description: This is a data device operation provided by OSD to start a transaction. The important thing about start is the setting the number of blocks involved in transaction. Also there is prepared commit callback and registered to JBD.

2.3.2 stop a transaction

```
void osd_trans_stop(const struct lu_context *ctx,
                    struct thandle *th)
```

Parameters: const struct lu_context *ctx - thread execution context;
struct thandle *th - transaction handle.

Return Value: void

Description: stop a transaction when we have completed some operations. The transaction handle passed to this function should be the same returned by corresponding transaction start routine. After this call, this transaction handle has no meaning to user.

2.3.3 register transaction hook and callback

```
void dt_txn_callback_add(struct dt_device *dev,
                         struct dt_txn_callback *cb)
```

Parameters: struct dt_device *dev - data device to register transaction hook and callback to, mostly are OSD;
struct dt_txn_callback *cb - callback data structure filled with information by caller.

Return Value: void

Description: register a transaction hook & callback to OSD data device. This is called by MDT or other layer who is interested in transaction.

2.3.4 unregister transaction hook and callback

```
void dt_txn_callback_del(struct dt_device *dev,
                         struct dt_txn_callback *cb)
```

Parameters: struct dt_device *dev - data device to register transaction hook and callback to, mostly are OSD;
struct dt_txn_callback *cb - callback data structure.

Return Value: void

Description: unregister the hook and callback from underlying data device.

3 Use Cases

3.1 What to do with transactions?

This use case shows things and algorithms related to new transaction APIs.

1. MDT gets a request from client or other server, unpacking this request, doing some checking and preparation, and calling corresponding meta data operation of the underlying device (that is CMM). MDT registers a transaction callback to OSD when startup.
2. CMM decides how to distribute metadata in cluster, remotely or local (by asking FLD). Then CMM dispatches this request to local MDD or to remote MDT through mdc. MDD registers a transaction callback to OSD when startup
3. MDD starts a new transaction if necessary by calling the start transaction API from OSD. For example, MDD need to start transaction for create, setattr, etc.
   (a) OSD start transaction API is called. OSD calls all registered start transaction hooks before really starts a new transaction.
      i. Because MDT have registered a transaction start hook, the hook function will be called, and MDT can do something with the transaction parameter.
   (b) OSD starts a new transaction, initializes the transaction context, adds commit callback for this transaction to JBD.

4. MDD does some operations according to the request, by calling corresponding OSD interfaces and operations.
5. MDD stops this transaction after completion.
   (a) OSD stop transaction API is called. OSD calls all registered stop transaction hooks before really stop the transaction.
      i. MDT registered stop transaction hook is called. So MDT can do something with this transaction, such as assign a transaction number to it.
   (b) OSD stops the transaction. At this moment, the transaction has not committed yet, so the transaction handle will be deallocated when the transaction is committed sometime in the future.
6. Control returns from MDD, CMM to MDT. MDT should pack the reply message, return results to clients or other servers. If some transaction is started and not committed yet, the request is kept for replay. Please refer to recovery HLD and DLD.
7. (sometime later) the transaction is committed in JBD. So the OSD commit callback is called by JBD.
   (a) OSD calls all registered commit callbacks one by one.
      i. So MDT registered commit transaction callback is called, and MDT should update its last committed transaction number, schedule some requests with committed transaction to be replied.
   (b) This transaction handle will be destroyed here. PLEASE NOTICE, THIS COMMIT MAYBE HAPPEN RIGHT AFTER TRANSACTION IS STOPPED, AND BEFORE ANY OTHER OPERATION IN (5b).

3.2 Use case in MDD

This is the unlink handler in MDD, showing how to use OSD transaction API and lock API:

1. start a transaction by calling osd transaction API;
2. lock some object if necessary;
3. do some operation, such as insert index, add reference;
4. unlock the object;
5. stop the transaction.

```
static struct thandle* mdd_trans_start(const struct lu_context *ctxt,
                                       struct mdd_device *mdd)
{
        struct txn_param *p = &mdd_ctx_info(ctxt)->mti_param;
        return mdd_child_ops(mdd)->dt_trans_start(ctxt, mdd->mdd_child, p);
}
static void mdd_trans_stop(const struct lu_context *ctxt,
                           struct mdd_device *mdd, struct thandle *handle)
{
        mdd_child_ops(mdd)->dt_trans_stop(ctxt, handle);
}
static int mdd_link(const struct lu_context *ctxt, struct md_object *tgt_obj
                    struct md_object *src_obj, const char *name)
{
        struct mdd_object *mdd_tobj = md2mdd_obj(tgt_obj);
        struct mdd_object *mdd_sobj = md2mdd_obj(src_obj);
        struct mdd_device *mdd = mdo2mdd(src_obj);
        struct thandle *handle;
        int rc;
        ENTRY;
        mdd_txn_param_build(ctxt, &MDD_TXN_LINK);
        handle = mdd_trans_start(ctxt, mdd);
        if (IS_ERR(handle))
                RETURN(PTR_ERR(handle));
        mdd_lock2(ctxt, mdd_tobj, mdd_sobj);
        rc = mdd_link_sanity_check(ctxt, mdd_tobj, mdd_sobj);
        if (rc)
                GOTO(out, rc);
        rc = __mdd_index_insert(ctxt, mdd_tobj, lu_object_fid(&src_obj->mo_l
                                name, handle);
        if (rc == 0)
                __mdd_ref_add(ctxt, mdd_sobj, handle);
out:
        mdd_unlock2(ctxt, mdd_tobj, mdd_sobj);
        mdd_trans_stop(ctxt, mdd, handle);
        RETURN(rc);
}
```

4 Logic Specification

4.1 start a transaction

```
struct thandle *osd_trans_start(const struct lu_context *ctx,
```

```
                                struct dt_device *d,
                                struct txn_param *p)
{
        struct osd_device  *dev = osd_dt_dev(d);
        handle_t           *jh;
        struct osd_thandle *oh;
        struct thandle     *th;
        int hook_res;
        ENTRY;
        hook_res = dt_txn_hook_start(ctx, d, p);
        if (hook_res != 0)
                RETURN(ERR_PTR(hook_res));
        if (osd_param_is_sane(dev, p)) {
                OBD_ALLOC_GFP(oh, sizeof *oh, GFP_NOFS)
                if (oh != NULL) {
                        /*
                         * XXX temporary stuff. Some abstraction layer shoul
                         * be used.
                         */
                        jh = journal_start(osd_journal(dev), p->tp_credits);
                        if (!IS_ERR(jh)) {
                                oh->ot_handle = jh;
                                th = &oh->ot_super;
                                th->th_dev = d;
                                lu_device_get(&d->dd_lu_dev);
                                /* add commit callback */
                                lu_context_init(&th->th_ctx, LCT_TX_HANDLE);
                                journal_callback_set(jh, osd_trans_commit_cb
                                                     (struct journal_callbac
                        } else {
                                OBD_FREE_PTR(oh);
                                th = (void *)jh;
                        }
                } else
                        th = ERR_PTR(-ENOMEM);
        } else {
                CERROR("Invalid transaction parameters\n");
                th = ERR_PTR(-EINVAL);
        }
        RETURN(th);
}
```

4.2 stop a transaction

```
static void osd_trans_stop(const struct lu_context *ctx, struct thandle *th)
{
        int result;
        struct osd_thandle *oh;
        ENTRY;
        oh = container_of0(th, struct osd_thandle, ot_super);
        if (oh->ot_handle != NULL) {
                handle_t *hdl = oh->ot_handle;
                result = dt_txn_hook_stop(ctx, th->th_dev, th);
                if (result != 0)
                        CERROR("Failure in transaction hook: %d\n", result);
                /*
                 * XXX temporary stuff. Some abstraction layer should be use
                 */
                oh->ot_handle = NULL;
                result = journal_stop(hdl); /* after this, oh may be invalid
                if (result != 0)
                        CERROR("Failure to stop transaction: %d\n", result);
        }
        EXIT;
}
```

4.3 start transaction hook

```
int dt_txn_hook_start(const struct lu_context *ctx,
                      struct dt_device *dev, struct txn_param *param)
{
        int result;
        struct dt_txn_callback *cb;
        result = 0;
        list_for_each_entry(cb, &dev->dd_txn_callbacks, dtc_linkage) {
                if (cb->dtc_txn_start == NULL)
                        continue;
                result = cb->dtc_txn_start(ctx, dev, param, cb->dtc_cookie);
                if (result < 0)
                        break;
        }
        return result;
}
```

4.4 stop transaction hook

```
int dt_txn_hook_stop(const struct lu_context *ctx,
                     struct dt_device *dev, struct thandle *txn)
{
        int result;
        struct dt_txn_callback *cb;
        result = 0;
        list_for_each_entry(cb, &dev->dd_txn_callbacks, dtc_linkage) {
                if (cb->dtc_txn_stop == NULL)
                        continue;
                result = cb->dtc_txn_stop(ctx, dev, txn, cb->dtc_cookie);
                if (result < 0)
                        break;
        }
        return result;
}
```

4.5 commit transaction callback

This is a callback registered to JBD. JBD will call it when some transaction is committed.

```
static void osd_trans_commit_cb(struct journal_callback *jcb, int error)
{
        struct osd_thandle *oh = container_of0(jcb, struct osd_thandle, ot_j
        struct thandle     *th = &oh->ot_super;
        /* there is no thread context available */
        dt_txn_hook_commit(&th->th_ctx, th->th_dev, th);
        if (th->th_dev != NULL) {
                lu_device_put(&th->th_dev->dd_lu_dev);
                th->th_dev = NULL;
        }
        lu_context_fini(&th->th_ctx);
        OBD_FREE_PTR(oh);
}
```

call registered transaction commit callback one by one:

```
int dt_txn_hook_commit(const struct lu_context *ctx,
                       struct dt_device *dev, struct thandle *txn)
{
        int result;
```

```
        struct dt_txn_callback *cb;
        result = 0;
        list_for_each_entry(cb, &dev->dd_txn_callbacks, dtc_linkage) {
                if (cb->dtc_txn_commit == NULL)
                        continue;
                result = cb->dtc_txn_commit(ctx, dev, txn, cb->dtc_cookie);
                if (result < 0)
                        break;
        }
        return result;
}
```

5 State Specification

5.1 No Nested Transaction Support

Some file systems support nested transactions, like ext3/JBD. BUT, PLEASE ATTENTION: WE DO NOT SUPPORT NESTED TRANSACTION NOW.

5.2 Transaction and DLM Lock

DLM lock should NOT be taken in transaction.

5.3 Transaction and RPC invocation

RPC should be be invoked in transaction.

6 Focus for Inspection