DLD Size-on-MDS Replay and Re-Send.

Vitaly Fertman

2007-08-06

1 Introduction.

Size-on-MDS metadata improvement includes the caching of the inode size, blocks, ctime and mtime on the MDS. Such an attribute caching allows clients to avoid making RPCs to the OSTs to find the attributes encoded in the file objects kept on those OSTs what results in the significantly improved performance of listing directories.

The current DLD describes the changes in the replay and resend procedures made for the SOM related RPC handling.

2 Requirements.

After the recovery, the inode state is properly recovered on MDS and replies are properly reconstructed for resent RPCs.

2.1 Functional requirement.

- Open RPC is kept in the replay list on the client not until close is committed on the sever only, but until IO epoch is closed and SOM attribute update is committed as well.
- MDS reconstructs the replies for SOM related resent RPC.

3 Use Cases.

1. MDS failover after CLOSE, before DONE_WRITING occurs.
   - OPEN for write, WRITE, CLOSE;
   - MDS failover;

- DONE_WRITING and SETATTR send the updated file handle so that MDS can find the object with it.

2. MDS failover after DONE_WRITING, before SETATTR occures.

   Similar to the previous Use Case, but the failover occurs after DONE_WRITING.

3. MDS failover after SETATTR.

   Similar to the previous Use Case, but the failover occurs after SETATTR.

4. MDS failover after the _committed_ CLOSE, before DONE_WRITING occurs.

   - OPEN for write, WRITE, CLOSE;
   - commit all transactions on MDS including CLOSE, inform the client CLOSE is committed;
   - MDS failover, OPEN and CLOSE are replayed so that MDS reconstruct mfd in the proper state.
   - DONE_WRITING and SETATTR send the updated file handle so that MDS can find the object with it.

5. MDS failover after DONE_WRITING, when CLOSE is _committed_, before SETATTR occurs.

   Similar to the previous Use Case, but the failover occurs after DONE_WRITING.

6. MDS failover after SETATTR, when CLOSE is _committed_.

   Similar to the previous Use Case, but the failover occurs after SETATTR.

7. MDS failover after the _committed_ DONE_WRITING, before SETATTR occurs.

   Similar to the previous Use Case, but the failover occurs after DONE_WRITING, which is also committed.

8. MDS failover after SETATTR, when DONE_WRITING is _committed_.

   Similar to the previous Use Case, but the failover occurs after SETATTR.

9. MDS failover after the _committed_ SETATTR.

   Similar to the previous Use Case, but SETATTR is also committed.

10. The reply on a SOM related RPC is lost.

    - CLOSE, DONE_WRITING, truncate SETATTR or attribute update SETATTR comes on MDS;
    - MDS handles it;
    - the reply is lost and the original request is re-sent by the client;
    - MDS properly reconstructs the reply;

11. UNLINK of files with not yet updated SOM attributes.

    - OPEN, WRITE, CLOSE, DONE_WRITING.
    - UNLINK, it destroys OST objects.
    - SOM attribute update performs obd_getattr and gets -ENOENT as objects are already destroyed, it is propagated on MDS to inform it mfd handle is already obsolete.

4 Functional Specification.

4.1 Data definitions.

1. A request sequence is added.

   This is a list of requests that cannot be dropped from the replay list once the sever reports they are committed, it is allowed only after the last request in the sequence is committed.

   ```
   struct ptlrpc_request {
       ...
       struct list_head rq_mod_list;
   };
   ```

2. Mark a request as the sequence end.

   ```
   struct ptlrpc_request {
       ...
       unsigned int rq_sequence:1;
   };
   ```

3. The mdc open data request sequence.

   ```
   The sequence of OPEN, CLOSE, DONE_WRITING, SETATTR requests, as well as SETATTR, DO
   struct mdc_open_data {
       ...
       struct list_head mod_replay_list;
   };
   ```

4.2 MD Interface.

1. int (*m_close)(struct obd_export *, struct md_op_data *, struct mdc_open_data *, struct ptlrpc_request **);

   Work on mdc_open_data instead of obd_client_handle. it is enough to handle the request sequence and even detach it from the och if needed.

2. int (*m_done_writing)(struct obd_export *, struct md_op_data *, struct mdc_open_data *);

   Work on mdc_open_data instead of obd_client_handle. it is enough to handle the request sequence and even detach it from the och if needed.

3. int (*m_setattr)(struct obd_export *, struct md_op_data *, void *, int , void *, int, struct ptlrpc_request **, struct mdc_open_data **mod);

   The new parameter @mod is added. This is a double pointer to mdc_open_data, because truncate starts with SETATTR and opens the sequence and @mod is to be allocated here.

5 Logical Specification.

5.1 PTLRPC.

1. static inline void ptlrpc_close_replay_seq(req)

   The macro to mark the request as the sequence end and drop the replay-delayed flag.

```
spin_lock(&req->rq_lock);
req->rq_replay = 0;
req->rq_sequence = 1;
spin_unlock(&req->rq_lock);
```

5.2 LLITE.

1. static int ll_close_inode_openhandle(struct obd_export *md_exp, struct inode *inode, struct obd_client_handle *och)

   Check if DONE_WRITING or SETATTR is to be sent to server and mark the CLOSE RPC as the sequence end if not.

```
rc = md_close(md_exp, op_data, och->och_mod, &req);
if (rc != -EAGAIN)
    seq = req;
...
if ((exp->exp_connect_flags & OBD_CONNECT_SOM) && !epoch_close &&
    S_ISREG(inode->i_mode) && (och->och_flags & FMODE_WRITE)) {
    ll_queue_done_writing(inode, LLIF_DONE_WRITING);
} else {
    if (seq)
        ptlrpc_close_replay_seq(req);
    ...
}
```

```
if (req)
    ptlrpc_req_finished(req);
return rc;
```

2. int ll_sizeonmds_update(struct inode *inode, struct mdc_open_data *mod, struct lustre_handle *fh, __u64 ioepoch)

   The new parameter @mod is given, pass it to ll_md_setattr.

   If ll_inode_getattr() returns -ENOENT, zero obdo attributes before proceeding, this SOM attribute update with no valid attributes propagates the error on the server to let it clean the SOM related structures on the inode after unlink.

```
rc = ll_inode_getattr(inode, oa);
if (rc == -ENOENT) {
    oa->o_valid = 0;
    CDEBUG(D_INODE, "objid "LPX64" is already destroyed\n",
           lli->lli_smd->lsm_object_id);
}
...
rc = ll_md_setattr(inode, op_data, &mod);
```

3. int ll_md_setattr(struct inode *inode, struct md_op_data *op_data, struct mdc_open_data **mod)

   The new parameter @mod is given, it is a double pointer to mdc_open_data, pass it to md_setattr().

```
rc = md_setattr(sbi->ll_md_exp, op_data, NULL, 0, NULL,
                0, &request, mod);
```

4. int ll_setattr_raw(struct inode *inode, struct iattr *attr)

   mdc_setattr() on truncate is supposed to allocate a new mdc_open_data because this SETATTR is the first RPC in the sequence. Pass a double pointer to this structure to ll_md_setattr() and pass it later to ll_setattr_done_writing().

```
rc = ll_md_setattr(inode, op_data, &mod);
...
if (op_data) {
    if (op_data->op_ioepoch)
        rc1 = ll_setattr_done_writing(inode, op_data, mod);
    ...
}
```

5. static int ll_setattr_done_writing(struct inode *inode, struct md_op_data *op_data, struct mdc_open_data *mod)

   The new parameter @mod is given, pass it to md_done_writing() and ll_sizeonmds_update().

```
rc = md_done_writing(ll_i2sbi(inode)->ll_md_exp, op_data, mod);
if (rc == -EAGAIN) {
    rc = ll_sizeonmds_update(inode, mod, &op_data->op_handle,
                             op_data->op_ioepoch);
}
```

5.3 MDC.

1. int mdc_setattr(struct obd_export *exp, struct md_op_data *op_data, void *ea, int ealen, void *ea2, int ea2len, struct ptlrpc_request **request, struct mdc_open_data **mod)

   Allocate a new mdc_open_data, if this is the EPOCH_OPEN request initiated by truncate, this is the first request in the request sequence.

   Install @mod to the request if given or a new one is allocated, install commit handler, add the request into the sequence list. If this is EPOCH_OPEN, mark the request as replay-delayed, otherwise as sequence end, for truncate and SOM attribute update correspondingly.

   In the case an error occurs, the commit callback is called to clean the request sequence.

```
if (mod && (op_data->op_flags & MF_EPOCH_OPEN) &&
    req->rq_import->imp_replayable)
{
    LASSERT(*mod == NULL);
    OBD_ALLOC_PTR(*mod);
    if (*mod == NULL) {
        DEBUG_REQ(D_ERROR, req, "Can't allocate "
                  "mdc_open_data");
    } else {
        CFS_INIT_LIST_HEAD(&(*mod)->mod_replay_list);
    }
}
if (mod && *mod) {
    req->rq_cb_data = *mod;
    req->rq_commit_cb = mdc_commit_delayed;
    list_add_tail(&req->rq_mod_list, &(*mod)->mod_replay_list);
    if (op_data->op_flags & MF_EPOCH_OPEN)
        req->rq_replay = 1;
    else
        req->rq_sequence = 1;
}
rc = mdc_reint(req, rpc_lock, LUSTRE_IMP_FULL);
...
```

```
if (rc && req->rq_commit_cb)
    req->rq_commit_cb(req);
```

2. int mdc_set_open_replay_data(struct obd_export *exp, struct obd_client_handle *och, struct ptlrpc_request *open_req)

   Initialize the sequence list in mdc_open_data when allocated, insert the given request into it, install mdc_commit_delayed() as the commit callback.

```
CFS_INIT_LIST_HEAD(&mod->mod_replay_list);
...
list_add_tail(&open_req->rq_mod_list, &mod->mod_replay_list);
open_req->rq_commit_cb = mdc_commit_delayed;
```

3. int mdc_close(struct obd_export *exp, struct md_op_data *op_data, struct mdc_open_data *mod, struct ptlrpc_request **request)

   Add the request into the sequence list in @mod, if not NULL @mod is given. Assign mdc_commit_delayed() as the commit callback and mark the request as replay-delayed.

   In the case an error occurs, close the sequence right here and call the commit callback.

```
if (likely(mod != NULL))
    list_add_tail(&req->rq_mod_list, &mod->mod_replay_list);
else
    CDEBUG(D_HA, "couldn't find open req; expecting close error\n");
...
req->rq_commit_cb = mdc_commit_delayed;
req->rq_cb_data = mod;
req->rq_replay = 1;
...
if (rc != 0 && rc != -EAGAIN && req && req->rq_commit_cb) {
    ptlrpc_close_replay_seq(req);
    req->rq_commit_cb(req);
}
```

4. int mdc_done_writing(struct obd_export *exp, struct md_op_data *op_data, struct mdc_open_data *mod)

   The same as for mdc_close().

5. static void mdc_replay_open(struct ptlrpc_request *req)

   Walk through the request sequence in the mdc_open_data attached to the request, update epoch data, namely the file handle, in CLOSE, DONE_WRITING and SETATTR requests.

```
list_for_each_entry_safe(cur, tmp, &mod->mod_replay_list,
                         rq_mod_list)
{
    opc = lustre_msg_get_opc(cur->rq_reqmsg);
    if (opc == MDS_CLOSE || opc == MDS_DONE_WRITING) {
        epoch = lustre_msg_buf(cur->rq_reqmsg,
                               REQ_REC_OFF, sizeof(*epoch));
        LASSERT(epoch);
        DEBUG_REQ(D_HA, cur, "updating %s body with new fh",
                  opc == MDS_CLOSE ? "CLOSE" : "DONE_WRITING");
    } else if (opc == MDS_REINT) {
        rec = lustre_msg_buf(req->rq_reqmsg,
                             REQ_REC_OFF, sizeof (*rec));
        LASSERT(rec && rec->sa_opcode == REINT_SETATTR);
        epoch = lustre_msg_buf(cur->rq_reqmsg,
                               REQ_REC_OFF + 2, sizeof(*epoch));
        LASSERT(epoch);
        DEBUG_REQ(D_HA, cur, "updating REINT_SETATTR body "
                  "with new fh");
    }
    if (epoch) {
        if (och != NULL)
            LASSERT(!memcmp(&old, &epoch->handle,
                    sizeof(old)));
        epoch->handle = body->handle;
    }
}
```

6. void mdc_commit_delayed(struct ptlrpc_request *req)

   The generic commit callback. It detaches the current request from the mdc_open_data, removes it from the sequence list.

   Checks if this is the sequence end, if so walk through the sequence list, remove all the requests from it, clearing replay-delayed flag on them to let these requests to be removed from the replay queue.

   It also deallocates mdc_open_data if no other requests are left in the request sequence list.

```
void mdc_commit_delayed(struct ptlrpc_request *req)
{
    struct mdc_open_data *mod = req->rq_cb_data;
    struct ptlrpc_request *cur, *tmp;
    DEBUG_REQ(D_HA, req, "req committed");
    if (mod == NULL)
        return;
```

```
        req->rq_cb_data = NULL;
        req->rq_commit_cb = NULL;
        list_del_init(&req->rq_mod_list);
        if (req->rq_sequence) {
            list_for_each_entry_safe(cur, tmp, &mod->mod_replay_list,
                                     rq_mod_list)
            {
                LASSERT(cur != LP_POISON);
                LASSERT(cur->rq_type != LI_POISON);
                DEBUG_REQ(D_HA, cur, "req balanced");
                LASSERT(cur->rq_transno != 0);
                LASSERT(cur->rq_import == req->rq_import);
                list_del_init(&cur->rq_mod_list);
                spin_lock(&cur->rq_lock);
                cur->rq_replay = 0;
                spin_unlock(&cur->rq_lock);
            }
        }
        if (list_empty(&mod->mod_replay_list)) {
            if (mod->mod_och != NULL)
                mod->mod_och->och_mod = NULL;
            OBD_FREE_PTR(mod);
        }
    }
```

5.4 MDT.

1. MDT_CHECK_RESENT(req, reconstruct)

   A macro that checks if this is a MGS_RESENT request, calls the given reconstructor and returns the proper return value.

```
    if (lustre_msg_get_flags(req->rq_reqmsg) & MSG_RESENT) {          \
        struct mdt_client_data *mcd =                                 \
            req->rq_export->exp_mdt_data.med_mcd;                     \
        if (le64_to_cpu(mcd->mcd_last_xid) == req->rq_xid) {          \
            reconstruct;                                              \
            RETURN(le32_to_cpu(mcd->mcd_last_result));                \
        }                                                             \
        if (le64_to_cpu(mcd->mcd_last_close_xid) == req->rq_xid) {    \
            reconstruct;                                              \
            RETURN(le32_to_cpu(mcd->mcd_last_close_result));          \
        }                                                             \
        DEBUG_REQ(D_HA, req, "no reply for RESENT req (have "LPD64")",\
                  mcd->mcd_last_xid);                                 \
```

```
}
```

2. int mdt_close(struct mdt_thread_info *info)

   Check if this is the RESENT RPC and if its xid equals to the last handled RPC one. If so, reconstruct the CLOSE reply.

```
MDT_CHECK_RESENT(req, mdt_reconstruct_generic(info, NULL));
```

3. int mdt_done_writing(struct mdt_thread_info *info)

   Check if this is the RESENT RPC and if its xid equals to the last handled RPC one. If so, reconstruct the CLOSE reply.

```
MDT_CHECK_RESENT(req, mdt_reconstruct_generic(info, NULL));
```

4. static void mdt_reconstruct_setattr(struct mdt_thread_info *mti, struct mdt_lock_handle *lhc)

   Check if this SETATTR opens IO epoch, if so find the proper mfd in the mdt_export_data and put the proper file handle into the reconstructed reply.

```
if (mti->mti_epoch->flags & MF_EPOCH_OPEN) {
    struct mdt_file_data *mfd;
    struct mdt_body *repbody;
    repbody = req_capsule_server_get(&mti->mti_pill, &RMF_MDT_BODY);
    repbody->ioepoch = obj->mot_ioepoch;
    spin_lock(&med->med_open_lock);
    list_for_each_entry(mfd, &med->med_open_head, mfd_list) {
        if (mfd->mfd_xid == req->rq_xid)
            break;
    }
    LASSERT(&mfd->mfd_list != &med->med_open_head);
    spin_unlock(&med->med_open_lock);
    repbody->handle.cookie = mfd->mfd_handle.h_cookie;
}
```

5.5 LibLustre.

1. static int llu_md_close(struct obd_export *md_exp, struct inode *inode)

   Check if SETATTR is to be sent and mark the CLOSE RPC as the sequence end if not.

```
rc = md_close(md_exp, op_data, och->och_mod, &req);
if (rc != -EAGAIN)
    seq = req;
...
if (seq)
    ptlrpc_close_replay_seq(req);
```

2. int llu_sizeonmds_update(struct inode *inode, struct mdc_open_data *mod, struct lustre_handle *fh, __u64 ioepoch)

   The new parameter @mod is given, pass it to ll_md_setattr.

   If llu_inode_getattr() returns -ENOENT, zero obdo attributes before proceeding, this SOM attribute update with no valid attributes propagates the error on the server to let it clean the SOM related structures on the inode after unlink.

```
rc = llu_inode_getattr(inode, oa);
if (rc == -ENOENT) {
    oa.o_valid = 0;
    CDEBUG(D_INODE, "objid "LPX64" is already destroyed\n",
           lli->lli_smd->lsm_object_id);
}
...
rc = llu_md_setattr(inode, op_data, &mod);
```

3. int llu_md_setattr(struct inode *inode, struct md_op_data *op_data, struct mdc_open_data **mod)

   The new parameter @mod is given, it is a double pointer to mdc_open_data, pass it to ll_md_setattr().

```
rc = md_setattr(sbi->ll_md_exp, op_data, NULL, 0, NULL,
                0, &request, mod);
```

4. int llu_setattr_raw(struct inode *inode, struct iattr *attr)

   mdc_setattr() on truncate is supposed to allocate a new mdc_open_data because this SETATTR is the first RPC in the sequence. Pass a double pointer to this structure to llu_md_setattr() and pass it later to llu_setattr_done_writing().

```
rc = llu_md_setattr(inode, op_data, &mod);
...
if (op_data.op_ioepoch)
    rc1 = llu_setattr_done_writing(inode, &op_data, mod);
```

5. static int llu_setattr_done_writing(struct inode *inode, struct md_op_data *op_data, struct mdc_open_data *mod)

   The new parameter @mod is given, pass it to md_done_writing() and llu_sizeonmds_update().

```
rc = md_done_writing(ll_i2sbi(inode)->ll_md_exp, op_data, mod);
if (rc == -EAGAIN) {
    rc = llu_sizeonmds_update(inode, mod, &op_data->op_handle,
                              op_data->op_ioepoch);
}
```

5.6 PTLRPC.

1. struct ptlrpc_request * ptlrpc_prep_req_pool(struct obd_import *imp, __u32 version, int opcode, int count, int *lengths, char **bufs, struct ptlrpc_request_pool *pool, struct ptlrpc_cli_ctx *ctx)

   Initialize the sequence list.

```
CFS_INIT_LIST_HEAD(&request->rq_mod_list);
```

6 Protocol, API, disk format.

No changes.

7 Scalability and performance.

No changes.

8 Test plan.

The test is out of scope of this document and will be elaborated as a separate task.

9 Focus on inspection.