Version-Based Recovery DLD

Mike Pershin

2008-04-15

1 Requirements

1. a server only replays a client update request when the update affects exactly the same versions of objects as it did in the original execution;
2. a client is guaranteed that all information that it has obtained from the server is present after recovery;
3. delayed recovery possibility;
4. disconnected operations support.

2 Functional Specification

2.1 Definitions:

version: {`epoch, transno`} pair labeling objects (MDT inodes, OST objects), where `epoch` is boot cycle sequence number and `transno` is last transaction of the object.

preop-version: the version of the inodes when it have just been locked.

postop-version: the version of the inodes as set by the operation

primary recovery stage: first recovery stage, when server waits for all clients to reconnect and tries to replay them in transaction order (used now as only recovery way)

delayed recovery stage: recovery of clients which weren't able to reconnect in time and doing that later.

commit-on-sharing: technique that helps to avoid dependencies during recovery by doing commit if some info is going to be shared between two nodes.

2.2 Basic structures

2.2.1 Version

Version of object is server maintained pair {epoch, transaction}. It is stored as __u64 value where the epoch is highest 32 bits and transno is lowest 32 bit.

```
#define LU_EPOCH_BITS 32
#define lu_ver_epoch(a) (a >> LU_EPOCH_BITS)
#define lu_ver_transno(a) (a & (~0UL >> LU_EPOCH_BITS)
```

2.2.2 Set/get version

There are two methods in Lustre for getting/setting inode versions:

```
__u64 fsfilt_ext3_get_version(struct inode *inode);
```

```
__u64 fsfilt_ext3_set_version(struct inode *inode, __u64 new_version);
```

The Lustre 1.8 will need the similar method to support inode version handling in OSD and new method in lu_device API for that.

2.2.3 RPC buffers

New buffer is added to several RPCs to provide version support. It contains preop and postop versions for objects involved in operation. There is only one post version because all objects involved in operation were updated in the same transaction therefore they have the same postop version.

```
struct lu_versions {
        __u64 lv_post_version;
        __u64 lv_pre_versions[4];
};
extern void lustre_swab_version(struct lu_version *lv);
```

Reply message contains additional buffer with versions.

These versions are saved to original request for replay. The simplest way is to add additional buffer at the end of request so we don't need to allocate it after reply for replay. MDC code should be modified with adding new buffer for reint requests and open.

2.2.4 Last_rcvd

Client data in *last_rcvd* contains not only last_transno (which is actually postop version now) but also preop version to reconstruct reply fully:

```
struct mds_client_data {
        __u8 mcd_uuid[40];      /* client UUID */
        __u64 mcd_last_version; /* last completed version */
        __u64 mcd_pre_versions[4]; /* preop versions for objects in last operation */
        __u64 mcd_last_xid;     /* xid for the last transaction */
        ...
        __u8 mcd_padding[LR_CLIENT_SIZE - 120];
};
```

2.3 Version handling

2.3.1 Version and epoch maintaining

The epoch is maintained by server:

- the server data in the *last_rcvd* file contains the `last_version` instead of `last_transno;`
- the epoch is got from `last_version` and increased upon each boot cycle in `mds_init_server_data();`
- the server transno is started from 1 after each epoch increase. This allows to don't know the last used transaction before recovery and server doesn't depend on disconnected clients.

2.3.2 Updating the version

The versions of objects involved in operation are get during operation and stored in reply buffer. The new version is the transaction assigned for current operation, therefore in should be set during `mds_finish_transno()` but before the transaction is stopped.

The `mds_finish_transno()` should be changed to receive as parameter not only one inode but all inodes involved in operation so their versions can be updated under the same lock as used to protect transaction:

```
static inline void mds_versions_set(struct obd_device *obd,
                                    struct inode **inodes, __u64 version)
{
        if (inodes == NULL)
                return;
        fsfilt_set_version(obd, inodes[0], version);
        fsfilt_set_version(obd, inodes[1], version);
        fsfilt_set_version(obd, inodes[2], version);
        fsfilt_set_version(obd, inodes[3], version);
}
static inline void mds_versions_get(struct obd_device *obd,
                                    struct lu_version *lv,
                                    struct inode **inodes)
{
        if (inodes == NULL)
                return;
        lv->lv_pre_ver[0] = fsfilt_get_version(obd, inodes[0]);
        lv->lv_pre_ver[1] = fsfilt_get_version(obd, inodes[1]);
        lv->lv_pre_ver[2] = fsfilt_get_version(obd, inodes[2]);
        lv->lv_pre_ver[3] = fsfilt_get_version(obd, inodes[3]);
}
int mds_finish_transno(struct mds_obd *mds, struct inode **inodes, void *handle,
                       struct ptlrpc_request *req, int rc, __u32 op_data)
{
...
                spin_lock(&mds->mds_transno_lock);
                transno = ++mds->mds_last_transno;
                /* versions */
                lv->lv_post_ver = transno;
                if (op_data == 0) /* not open */
                        mds_versions_set(obd, lv, inodes, transno);
                spin_unlock(&mds->mds_transno_lock);
...
                /* save versions in last_rcvd for reconstruct */
                if (lv) {
                        mcd->mcd_last_version = cpu_to_le64(lv->lv_post_ver);
                        mcd->mcd_pre_versions[0] = cpu_to_le64(lv->lv_pre_ver[0]);
                        mcd->mcd_pre_versions[1] = cpu_to_le64(lv->lv_pre_ver[1]);
                        mcd->mcd_pre_versions[2] = cpu_to_le64(lv->lv_pre_ver[2]);
                        mcd->mcd_pre_versions[3] = cpu_to_le64(lv->lv_pre_ver[3]);
                }
}
```

Note: `fsfilt_set_version()` should mark inode dirty to ensure that changes will be committed.

Note: getting the versions can be done not in `mds_finish_transno()` but in caller function though we use `mds_finish_transno()` just to avoid code duplicating across reint functions.

Lustre 1.8 The new version is set in `mdt_txn_stop_cb()`. Right before setting new version the old one should be saved as preop version in reply. This is important for parent directory mostly because it can be used by several threads at once due to pdirops feature, therefore the only way to get correct preop version is reading it right before writting the new one because that is serialized.

2.4 Version based recovery

The two requirements should be met:

1. During replay of requests version-based recovery will allow clients to replay if and only if the objects the client is using have exactly the same version as during the original execution.
2. After replay, recovery can complete successfully if the data *clients* obtained from servers before recovery is assured to be present and not to have rolled back. A key issue, subject to policy is if, *client* designates the *client Lustre file system* or *client applications* using the file system.

If both conditions can be met jobs can continue without errors. If the conditions cannot be met, eviction will be the normal result, but more relaxed recovery options can be made available to the client.

2.4.1 Normal operations

1. For any operation with transaction the server does the following:
   (a) the preop versions of objects are written to `struct lu_version` in extra reply buffer
   (b) the new version is set after assigning the transno and stored in `struct lu_version` also
   (c) last_rcvd record is updated with versions too

   Note: open shouldn't set new version upon getting new transno, but should get preop versions and save them in reply for version checking. Close() doesn't get or set versions.
2. For all other operations the server determines the `highest_used_version` - the biggest one for objects involved in operation - and send it back to the client. It will be used to determine does client depends on gap or not.
3. server replies to the client with version info
4. Client gets reply and:
   (a) in `after_reply()` client saves the `ll_version` into the original request for replay
   (b) compares `highest_version` from reply with client `highest_version` value to determine new value.

2.4.2 Version checking

Each replayed operation checks version. That should be done after all involved objects are locked, the preop versions are got here also:

```
static inline int mds_version_get_check(struct ptlrpc_request *req,
                                        struct inode *inode,
                                        __u64 ver_old,
                                        __u64 *ver_new)
{
        struct obd_device *obd = req->rq_export->exp_obd;
        LASSERT(inode);
        *ver_new = fsfilt_get_version(obd, inode);
        if (lustre_msg_get_flags(req->rq_reqmsg) & MSG_REPLAY) {
                if (ver_old != *ver_new)
```

```
                        RETURN (-ENOTSYNC);
        }
        RETURN(0);
}
mds_reint_setattr()
{
...
        rc = mds_version_get_check(req, inode, rec->ur_lv->lv_pre_ver[0],
                                   &lv->lv_pre_ver[0]);
        if (rc)
                GOTO(cleanup, rc);
...
}
```

2.4.3 Primary recovery phase

1. server completes its initialization and starts to accept connecting request from clients. The server, as before goes into recovery mode if old exports are found.
2. server checks the last_committed epoch per export to decide is it stale export or it was participating in previous epoch.
3. server waits `OBD_RECOVERY_TIMEOUT`seconds to allow clients to connect, those clients which connected with server are counted as normal recovery clients. Each time a client connects the recovery connect window is grown up to a maximum value. `target_start_recovery_timer()`should extend timeout upon each new connection.
4. In the connection handshake the server reports to the client what the last transaction is that it has committed.
5. The server begins to receive replay requests from clients, if the transno of the request is in the right order it continues to process it.
6. the `process_recovery_queue()` wait for next transno and upon timeout don't abort recovery but write down the gap info and do the following:
   (a) waits at least `obd_timeout` to allow other clients to join in and close the gap
   (b) the replay continues with version checking for integration. Mismatched replays will be answered with -ENOTSYNC but all replays from client should be processed in any case.
7. when server completes recovery all obd_exports for still disconnected clients are retained with the last_transaction committed value and gap informanion for delayed replay.

Optional: the server can avoid ordering all clients by transactions. Instead it can allow client to replay out-of-order if there is version match. Otherwise if version is smaller than needed then server put this request to the waiting queue until needed transaction will be replayed. That can lead to faster recovery.

Note:

1. While no gap has been encountered it is not necessary to check versions, but it is not harmful and Lustre will check versions always.
2. **Optional:** If a process has an open file handle or active file system lock and versions for the inodes do not match, then the client can kill the process if it receives an -ENOTSYNC message.

2.4.4 Delayed recovery phase

1. A client connects to the server which has already completed recovery.
2. the server do the following:
   (a) finds an export for the client;
   (b) set connection flag `MSG_CONNECT_DELAYED`;
   (c) answer with last_version value and the gap information if present. The last_version is actual last committed value for that client, the obd_last_committed cannot be used during delayed connection.
3. The clients sends its replay requests.
4. All the replay requests will replay based on version checking. Requests will get NEW server transaction numbers but versions in reply remain unchanged.
5. Server don't use `target_queue_recovery_request()` but process requests directly, because this recovery is from only one client and all requests are already ordered by transno.
6. Client doesn't replay locks.
7. The last_committed is not sent to the client during replay phase. Only the last_version is reported during connect as transno to start replay with.
8. The client performs an eviction in case of version mismatches or completes recovery.

Problem: delayed client will not be able to open unlinked files because they will be deleted after primary recovery phase, so orphans shouldn't be cleared while any delayed export presents.

Problem: during recovery all operations are serialized, so recovery code rely on that but it will be not so during delayed recovery phase. Recovery code paths should be reviewed keeping that in mind

2.4.5 Aborted delayed recovery

If delayed recovery was aborted due to server failure then the client will have request from old epoch and from new one in the replay queue. These requests shouldn't be dropped by last_committed at the next recovery attempt. This can be avoided by using both version and transaction

version - original version of operation to be replayed. This is used for ordering the replays and stored as version of object at server.

transaction - transaction number used during replay. This is needed to know when we should remove this request from replay queue.

Client replay queue states can be represented by next variants:

1. all replays are from the same epoch.
2. with old replays there are also several new which got reply from server during aborted recovery

Replay process during primary recovery phase:

1. Got last_committed_transno and last_committed_version from server during connect.
2. drop all replays with version less than last_committed_version because they were committed
3. replay all requests with transno >= last_committed_transo. These requests were replayed at last time and other clients may depend on them.
4. all remaining requests are old, from some previous epoch and should be replayed as delayed recovery replays after main recovery will finish.

2.4.6 Gap handling

1. The server maintains the `first_gap_transno` which is updated after each replay and supply the client with it during recovery.

   (a) For any unconnected clients in primary phase the server writes gap info into corresponding records in last_rcvd file when primary phase is complete, so client will be able to know gap data during delayed recovery.

   (b) During recovering of such client the gap can be shrank and should be updated for remaining disconnected clients. It is possible that after several failures there can be gaps from different epochs, so only gaps with the same value of epoch should be updated.

2. Every client maintains the `highest_used_transno` and compare it with `first_gap_transno`. If the `highest_used_transno` is earlier than the `first_gap_transno`, the client is up to date fully.
3. Otherwise:

   (a) the client compares that each of its cached objects newer than the `first_gap_transno`. If their versions are equal or newer then client can recover completely.

   (b) Finally the client can purge its cache and repeat the previous step for the cached objects that are in use. If this test succeed the client performs *weak file system recovery.*

4. **Optionally** the client can request eviction if full node recovery or full file system recovery cannot succeed.
5. **Optionally -** If the client can record in each process what the newest object is that the process may have depended on in a system call, then the client can determine which processes have seen stale data, although processes may have exited already and seen stale data. Optionally all processes that have seen stale data could be killed.

2.4.7 obd_export maintenance

To maintain `last_committed`for not-connected clients the info about non-connecting client should be retained.

1. Export information for disconnected clients can be cleaned up by an lctl command
2. If a client at a NID connects with a new UUID any old export can be cleaned up
3. no more than `MAX_DISCONN_EXPORTS`of disconnected exports are retained
4. Until there is disconnected exports the orphans should be preserved to keep open-unlink inodes intact
5. Each export keep information about gap info in the last epoch when this client was connected. Also export keep information about recovery stage type - primary or delayed to determine what kind of recovery should be done.

2.5 Compatibility

Version recovery with commit on sharing should appear in both Lustre 1.6 and 1.8. Considering their differences in many areas the current design has some limits and restrictions.

2.5.1 Pdirops and preop version

Pdirops feature allows many threads to access the same directory based on locking per name_hash. That means that preop version can be the same for two concurrent process if versions are taken before any of process write new versions. In that case the preop version can be taken right before setting new one, this is done under transno lock and serialized for all concurrent threads.

3 Use Cases

3.1 Versions

3.1.1 Version setting

New version should be set in following operations:

- create - for parent and new child
- link - for parent and child
- unlink - for parent and child
- setattr(setxattr) - for object, except changing time and size
- rename - for all objects involved

3.1.2 Version getting

The version of objects should be get before operation and returned to the caller in reply.

For getattr/open operations the version of objects involved in operation should be returned too, so client can know the latest 'used' version (so the transaction)

3.1.3 Version checking

Versions should be checked during replay:

- during primary recovery phase if gap is encountered
- during delayed recovery phase

3.2 Primary recovery phase

3.2.1 Normal recovery, all clients are connected in time

Versions are checked during normal recovery and there should be not version mismatches

3.2.2 One client is not connected and gap presents

1. clients has both `last_committed_transno` and `highest_used_trasno` which are less than `first_gap_transno` and should recover fully
2. client `last_committed_transno` less than than `first_gap_transno` but `highest_used_transno` is bigger than `first_gap_trans`
   Such client should recover without errors but
   (a) all objects with version higher than `first_gap_transno` should be dropped from cache and client can complete recovery.
   (b) if some of them are in use than client should evict
3. client's `last_committed_transno` is bigger than `first_gap_transno`. Recovery proceed with version checking
   (a) client encounters no version mismatches and complete recovery as described in step 2 above.
   (b) client found version mismatch but recovery should finish and client should evict at the end.

3.3 Delayed recovery phase

3.3.1 Export and recovery data states

All information about client should be preserved on server after primary recovery phase - latest export data and gap information

3.3.2 Replay stage

1. Committed open replay should work is version is matched and unlinked orphans should be available
2. All replays have version matched and client is recovering with same process as in primary phase
3. version mismatch leads to client eviction after recovery
4. locks are not replayed but canceled

3.3.3 Delayed client reconnects during primary recovery phase

The timeout occurred and primary recovery has been started with gap and version checking, then delayed client reconnects while primary recovery is still in progress. The new client should be integrated in recovery process.

3.3.4 Version getting

The preop version can't be got after locking because it can be the same for different threads

4 Logic Specification

4.1 Version handling

4.1.1 mds_versions_get

Get current version

```
static inline void mds_versions_get(struct obd_device *obd,
                                    struct lu_version *lv,
                                    struct inode **inodes)
{
        if (inodes == NULL)
                return;
        lv->lv_pre_ver[0] = fsfilt_get_version(obd, inodes[0]);
        lv->lv_pre_ver[1] = fsfilt_get_version(obd, inodes[1]);
        lv->lv_pre_ver[2] = fsfilt_get_version(obd, inodes[2]);
        lv->lv_pre_ver[3] = fsfilt_get_version(obd, inodes[3]);
}
```

4.1.2 mds_versions_set

```
static inline void mds_versions_set(struct obd_device *obd,
                                    struct inode **inodes, __u64 version)
{
        if (inodes == NULL)
                return;
        fsfilt_set_version(obd, inodes[0], version);
        fsfilt_set_version(obd, inodes[1], version);
        fsfilt_set_version(obd, inodes[2], version);
        fsfilt_set_version(obd, inodes[3], version);
}
```

4.1.3 mds_versions_get_check

```
static inline int mds_version_get_check(struct ptlrpc_request *req,
                                        struct inode *inode,
                                        __u64 ver_old,
                                        __u64 *ver_new)
{
        struct obd_device *obd = req->rq_export->exp_obd;
        LASSERT(inode);
        *ver_new = fsfilt_get_version(obd, inode);
        if (lustre_msg_get_flags(req->rq_reqmsg) & MSG_REPLAY) {
                if (ver_old != *ver_new)
                        RETURN (-EOVERFLOW);
        }
        RETURN(0);
}
```

4.1.4 mds_finish_transno

```
int mds_finish_transno(struct mds_obd *mds, struct inode **inodes, void *handle,
                       struct ptlrpc_request *req, int rc, __u32 op_data)
{
        ...
        struct inode *inode = inodes ? inodes[0] : NULL;
        struct lu_version *lv = NULL;
        int version_set = handle ? 1 : 0;

        ...

        /* Version Based Recovery */
        if (inodes)
                lv = mds_rep2ver(req);
        ...
        transno = lustre_msg_get_transno(req->rq_reqmsg);
        if (rc != 0) {
        ...
        } else if (transno == 0) {
                spin_lock(&mds->mds_transno_lock);
                transno = ++mds->mds_last_transno;
                /* versions */
                if (lv) {
                        lv->lv_post_ver = transno;
                        if (version_set)
                                mds_versions_set(obd, inodes, transno);
                }
                spin_unlock(&mds->mds_transno_lock);
        } else {
                spin_lock(&mds->mds_transno_lock);
                if (transno > mds->mds_last_transno)
                        mds->mds_last_transno = transno;
                /* replay case. So preop versions are checked already
                 * and lv is filled from request too. Set new versions only */
                mds_versions_set(obd, inodes, transno);
                spin_unlock(&mds->mds_transno_lock);
        }
        req->rq_transno = transno;
        lustre_msg_set_transno(req->rq_repmsg, transno);
        if (lustre_msg_get_opc(req->rq_reqmsg) == MDS_CLOSE) {
                prev_transno = le64_to_cpu(mcd->mcd_last_close_transno);
                mcd->mcd_last_close_transno = cpu_to_le64(transno);
                mcd->mcd_last_close_xid = cpu_to_le64(req->rq_xid);
                mcd->mcd_last_close_result = cpu_to_le32(rc);
                mcd->mcd_last_close_data = cpu_to_le32(op_data);
        } else {
                prev_transno = le64_to_cpu(mcd->mcd_last_transno);
                /* save versions in last_rcvd for reconstruct */
                if (lv) {
                        mcd->mcd_last_version = cpu_to_le64(lv->lv_post_ver);
                        mcd->mcd_pre_versions[0] = cpu_to_le64(lv->lv_pre_ver[0]);
                        mcd->mcd_pre_versions[1] = cpu_to_le64(lv->lv_pre_ver[1]);
                        mcd->mcd_pre_versions[2] = cpu_to_le64(lv->lv_pre_ver[2]);
                        mcd->mcd_pre_versions[3] = cpu_to_le64(lv->lv_pre_ver[3]);
                }
                mcd->mcd_last_transno = cpu_to_le64(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);
        }
        ...
}
```

4.2 Versions in RPCs

4.2.1 pack versions

Each operation that need versions is changed to keep additional buffer in both request and reply. There is example with `setattr()`:

```
int mdc_setattr(struct obd_export *exp, struct mdc_op_data *data,
                struct iattr *iattr, void *ea, int ealen, void *ea2, int ea2len,
                struct ptlrpc_request **request)
{
        ...
        int size[5] = { sizeof(struct ptlrpc_body),
                        sizeof(*rec), ealen, ea2len };
        int bufcount = 2, rc;
        ...
        /* version recovery */
        size[bufcount++] = sizeof(struct lu_version);
        req = ptlrpc_prep_req(class_exp2cliimp(exp), LUSTRE_MDS_VERSION,
                              MDS_REINT, bufcount, size, NULL);
        ...
        size[REPLY_REC_OFF] = sizeof(struct mds_body);
        /* versions of objects */
        size[REPLY_REC_OFF + 1] = sizeof(struct lu_version);
        ptlrpc_req_set_repsize(req, 3, size);
        rc = mdc_reint(req, rpc_lock, LUSTRE_IMP_FULL);
        ...
}
```

4.2.2 unpack versions

Versions are unpacked and saved in `mds_update_record`. This is needed only for recovery so replay flag is checked.

```
static inline int mds_version_unpack(struct ptlrpc_request *req, int offset,
                                     struct mds_update_record *r)
{
        if (lustre_msg_get_flags(req->rq_reqmsg) & MSG_REPLAY) {
                if (lustre_msg_bufcount(req->rq_reqmsg) <= offset) {
                        CERROR("No versions in replay\n");
                        RETURN (-EFAULT);
                }
                r->ur_lv = lustre_swab_reqbuf(req, offset,
                                              sizeof(struct lu_version),
                                              lustre_swab_version);
                if (r->ur_lv == NULL) {
                        CDEBUG(D_ERROR, "no version\n");
                        RETURN (-EFAULT);
                }
        }
        RETURN (0);
}
```

4.2.3 get versions buffer in reply

```
static inline struct lu_version *mds_rep2ver(struct ptlrpc_request *req)
{
        struct lu_version *lv = NULL;
        int offset = lustre_msg_bufcount(req->rq_repmsg) - 1;
        if (lustre_msg_buflen(req->rq_repmsg, offset) > 0) {
                LASSERT(lustre_msg_buflen(req->rq_repmsg, offset) ==
                        sizeof(struct lu_version));
                CWARN("Version buf offset is %i\n", offset);
                lv = lustre_msg_buf(req->rq_repmsg, offset,
                                    sizeof(struct lu_version));
        }
        return lv;
}
```

4.2.4 save versions from reply in request for replay

```
static void ptlrpc_save_versions (struct ptlrpc_request *req)
{
        struct lustre_msg *repmsg = req->rq_repmsg;
        struct lustre_msg *reqmsg = req->rq_reqmsg;
        int rq_off = lustre_msg_bufcount(reqmsg) - 1;
        int rp_off = lustre_msg_bufcount(repmsg) - 1;
        struct lu_version *lv, *lvp;
        ENTRY;
        if (lustre_msg_buflen(reqmsg, rq_off) != sizeof(struct lu_version)) {
                DEBUG_REQ(D_ERROR, req,
                          "Wrong buffer %i (%i vs %i) for version to save\n",
                          rq_off, lustre_msg_buflen(reqmsg, rq_off),
                          sizeof(struct lu_version));
                return;
        }
        if (lustre_msg_buflen(repmsg, rp_off) != sizeof(struct lu_version)) {
                DEBUG_REQ(D_ERROR, req,
                          "Wrong buffer %i (%i vs %i) with versions\n",
                          rp_off, lustre_msg_buflen(repmsg, rp_off),
                          sizeof(struct lu_version));
                return;
        }
        lv = lustre_msg_buf(reqmsg, rq_off, sizeof(struct lu_version));
        lvp = lustre_msg_buf(repmsg, rp_off, sizeof(struct lu_version));

        *lv = *lvp;
        EXIT;
}
static int after_reply(struct ptlrpc_request *req)
{
        ...
        /* Store transno in reqmsg for replay. */
        req->rq_transno = lustre_msg_get_transno(req->rq_repmsg);
        lustre_msg_set_transno(req->rq_reqmsg, req->rq_transno);
        if (req->rq_import->imp_replayable) {
                spin_lock(&imp->imp_lock);
                /* no point in adding already-committed requests to the replay
                 * list, we will just remove them immediately. b=9829 */
                if (req->rq_transno != 0 &&
                    (req->rq_transno >
                     lustre_msg_get_last_committed(req->rq_repmsg) ||
                     req->rq_replay))
                        /* version recovery */
                        ptlrpc_save_versions(req);
                        ptlrpc_retain_replayable_request(req, imp);
                else if (req->rq_commit_cb != NULL) {
                        ...
                }
}
```

4.3 Recovery

4.3.1 Primary recovery phase

```
static void target_recovery_expired(unsigned long castmeharder)
{
        struct obd_device *obd = (struct obd_device *)castmeharder;
        CERROR("%s: recovery timed out, aborting\n", obd->obd_name);
        spin_lock_bh(&obd->obd_processing_task_lock);
        /* version recovery */
        if (obd->obd_recovering) {
                obd->obd_version_recovery = 1;
                obd->obd_gap_transno = obd->obd_next_recovery_transno;
        }
        cfs_waitq_signal(&obd->obd_next_transno_waitq);
        spin_unlock_bh(&obd->obd_processing_task_lock);
}
static int check_for_next_transno(struct obd_device *obd)
{
        struct ptlrpc_request *req;
        ...
        CDEBUG(D_HA,"max: %d, connected: %d, completed: %d, queue_len: %d, "
               "req_transno: "LPU64", next_transno: "LPU64"\n",
               max, connected, completed, queue_len, req_transno, next_transno);
        if (obd->obd_abort_recovery) {
                ...
        } else if (obd->obd_version_recovery) {
                /* skip checking for next_transno,
                 * just take next one in queue */
                obd->obd_next_recovery_transno = req_transno;
                wake_up = 1;
        }
        spin_unlock_bh(&obd->obd_processing_task_lock);
        LASSERT(lustre_msg_get_transno(req->rq_reqmsg) >= next_transno);
        return wake_up;
}
```

4.3.2 Delayed recovery phase

```
int target_handle_connect(struct ptlrpc_request *req, svc_handler_t handler)
{
         ...
         /* for delayed connections say they should start recovery */
         if (export->exp_delayed) {
                 lustre_msg_add_op_flags(req->rq_repmsg, MSG_CONNECT_DELAYED);
                 target_start_recovery_timer(target, handler);
         }
         ...
}
int ldlm_replay_locks(struct obd_import *imp)
{
        struct ldlm_namespace *ns = imp->imp_obd->obd_namespace;
        struct list_head list;
        struct ldlm_lock *lock, *next;
        int rc = 0;
        ENTRY;
        CFS_INIT_LIST_HEAD(&list);
        LASSERT(atomic_read(&imp->imp_replay_inflight) == 0);
        /* ensure this doesn't fall to 0 before all have been queued */
        atomic_inc(&imp->imp_replay_inflight);
        (void)ldlm_namespace_foreach(ns, ldlm_chain_lock_for_replay, &list);
        list_for_each_entry_safe(lock, next, &list, l_pending_chain) {
                list_del_init(&lock->l_pending_chain);
                if (imp->imp_delayed_recovery) {
                        ldlm_lock_cancel(lock);
                        continue;
                }
                if (rc)
                        continue; /* or try to do the rest? */
                rc = replay_one_lock(imp, lock);
        }
        atomic_dec(&imp->imp_replay_inflight);
        RETURN(rc);
}
```

4.4 Example of version handling/checking

4.4.1 Unlink operation

```
static int mds_reint_unlink(struct mds_update_record *rec, int offset,
                            struct ptlrpc_request *req,
                            struct lustre_handle *lh)
{
        ...
        struct inode *inodes[4];
        struct lu_version *lv = mds_rep2ver(req);
        ...
        rc = mds_get_parent_child_locked(obd, mds, rec->ur_fid1,
                                         &parent_lockh, &dparent, LCK_EX,
                                         MDS_INODELOCK_UPDATE,
                                         rec->ur_name, rec->ur_namelen,
                                         &child_lockh, &dchild, LCK_EX,
                                         MDS_INODELOCK_FULL);
        if (rc)
                GOTO(cleanup, rc);
        cleanup_phase = 1; /* dchild, dparent, locks */
        /* version recovery check for parent */
        LASSERT(lv);
        LASSERT(rec->ur_lv);
        rc = mds_version_get_check(req, dparent->d_inode,
                                   rec->ur_lv->lv_pre_ver[0],
                                   &lv->lv_pre_ver[0]);
        if (rc)
                GOTO(cleanup, rc);
        dget(dchild);
        child_inode = dchild->d_inode;
        if (child_inode == NULL) {
                CDEBUG(D_INODE, "child doesn't exist (dir %lu, name %s)\n",
                       dparent->d_inode->i_ino, rec->ur_name);
                GOTO(cleanup, rc = -ENOENT);
        }
        ...
        /* version recovery check */
        LASSERT(lv);
        LASSERT(rec->ur_lv);
        rc = mds_version_get_check(req, child_inode,
                                   rec->ur_lv->lv_pre_ver[1],
                                   &lv->lv_pre_ver[1]);
        if (rc)
                GOTO(cleanup, rc);
        ...

cleanup:
        if (rc == 0) {
                struct iattr iattr;
                int err;
                iattr.ia_valid = ATTR_MTIME | ATTR_CTIME;
                LTIME_S(iattr.ia_mtime) = rec->ur_time;
                LTIME_S(iattr.ia_ctime) = rec->ur_time;
                err = fsfilt_setattr(obd, dparent, handle, &iattr, 0);
                if (err)
                        CERROR("error on parent setattr: rc = %d\n", err);
        }
        inodes[0] = dparent ? dparent->d_inode : NULL;
```

```
                inodes[1] = child_inode;
                inodes[2] = NULL;
                inodes[3] = NULL;
                rc = mds_finish_transno(mds, inodes, handle, req, rc, 0);
                if (!rc)
                        (void)obd_set_info_async(mds->mds_osc_exp, strlen("unlinked"),
                                                 "unlinked", 0, NULL, NULL);
                ...
        }
```

1.

5 State Management

5.1 Recovery changes

New recovery stage is added - delayed recovery which make possible the full or partial recovery of liblustre clients and any others. Primary doesn't stop if gap in transactions occurs and keep stale exports untouched

5.2 Disk format changes

Inode will store version on disk, it is scope of another HLD

5.3 Wire format changes

Request and reply structures are changed and should contain version fields for all objects involved in operation

5.4 Old clients compatibility

Old clients are able to connect but will be unable to recover delayed - they don't provide versions in reply so delayed recovery will fail. Another bad effect of old clients is more other clients will fail to recover if they dependent on operations from old client. The version recovery will fail too. Therefore the only side-effect of old clients is less possibility to recover successfully for other clients.